Bounded-Bypass Mutual Exclusion with Minimum Number of Registers

Sheng-Hsiung Chen and Ting-Lu Huang

Abstract—A mutual exclusion mechanism that is both fair and space efficient can be highly valuable for shared memory systems under time and memory constraints such as embedded real-time systems. Several algorithms that utilize only one shared variable and guarantee a certain level of fairness have been proposed. However, these use hypothetical read-modify-write operations that have never been implemented in any system. This paper presents two fair algorithms that do not use such operations, each of which uses a single additional shared variable. The proposed algorithms employ commonly available operations, *fetch*&*store* and *read/write*, on two shared variables. The first algorithm satisfies the bounded-bypass condition. The second is an improvement on the first that satisfies the FIFO condition, which is the most stringent fairness condition. Additionally, it is shown that achieving the bounded-bypass condition using the same set of operations requires two shared variables. Both of the algorithms are thus optimal with respect to the number of shared variables.

Index Terms—Mutual exclusion, shared memory systems, space complexity, *fetch*&*store*, swap, bounded bypassing, FIFO.

1 Introduction

Mutual exclusion [6] is fundamental in multiprocessing systems for managing access to a single indivisible resource. In mutual exclusion, a process accesses the resource within a distinct part of code known as the *critical region*. A process executes trying and exit regions, respectively, before and after executing the critical region, to guarantee the following basic requirements:

- **Mutual Exclusion.** At most one process at a time is permitted to enter the critical region.
- **Progress.** If at least one process is in the trying region and no process is in the critical region, then at some later point, some process enters the critical region. In addition, if at least one process is in the exit region, then at some later point, some process enters the rest of the code, called the remainder region.

Embedded real-time systems, e.g., automotive control systems, mobile computing devices, and home electronics, have received increasing interest in recent years. An algorithm for such systems should consider time and memory constraints. The time constraint imposes a deadline for each process in executing a particular job because the process often interacts with users or a dynamic environment. Additionally, embedded systems often have small memory (about 32-64 kBytes) since minimizing production costs, weight, and power consumption is a primary concern in their designs [12], [17], [18]. As shown below, a mutual exclusion algorithm, in particular, should consider fairness and space efficiency.

Since a process can remain in the critical region for an arbitrarily long time, no algorithm can ensure that each waiting process will gain the permission to enter the critical region before its deadline. This creates an inherent difficulty in the mutual exclusion problem, especially for systems under the time constraint. Thus, algorithm designers attempt to improve the feasibility of mutual exclusion algorithms by designing them to grant the critical region fairly to each process. A mutual exclusion algorithm that satisfies the basic requirements may not guarantee such fairness. That is, a process may be indefinitely denied access to the critical region. Hence, the worst case waiting time may be infinite even when each process always returns the resource quickly. A fair mutual exclusion algorithm tries to reduce the worst-case waiting time by scheduling requests fairly, and thereby, improves the feasibility of the algorithm.

A space-efficient mutual exclusion algorithm largely focuses on reducing the memory consumption. This requirement is crucial for systems under the memory constraint. In terms of space complexity, most n-process mutual exclusion prior art algorithms use at least n shared variables, as shown in surveys by Anderson et al. [2] and Raynal [15]. For systems with limited memory, an algorithm using a constant number of shared variables would be more suitable.

This work proposes two fair and space-efficient mutual exclusion algorithms. A 2-bounded-bypass algorithm with two shared variables is first presented to show the basic idea. A first-in-first-out (FIFO) algorithm, which is based on the first algorithm, and uses the same number of shared variables, is then presented.

To define the bounded-bypass property, the trying region of each process is divided into two parts, a doorway and a waiting part. The property guarantees that a process that has completed its doorway cannot be bypassed more than a constant number of times by any other process. To be

- *S.-H. Chen is with the Physical Design Group, SpringSoft, Inc., No. 25, Industrial East 4th Rd., Hsin-Chu Science-Based Industrial Park, Hsin-Chu 300, Taiwan. E-mail: chenss@csie.nctu.edu.tw.*
- *T.-L. Huang is with the Department of Computer Science, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsin-Chu 300, Taiwan. E-mail: tlhuang@cs.nctu.edu.tw.*

Manuscript received 30 Aug. 2008; revised 9 May 2008; accepted 10 Feb. 2009; published online 13 Feb. 2009.
Recommended for acceptance by M. Yamashita.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-08-0302.
Digital Object Identifier no. 10.1109/TPDS.2009.28.

more specific, a mutual exclusion algorithm is defined to satisfy the b-bounded-bypass property if no process that has finished its doorway can be bypassed more than b times by any other process when competing for a resource. An algorithm is said to be bounded-bypass if it is b-bounded-bypass for some constant b. In addition, an algorithm is said to be FIFO if when a process i passes through its doorway before j performs a step in its doorway, then j cannot enter its critical region before i does so. It is clear that an FIFO algorithm is also bounded-bypass. A fair mutual exclusion algorithm that satisfies the bounded-bypass property should be provided for systems under the time constraint, in which a process can roughly estimate the waiting time. For instance, in the proposed 2-bounded-bypass algorithm, a process cannot be bypassed more than $2(n-1)$ times by other processes, where n denotes the number of all processes, after requesting the critical region. In contrast, a process might be bypassed without limitation in an algorithm that does not satisfy bounded-bypass, easily violating the deadline for executing a particular job.

In terms of space complexity, only two shared variables are utilized in each of the algorithms. Hence, no dynamic memory allocation is needed when executing the algorithm, so the system overhead is reduced. Since mutual exclusion is a basic synchronization mechanism frequently used in multiprocessing systems both in operating system kernel level and in users' application level [14], the system performance can be significantly improved.

Both the algorithms are implemented by *fetch&store* along with atomic *read* and *write* operations. Burns and Lynch [4] showed that n shared variables are necessary to solve the n-process mutual exclusion problem if only *read* and *write* are available. Fich et al. [8] recently extended the linear lower bound to systems that support *conditional* read-modify-write (RMW) operations, such as *compare&swap*. An RMW operation is said to be conditional provided that it changes the value of a variable in a single step only if the variable has a particular value. Hence, some operations other than *read/write* and conditional RMW operations are needed to decrease the space requirement. Operation *fetch&store* is adopted to implement the algorithms since it is commonly supported in modern microprocessors such as a series of processors of Intel and AMD, Motorola 88000, and SPARC [16], and is also available in the ARM processor family [1],[1] which is arguably the most popular embedded architecture today. Thus, *fetch&store* improves the portability of the algorithm.

Several algorithms that use only a single shared variable and guarantee a certain level of fairness have been presented. For instance, Fischer et al. [10] devised an FIFO algorithm, and Burns et al. [3] devised a bounded-bypass algorithm and a starvation-free algorithm. Unfortunately, all of these algorithms used hypothetical RMW operations that have not yet been implemented in any system. In contrast, none of the algorithms we propose use a hypothetical RMW operation, and each of them requires only one more shared variable than these algorithms.

The proposed algorithms are inspired by the circular list-based mutual exclusion algorithm presented by Fu and Tzeng [11]. The proposed algorithms, like that of Fu and Tzeng, organize waiting processes into lists, but pass the permission within and among lists very differently. Fu and Tzeng's algorithm may block a process in the exit region. However, the proposed algorithms eliminate this drawback. Whereas Fu and Tzeng reduced the number of remote memory accesses, our algorithms target the space complexity and guarantee a certain level of fairness.

Furthermore, this work proves that two shared variables are necessary to solve the problem of mutual exclusion with b-bounded-bypass for any constant b using only *fetch&store* and *read/write*. This impossibility result is proven by showing a more general result that two object instances are required to implement a bounded-bypass mutual exclusion algorithm when using only historyless objects regardless of the size of the objects. The definition of a historyless object is given by Fich et al. [9] and is restated in Section 5. According to the definition, shared variables associated with *fetch&store* and *read/write* belong to the class of historyless objects, so the more general result implies the proposed algorithms are optimal with respect to the number of shared variables. Informally, an object is historyless if applying a sequence of operations yields the same value in the object as applying just the last nontrivial operation in the sequence. A nontrivial operation is one that writes a value to the object.

The lower bound proof technique is related to an elegant 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 [4]. Their method, called the *covering argument*, aims at *read/write* objects, and is generalized herein to historyless objects.

The rest of this paper is organized as follows: Section 2 provides the system model and definitions of the problem. Section 3 presents the 2-bounded-bypass algorithm and Section 4 the FIFO algorithm. Next, Section 5 gives an impossibility result. Conclusions are finally drawn in Section 6.

2 System Model and Definitions

2.1 Asynchronous Shared Memory Model

Our model of an asynchronous shared memory system is based on the model described by Lynch in [13].

An algorithm is modeled 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 system. Each variable $v \in \mathcal{V}$ has an associated set of values, among which some are designated as the initial values. Each process i consists of 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 a shared memory access, in which case it is assumed to access 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

[1] The ARM processor provides the SWP instruction, which performs the same functionality as *fetch&store*.

a system state s, $s(i)$, $i \in \mathcal{P}$, denotes the state of process i at s, and $s(v)$, $v \in \mathcal{V}$, denotes the value of shared variable v at s. A system state at which each process and each shared variable, respectively, have a start state and an initial value is called an *initial* system state.

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. If step e of process i does not access any shared variable, then only the state of i may be modified. That is, in any $(s, e, s') \in \delta$ with the step e, $s(j) = s'(j)$ for every process $j \neq i$ and $s(v) = s'(v)$ for every shared variable v. On the other hand, if e accesses a shared variable v, then only the state of i and the value of v may be modified.

A step e is enabled at system state s if a system state s' exists such that $(s, e, s') \in \delta$. We assume that whether a process step is enabled at a system state depends only on the process state. That is, 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 t at which $t(i) = s(i)$ holds.

An *execution fragment* is either a finite sequence, $s_0, e_1, s_1, \ldots, s_{l-1}, e_l, s_l$, or an infinite sequence, $s_0, e_1, s_1, \ldots$, of alternating system states and steps such that $(s_k, e_{k+1}, s_{k+1}) \in \delta$ for every $k \geq 0$. An *execution* is an execution fragment beginning with an initial system state. A system state s is said to be *reachable* if a finite execution ending with s exists.

2.2 The Operations

In the model, shared variables can be accessed by processes through atomic operations. The model supports *fetch&store* in addition to atomic *read* and *write*. This operation is formally defined as follows:

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

The operation atomically writes value u to variable v and returns the old value.

2.3 The Problem

An asynchronous shared memory model has been described so far. The mutual exclusion problem is formally defined below.

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 when no other process is in its own. A process executes the *trying region* code to gain the permission to enter its critical region, and after it passes through the critical region, it executes the *exit region* code and then returns to the *remainder region*.

For each process i, Σ_i is partitioned into nonempty disjoint subsets R_i, T_i, C_i, and E_i. Process i is said to be in the remainder region (R), trying region (T), critical region (C), or exit region (E) at system state s if $s(i)$ belongs to R_i, T_i, C_i, or E_i, respectively. A system state is said to be *idle* if all processes are in R. Each initial system state is assumed to be idle. Each process is also assumed to cycle through its remainder, trying, critical, and exit regions, in that order.

Finally, an algorithm that solves the mutual exclusion problem must meet the conditions below.

Mutual exclusion. There is no reachable system state at which more than one process is in C.

The next condition depends on an assumption about process scheduling in executions. No process "halts" anywhere except possibly in R. Executions with this property are defined to be *admissible*. An execution α is formally defined as admissible if for each process $i \in \mathcal{P}$ that takes only finitely many steps in α, i's final state belongs to R_i.

Progress. At any point in an admissible execution

1. if at least one process is in T and no other process is in C, then at some later point, some process enters C;
2. if at least one process is in E, then at some later point, some process enters R.

This condition is necessary for the system to make progress, but does not guarantee that the critical region is granted fairly to each process. A situation, in which some process trying to enter C is forever prevented from gaining the permission (known as lockout, or starvation), may occur. Thus, some level of fairness of granting the critical region is often desirable.

Before defining the fairness properties below, which guarantee a bound on the number of bypasses, we assume that the trying region of each process consists of two parts: a *doorway* followed by a *waiting* part. The doorway part is *wait-free*: its execution requires only a bounded number of steps. The following properties prevent any process that has finished its doorway from being bypassed an arbitrary number of times by any other process.

A mutual exclusion algorithm is said to be **bounded-bypass** if it guarantees a b-bounded-bypass for some constant b. The b-bounded-bypass condition is defined as follows.

b-Bounded-bypass. Once a process i has passed through its doorway, no process can enter its critical region more than b times before i does so.

A mutual exclusion algorithm is said to be **FIFO** if when process i completes its doorway before j performs a step in its doorway, then j cannot enter C before i does so. It is intuitively clear that an FIFO algorithm is also an algorithm satisfying the bounded-bypass condition.

3 The 2-Bounded-Bypass Algorithm

This section presents a bounded-bypass mutual exclusion algorithm using two shared variables, as shown in Fig. 2. Fig. 1 illustrates an example to help explain the working of the algorithm.

In summary, the algorithm links competing processes as circular lists by the *fetch&store* operation along with a shared variable. The permission to enter the critical region is transmitted along a list after its construction. While the permission is transmitted, subsequent requests constitute a new waiting list. The new list is closed when each process in the old list has left its critical region, at which time the new list receives the permission. Likewise, after the new list is closed, subsequent requesting processes form another list and wait for the permission. Roughly speaking, permission is conveyed along a list, and then passed to the next waiting list. The algorithm thus satisfies the bounded-bypass condition.

According to the construction of a list, a competing process has the identity of its predecessor rather than its

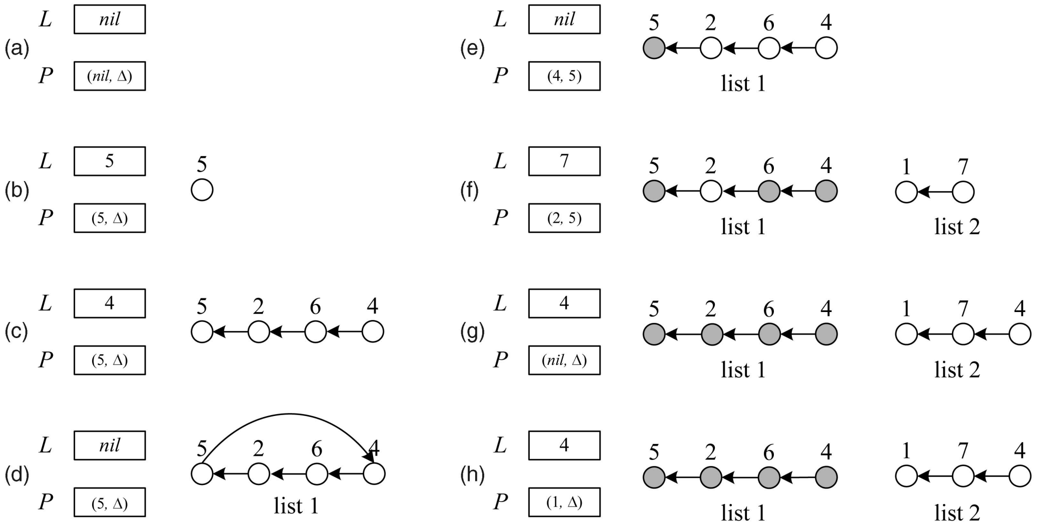

Fig. 1. An execution of the 2-bounded-bypass algorithm. A gray node indicates a process that has finished one life cycle. The symbol $\triangle$ denotes an arbitrary value. An arrow from process a to b represents that a has the identity of b.

successor. Consequently, the permission is conveyed from the tail of a list to the head, resulting in the failure to meet the FIFO condition. The next section describes a modified algorithm that eliminates this drawback and achieves the FIFO condition by initiating an additional phase for every list to redirect the links in the list.

3.1 An Informal Description of the Algorithm

The algorithm requires exactly two shared variables. Variable L organizes requests by processes to enter C, and variable P indicates which process has the permission to enter C. Additionally, each process has several private variables, which are not accessible to other processes.

As in Fu and Tzeng's algorithm [11], the proposed algorithm uses a *fetch&store* operation on a lock to link competing processes as a circular waiting list. Each process in its doorway executes *fetch&store* on shared variable L (i.e., the lock), announcing its identity and obtaining the identity of its predecessor if it has one. Thus, a waiting list is formed implicitly. Variable L is initially set to *nil*. A process that reads *nil* from L starts a waiting list and is the head of the list. Such a process in our design is responsible for closing the list and starting to transmit the permission along the list. Thus, the process is called the *controller* of the list.

After announcing its request by executing *fetch&store* on L, each process enters the waiting part of its trying region and starts to test P repeatedly until it has the permission to enter C. A controller repeatedly tests P until $P = nil$, which is the permission for a controller. A noncontroller repeatedly tests P until P equals its identity, which indicates that it gains the permission. Since $P = nil$ initially, the first controller always gains the permission.

A waiting list is closed when the controller of the list leaves C. The controller closes the list by executing *fetch&store(L, nil)*, which atomically returns the identity of the tail of the list and modifies L's value to *nil*. This closed waiting list contains all the processes making requests between the controller obtaining *nil* from L and modifying L's value to *nil*. Since L's value is changed to *nil*, subsequent requests constitute another waiting list.

After a list is closed, the permission is transmitted along the list from the tail to the head, allowing each process in the list to enter C in order. If the controller is not the only process in the list, then it passes the permission to the tail of the list by setting P to the identity of the tail. Each noncontroller in the list then hands the permission to its predecessor, by setting P to the identity of its predecessor, when it leaves C. However, if the predecessor is the head of the list, then the process passes the permission to the next waiting list rather than to the predecessor because the predecessor has left C. Some information is needed to check this situation, and is encoded into P. Let P hold a pair *(Receiver,Head)*, each being the identity of a process or *nil*.[2] The $Receiver$ component serves the original purpose of P, indicating which process can enter C, and the $Head$ component stores the identity of the head of the list so that each process can determine whether its predecessor is the head. If the predecessor's identity of a process is equal to $Head$, the process modifies $Receiver$'s value to *nil* instead of the predecessor's identity to convey the permission to the head of the next waiting list.

Fig. 1 illustrates an execution of the algorithm. Variables L and P are, respectively, set to *nil* and $(nil, \triangle)$, as shown in Fig. 1a. The symbol $\triangle$ in the *Head* component represents that the component is not used at this time and can be an arbitrary value.

In Fig. 1b, process 5 first makes a request by executing *fetch&store(L,5)* and obtains *nil* from L. Since *Receiver* $=$ *nil*, process 5 enters C after setting *Receiver* to 5. While process 5 is in C, processes 2, 6, and 4 make requests in turn. Processes 5, 2, 6, and 4 form a list as shown in Fig. 1c. Because none of processes 2, 6, and 4 succeeds in acquiring *nil* from L, they repeatedly test P until the $Receiver$ component of P equals their respective identities.

[2] Thus, P can assume $(n+1)^2$ distinct values, where n is the number of process, while L consists of $n+1$ values.

```
Shared variables:
$L \in \{nil, 1, \ldots, n\}$, initially $nil$
$P \in \{(Receiver, Head) \mid Receiver, Head \in \{nil, 1, \ldots, n\}\}$, initially $(nil, \triangle)$

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

Private variables of $i$:
$pred, tail, receiver, head \in \{nil, 1, \ldots, n\}$, initially arbitrary


     while $true$ do
R:       Remainder region
T1:      $pred := fetch\&store(L, i)$;
T2:      $(receiver, head) := P$;
T3:      if $pred = nil$ then                          ▷ as a controller
T4:          while $receiver \neq nil$ do              ▷ await $Receiver = nil$
T5:              $(receiver, head) := P$ od
T6:          $P := (i, \triangle)$;
         else                                          ▷ as a non-controller
T7:          while $receiver \neq i$ do                ▷ await $Receiver = i$
T8:              $(receiver, head) := P$ od
         fi
C:       Critical region
E1:      if $pred = nil$ then                          ▷ as a controller
E2:          $tail := fetch\&store(L, nil)$;           ▷ close the waiting list
E3:          if $tail \neq i$ then
E4:              $P := (tail, i)$;          ▷ wake up the tail and set $Head$ to $i$
             else
E5:              $P := (nil, \triangle)$ fi
         else                                          ▷ as a non-controller
E6:          if $pred = head$ then
E7:              $P := (nil, \triangle)$;
             else
E8:              $P := (pred, head)$ fi                ▷ wake up the predecessor
         fi
     od
```

Fig. 2. The 2-bounded-bypass algorithm.

When leaving C, process 5 closes the list, called list 1, by executing *fetch&store(L, nil)*. This operation obtains the identity of the tail and modifies L's value to *nil*. The edge from 5 to 4 in Fig. 1d indicates that the returned value is 4, i.e., the tail of the list is process 4. Process 5 then passes the permission to the tail and sets *Head* to 5 by writing (4, 5) to P, as shown in Fig. 1e. In Fig. 1f, process 4 gains the permission, enters C, and then passes the permission to process 6. Similarly, process 6 gains the permission from process 4, enters C, and then hands the permission to process 2 by writing $(2, 5)$ into P. In Fig. 1g, since the predecessor of process 2 is the head of the list, process 2 modifies *Receiver*'s value to *nil* to transmit the permission to the next waiting list, called list 2. Finally, in Fig. 1h, because *Receiver*'s value has been changed to *nil*, process 1, which is the head of list 2, enters C after setting *Receiver* to 1.

3.2 The Bounded-Bypass Algorithm

Variables L and P are initially set to *nil* and $(nil, \triangle)$, respectively. Each process stores the values of the two components of shared variable P into its private variables *receiver* and *head*.

In the trying region, the doorway is composed of line T1 in Fig. 2, and the waiting part is composed of the rest (T2-T8). A process i in the doorway executes *fetch&store(L, i)* (T1) to announce its request, and then enters the waiting part. If *pred* $=$ *nil*, i.e., the returned value of T1 is *nil*, then the process identifies itself as a controller and begins repeatedly checking P until the *Receiver* component of P equals *nil* (T4-T5). When *Receiver* $=$ *nil*, process i sets *Receiver* to i (T6) and then enters the critical region. In contrast, if $pred \neq nil$, then process i repeatedly tests P until *Receiver* $=$ i holds (T7-T8).

In the exit region, a controller closes the current waiting list by performing *fetch&store(L, nil)* (E2). If the returned value, which is stored in *tail*, is not equal to its identity (i.e., the list contains some other process), then the controller passes the permission to the tail of the list and sets *Head* to its identity (E4); otherwise, the controller just modifies *Receiver*'s value to *nil* (E5). For each noncontroller, if its predecessor is not the head of the list, then it simply transfers the permission to its predecessor by setting *Receiver* to *pred* (E8); otherwise, it modifies *Receiver*'s value to *nil* to convey the permission to the next waiting list (E7).

3.3 Correctness Proofs

Since each labeled instruction in the trying and exit regions accesses at most one shared variable, it is set to correspond to a step of a process. That is, each labeled instruction in the algorithm is atomic. For each process i, pc_i is defined as the program counter of i; for instance, $pc_i = \mathrm{T1}$ at a system state means that step T1 of process i is enabled. A private variable v of process i is denoted as v_i. Finally, a process i in T, C, or E is defined as a **controller** provided that $pred_i = nil$.

3.3.1 Mutual Exclusion

In the algorithm, whether a process in T can enter C depends on the value of *Receiver*. If *Receiver* $=$ *nil*, then a controller waiting for *nil* in T is permitted to enter C, while if *Receiver* $= i$, $1 \leq i \leq n$, then only process i is permitted to do so. Inspection of the algorithm clearly indicates that only the process in E can modify *Receiver*'s value to *nil* or the identity of some other process using one of steps E4, E5, E7, or E8. (Although a controller in T modifies *Receiver*'s value by executing T6, it sets *Receiver* to its identity, allowing no other process to enter C.) We show that a *nil* can be taken as the permission for at most one process. Hence, a process in E allows at most one process to enter C. Additionally, since *Receiver* is set to *nil* initially, and a *nil* permits at most one process to enter C, at most one process can enter C from the starting state. Thus, the mutual exclusion condition is ensured.

The following lemma states that at most one controller is at T4, T5, or T6. In other words, at most one controller is waiting for *nil* at any reachable system state. Once *Receiver* $=$ *nil*, the controller enters C after step T6, which sets *Receiver* to its identity, a non*nil* value. Thus, a *nil* in *Receiver* permits at most one process to enter C.

Lemma 1. *At any reachable system state,*

$$|\{i \in \mathcal{P} | pred_i = nil \wedge pc_i \in \{\mathrm{T4}, \mathrm{T5}, \mathrm{T6}\}\}| \leq 1.$$

Proof. Since each process is in R at an initial system state, no process is in the set, and thus, the statement is true. We then argue that if a process enters the set at a systems state, no other process can enter the set until it leaves the set. Consequently, starting from an initial state, at most one process is in the set at all reachable system states.

The steps that could cause processes to enter the set are considered. A process i can enter the set exactly if $pred_i = nil$ after step T1, which simultaneously sets $L := i$. Before process i modifies L's value to *nil* by executing step E2, no other process can obtain *nil* from L when executing step T1, and therefore, no other process will enter the set. That is, no process can enter the set until i leaves the set. □

Since a process in E allows at most one process to enter C and at most one process can enter C from the starting state, the following theorem holds:

Theorem 2. *The algorithm guarantees mutual exclusion.*

3.3.2 Progress

We argue that the algorithm satisfies the lockout-freedom condition, that if no process stays in C indefinitely, any process in T eventually enters C, and any process in E eventually enters R. A lockout-free algorithm is intuitively also an algorithm satisfying the progress condition.

Before proving lockout-freedom, we present a definition that intends to organize all requests in an execution. A *list* is a sequence of processes that execute step T1 in an execution fragment that starts with a step T1 that succeeds in acquiring *nil* from L, and ends with the following step E2, which modifies L's value to *nil*. The following lemma shows that starting from the last process in a list, we can trace the whole list from the tail to the head through the value of *pred* of each process in the list:

Lemma 3. *In any execution fragment that starts with a system state at which L has the nil value and ends with a system state at which L's value is changed to nil, each process i that performs step T1 has the identity of its predecessor in* $pred_i$ *if there is one.*

Proof. Due to the *fetch&store* primitive, a process i that performs step T1 stores L's value in $pred_i$ and, in the same step, modifies L's value to its identity. Thus, each process that performs step T1 obtains the identity of its predecessor if there is one. □

Theorem 4. *The algorithm guarantees lockout-freedom.*

Proof. The argument for the exit region is simply that since no loop occurs in the exit region, each process in E eventually enters R.

The lockout-freedom condition for the trying region is now considered. We first show that each request is properly recorded in a list, and then argue that each list will receive the permission to enter C.

In the algorithm, each process i makes a request by performing *fetch&store(L, i)* (T1). A process that succeeds in acquiring *nil* from L starts a waiting list and becomes the controller of the list. Suppose a list controller gains permission to enter C at a later point. After passing through C, the controller closes the list by executing E2 which obtains the identity of the tail and modifies L's value to *nil*, and then starts to convey the permission along the list from the tail. By Lemma 3, all processes that perform step T1 before the controller closes the list are organized into this list, in which each process except the controller has the identity of its predecessor. Since L's value is changed to *nil*, subsequent requests form a new list in the same way. Thus, each request is properly recorded in a list. Clearly, a closed list contains a *finite* number of waiting processes, since each process can occur in a list at most once.

To prove that each requesting process eventually enters C, it remains to be shown that each controller receives the permission. Since *Receiver* is initially set to *nil*, the first controller always gains the permission. The controller closes the list, and conveys the permission to the tail of the list, when it leaves C. Since a closed list contains a finite number of processes, if no process stays in C indefinitely, then each process in the list eventually gains the permission to enter C. When the process next to the controller receives the permission, since its *pred* equals *Head*, it redirects the permission to the next controller by setting *Receiver* to *nil* after passing through C. (From Lemma 1, if one controller is waiting for *nil*, exactly one such controller exists.) Thus, each controller eventually receives the permission. □

3.3.3 Bounded-Bypass

A process i is said to be in the doorway if $pc_i = \mathrm{T1}$, and it is said to be in the waiting part if $pc_i \in \{\mathrm{T2}, \ldots, \mathrm{T8}\}$. As shown in the proof of Theorem 4, a process is recorded in a waiting list after passing through its doorway (i.e., after executing T1). Since a list does not receive the permission until each process in the previous list has left C, a process in a list may be bypassed by those processes in the same list and in the previous list. In addition, because a process can occur in a list at most once, a waiting process may be bypassed by any individual process at most twice. In other words, the algorithm satisfies 2-bounded-bypass. The worst case, in which a process that has finished its doorway is bypassed twice by another process, may occur when a noncontroller in a list quickly makes a request appending to the new list after receiving the permission. For example, in Fig. 1f, 1g, and 1h, process 7 is bypassed twice by process 4, which makes a request after receiving the permission. Consequently, the following theorem holds:

Theorem 5. *The algorithm guarantees 2-bounded-bypass.*

4 The FIFO Algorithm

The above algorithm is 2-bounded-bypass. This section gives an FIFO algorithm, based on the 2-bounded-bypass algorithm, with the same number of shared variables and the same set of operations.

The FIFO algorithm follows the same concept of the 2-bounded-bypass algorithm, except that it initiates an additional phase to redirect the links in a list to meet the FIFO condition. Owing to the implementation of the communication phase, the number of values taken on by shared variable P increases from $(n+1)^2$ to $2(n+1)^3$.

4.1 An Informal Description of the Algorithm

The FIFO algorithm also organizes waiting processes into circular lists. Each process in its doorway announces its request by executing *fetch&store* on the shared variable L. In this step, a contending process obtains the identity of its predecessor if it has one, and replaces L with its identity. As in the 2-bounded-bypass algorithm, a process gaining *nil* from L is the head of the list that it closes, and takes the role of a controller. That is, the process closes the list, and starts to transmit the permission along the list, when it leaves the critical region.

However, the FIFO algorithm conveys the permission along a list in the reverse order. Recall that the 2-bounded-bypass algorithm is ordered from the tail to the head along a waiting list, causing it to fail the FIFO condition. Each process in a waiting list, except the head, has the identity of its predecessor rather than its successor. To achieve the FIFO requirement, an additional communication phase is required to inform each process of its successor's identity, so that the permission can be passed in the FIFO order.

The algorithm initiates such a phase by the controller of a list when the controller leaves C. Starting from the tail, each noncontroller except the immediate successor of the head writes a message, in turn, to inform its predecessor of its identity. The communication phase is completed when the immediate successor of the head receives its successor's identity. The permission is then conveyed from the successor of the head to the tail. The algorithm thus satisfies the FIFO condition.

Implementing this phase requires some communication mechanism. In the algorithm, the shared variable P is used for two purposes: to indicate which process is permitted to enter C and to inform processes of their respective successors. The use of a shared variable for these two purposes is inspired by the algorithms [3] proposed by Burns et al. To serve both purposes, the variable holds a 4-tuple (*Type, Receiver, Successor, Head*), where *Type* is a value in $\{\mathbf{Info}, \mathbf{Grant}\}$, and the other parts take on values from $\{nil, 1, \ldots, n\}$. The number of values taken on by P in this algorithm is $2(n+1)^3$, compared with $(n+1)^2$ in the 2-bounded-bypass algorithm.

The *Type* component represents the purpose of a variable. If *Type* has the value **Grant**, then variable P is adopted to convey the permission. In this case, the *Receiver* component represents the process that has the permission, while the *Successor* and *Head* are not used, and may have arbitrary values, denoted as $\triangle$ in the algorithm. Otherwise, if the *Type* component has the value **Info**, then variable P is used to inform some process of its successor. In this case, *Receiver* represents the receiver of the message; *Successor* represents the identity of the receiver's successor, and *Head* represents the identity of the head of the list.

Fig. 3 illustrates an execution process of the algorithm. The sequence of requests in list 1 is the same as that given to the 2-bounded-bypass algorithm in Fig. 1, but the order in which the permission is conveyed among noncontrollers is opposite.

Variables L and P are initially set to *nil* and $(\mathbf{Grant}, nil, \triangle, \triangle)$, respectively. In Fig. 3a, processes 5, 2, 6, and 4 make requests, in turn, and constitute a list. Because process 5 obtains *nil* from L, and receives a message that *Type* = **Grant** and *Receiver* = *nil*, it has the permission for a controller. It enters C after setting P to $(\mathbf{Grant}, 5, \triangle, \triangle)$. In contrast, processes 2, 6, and 4 repeatedly test P until they receive their respective messages. Process 5 performs *fetch&store(L, nil)* to obtain the identity of the tail, 4 in this case, and modifies L's value to *nil*, when it leaves C. It then starts a communication phase by writing $(\mathbf{Info}, 4, nil, 5)$, as shown in Fig. 3b. This message notifies process 4 that it is the tail of the list (because *Successor* = *nil*), and that the head is process 5. Process 4 then receives the message and writes a new message $(\mathbf{Info}, 6, 4, 5)$ to process 6, as shown in Fig. 3c. The new message informs process 6 that its successor is process 4, and that the head of the list is process 5. Similarly, process 6 receives the message from process 4, and writes a new message $(\mathbf{Info}, 2, 6, 5)$ to inform process 2, as shown in Fig. 3d.

Process 2 receives the message written by process 6, and becomes aware that it is the immediate successor of the head, because its predecessor is the head of the list. This means that the communication phase is completed. Process 2 enters C, and conveys the permission to its successor, process 6, by writing $(\mathbf{Grant}, 6, \triangle, \triangle)$ into P, as shown in Fig. 3e. Process 6 then gains the permission, and conveys it to process 4, as shown in Fig. 3f. Since process 4 is the tail of the list, process 4 hands the permission to the next waiting list, by setting P to

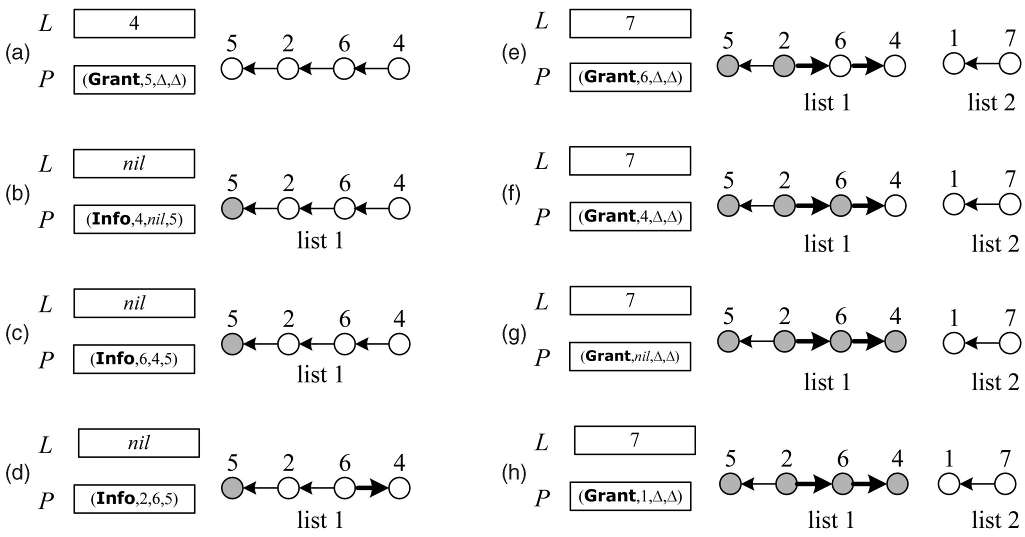

Fig. 3. An execution of the FIFO algorithm. The notation is the same as that in Fig. 1.

$(\mathbf{Grant}, nil, \triangle, \triangle)$, when it leaves C, as shown in Fig. 3g. Process 1, which is the head of the next list, then receives the permission to enter C, as shown in Fig. 3h, because $Type =$ **Grant** and *Successor* $=$ *nil*.

4.2 The FIFO Algorithm

This section describes the algorithm in more detail. Each process stores the values of the four components of shared variable P into its private variables $type$, $receiver$, $successor$, and $head$.

In the trying region, the doorway is composed of line T1 in Fig. 4, and the waiting part is composed of the rest (T2-T12). A process i in the doorway announces its request and obtains the predecessor's identity, if it has one, by performing *fetch&store(L, i)* (T1). Process i then enters the waiting part. If the returned value of T1 is *nil* (i.e., *pred* $=$ *nil*), then process i identifies itself as a controller, and repeatedly checks P until it gains the permission for a controller (i.e., $type =$ **Grant** and $receiver = nil$) (T4 and T5). When $type =$ **Grant** and $receiver = nil$, process i enters C after setting *Receiver* to i (T6). In contrast, if $pred \neq nil$, then process i repeatedly tests P until a message belonging to it is received (T8 and T9). If the received message has value **Grant** in the *Type* component, then i enters C immediately; otherwise, if the message has value **Info** in *Type*, then two cases may occur:

1. When *pred* $=$ *head*, process i is the immediate successor of the head. In other words, the communication phase is completed and i is permitted to enter C.
2. When $pred \neq head$, process i conveys its own identity and the identity of the head to its predecessor (T11), and continues to check P (T12).

In the exit region, a controller i performs *fetch&store(L, nil)* (E2) to close the current waiting list, and stores the returned value in *tail*. If *tail* is not equal to its identity, then the list is nonempty, and therefore, the controller starts a communication phase by writing $(\mathbf{Info}, tail, nil, i)$ into P (E4). This value indicates that the receiver is the tail of the list, that the tail has no successor because the *Successor* part is equal to *nil*, and that the head of the list is process i. Otherwise, if *tail* equals the controller's identity, then it just modifies P's value to the initial value $(\mathbf{Grant}, nil, \triangle, \triangle)$ (E5). For every noncontroller, if $successor = nil$ holds, then it realizes that it is the tail of the list, and gives the permission to the next list by writing $(\mathbf{Grant}, nil, \triangle, \triangle)$ into P (E7). Otherwise, it hands the permission to its successor by changing P's value to $(\mathbf{Grant}, successor, \triangle, \triangle)$ (E8).

4.3 Correctness Proofs

Since the correctness argument of the 2-bounded-bypass algorithm has shown the basic idea about arranging waiting processes into lists, this section simply provides a proof sketch. The notation and the definition of a controller are the same as those in Section 3.3.

The mutual exclusion condition is proven by the strategy adopted in the 2-bounded-bypass algorithm. We also show that a process in E enables at most one process to enter C, and at most one process can enter C from the starting state.

In the algorithm, a process i in T is permitted to enter C only if one of the following conditions holds.

Condition 1. $type =$ **Grant** and $receiver = nil$ hold. Informally, process i, which is a controller, obtains the permission to enter C.

Condition 2. $type =$ **Grant** and $receiver = i$ hold. Informally, process i, which is a noncontroller, obtains the permission to enter C.

Condition 3. $type =$ **Info**, $receiver = i$, and $pred = head$ hold. Informally, process i, which is aware that it is the immediate successor of the controller and that the communication phase is finished, obtains the permission to enter C.

Inspection of the algorithm indicates that a process in E performs exactly one of E4, E5, E7, and E8 to change P's value. As shown below, each step enables at most one process to enter C.

Step E4 modifies P's value to $(\mathbf{Info}, tail, nil, i)$. A value in P is said to be a *communication word* if $Type =$ **Info**. According to the algorithm (T10-T12), a communication word may enable the receiver of the word to satisfy

```
Shared variables:
$L \in \{nil, 1, \ldots, n\}$, initially $nil$
$P \in \{(Type, Receiver, Successor, Head) \mid Type \in \{\textsf{Info}, \textsf{Grant}\},$
      $Receiver, Successor, Head \in \{nil, 1, \ldots, n\}\}$, initially $(\textsf{Grant}, nil, \triangle, \triangle)$

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

Private variables of $i$:
$type \in \{\textsf{Info}, \textsf{Grant}\}$
$pred, tail, receiver, successor, head \in \{nil, 1, \ldots, n\}$, initially arbitrary


     while $true$ do
R:          Remainder region
T1:         $pred := fetch\&store(L, i)$;
T2:         if $pred = nil$ then                                      ▷ as a controller
T3:                $(type, receiver, successor, head) := P$;
T4:                while $type \ne \textsf{Grant}$ or $receiver \ne nil$ do
T5:                       $(type, receiver, successor, head) := P$ od
T6:                $P := (\textsf{Grant}, i, \triangle, \triangle)$;
            else                                                      ▷ as a non-controller
T7:                $(type, receiver, successor, head) := P$;
T8:                while $receiver \ne i$ do
T9:                       $(type, receiver, successor, head) := P$ od
T10:               if $type = \textsf{Info}$ and $pred \ne head$ then
T11:                      $P := (\textsf{Info}, pred, i, head)$;          ▷ inform its predecessor
T12:                      goto T7;
                   fi
            fi
C:          Critical region
E1:         if $pred = nil$ then                                      ▷ as a controller
E2:                $tail := fetch\&store(L, nil)$;                    ▷ close the waiting list
E3:                if $tail \ne i$ then
E4:                       $P := (\textsf{Info}, tail, nil, i)$;            ▷ inform the tail
                   else
E5:                       $P := (\textsf{Grant}, nil, \triangle, \triangle)$ fi
            else                                                      ▷ as a non-controller
E6:                if $successor = nil$ then
E7:                       $P := (\textsf{Grant}, nil, \triangle, \triangle)$;
                   else
E8:                       $P := (\textsf{Grant}, successor, \triangle, \triangle)$ fi   ▷ wake up the successor
            fi
     od
```

Fig. 4. The FIFO algorithm.

Condition 3. If not, the receiver writes a new communication word to its predecessor and backs to step T7. Thus, step E4 enables at most one process to satisfy Condition 3.

Step E8 modifies P's value to $(\textbf{Grant}, successor, \triangle, \triangle)$, making the process whose identity is equal to $successor$ to satisfy Condition 2.

Steps E5 and E7 set P to the initial value, $(\textbf{Grant}, nil, \triangle, \triangle)$. Using the argument in Lemma 1, we have the counterpart of Lemma 1 below:

Lemma 6. *At any reachable system state,*

$$|\{i \in \mathcal{P} | pred_i = nil \wedge pc_i \in \{\mathrm{T3, T4, T5, T6}\}\}| \le 1.$$

Namely, there is at most one controller that is blocked, waiting for Condition 1 to hold for it. Thus, the initial value allows at most one process to enter C. This implies that E5 and E7 each enable at most one process to gain the permission, and furthermore, implies that at most one process can enter C from the starting state, at which P has the initial value. Thus, the mutual exclusion condition is ensured.

We now argue that the proposed algorithm satisfies the lockout-freedom condition. The argument is similar to that of Theorem 4. Requesting processes are also organized into lists. A process i is said to be in the doorway if $pc_i = \mathrm{T1}$, and it is said to be in the waiting part if $pc_i \in \{\mathrm{T2}, \ldots, \mathrm{T12}\}$. The proof of Lemma 3 also holds for the FIFO algorithm. According to this lemma, each process that finishes its doorway has the identity of its predecessor if there is one, by which the head of a list initiates a communication phase to reverse the order in a list. Consequently, the permission can be conveyed according to the sequence of the requests; the algorithm thus satisfies not only the lockout-freedom condition, but also the FIFO condition.

5 An Impossibility Result

This section shows that the bounded-bypass mutual exclusion problem cannot be solved at all with less than two shared variables if only *fetch&store* and *read/write* are used. This result implies that both of our algorithms are optimal. In the proof of this impossibility result, a shared variable associated with *fetch&store* and *read/write* is modeled as a type of historyless object for two reasons. First, this approach simplifies the presentation of the proof. Second, a more general result is thus provided: using only historyless objects, two objects are required to solve the bounded-bypass mutual exclusion problem. Historyless objects, as proposed by Fich et al. [9], are defined below, and then the proof is presented.

A shared *object* has an associated set of possible values and supports a fixed set of operations that provide the only means to manipulate the object. An operation of an object is regarded as *trivial* if it leaves the value of the object unchanged. An operation e is said to overwrite 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; and operations *write* and *fetch&store* overwrite each other. Therefore, an object associated with any subset of *read, write,* and *fetch&store* is historyless, implying that the objects provided in our model are historyless. The value of a historyless object depends only on the last nontrivial operation applied to it, because the last nontrivial operation overwrites the value that might have been written to the object.

The proof can now be presented by following the proving strategy proposed by Burns and Lynch [4]. Two more definitions are needed. An indistinguishability relation, which is widely used in lower bound proofs, is first defined.

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

The second definition generalizes that of Burns and Lynch [4], which states that a process *covers* shared variable x provided that a *write* operation of the process is enabled to write to x. An enabled *write* operation can overwrite the variable involved. Similarly, an enabled nontrivial operation of a historyless object can also overwrite the object. Thus, the concept of "covering" is generalized to historyless objects.

Definition 2. *Process i covers a historyless object x at system state s provided that a nontrivial operation of i is enabled to manipulate x.*

That is, when process i covers a historyless object x, i can overwrite the value of x in its next step.

A basic lemma showing that any process that reaches R from C on its own must take a nontrivial operation to some object is presented before proving the lower bound. The proof of this lemma is similar to the result provided by Lynch in [13, pp. 301-302] which shows that a process reaching C from R on its own must write something into shared memory before doing so.

Lemma 7. *Suppose that A is a mutual exclusion algorithm, shared by $n \geq 2$ processes, using only historyless objects. Let s be a reachable system state of A at which process i is in C. If process i reaches R in a finite execution fragment starting from s that involves steps of i only, then it must take a nontrivial operation to some object along the way.*

Proof. Let α_1 be a finite execution fragment that starts from s (at which i is in C), involves steps of i only, and ends with process i in R. By contradiction, suppose that α_1 does not include any nontrivial operation to any object. An execution that violates the mutual exclusion condition is constructed herein.

Let s_1 be the system state at the end of α_1. Since process i does not write anything to any object, then $s \overset{j}{\sim} s_1$ for every $j \neq i$.

According to the progress condition, a finite execution fragment α_2, starting from s_1 and not including any step of process i, exists such that some process reaches C. Because $s \overset{j}{\sim} s_1$ for every $j \neq i$, α_2 is also executable from s.

An execution α violating the mutual exclusion condition can be easily constructed as follows: Execution α begins with a finite execution fragment leading to reachable system state s, and then continues with α_2. However, two processes are in C at the end of α, contradicting the mutual exclusion condition. □

The main idea of the lower bound proof is that when a process covers a historyless object x, it can overwrite the information that other processes might have written to x in its next step. If a request of some process is overwritten, another process may enter C an arbitrary number of times, violating the bounded-bypass condition.

Theorem 8. *If algorithm A solves the bounded-bypass mutual exclusion problem for n processes where $n > 2$, using only historyless objects, then A must use at least two objects.*

Proof. Suppose for the sake of contradiction that there is such an algorithm A using only one historyless object, say x, and guaranteeing b-bounded-bypass for some constant b. Let s be an initial system state. An execution of A that violates the bounded-bypass condition is constructed below and is depicted in Fig. 5.

The progress condition implies that there is an execution involving process i only, starting from s, that causes process i to enter C once and back to an idle system state s'. Lemma 7 implies that process i must take a nontrivial operation to some object in E in this solo execution. Since only one object is used, process i must take a nontrivial operation to the historyless object x in E. Thus, i must cover x at some point in E.

Let α_1 be the prefix of this solo execution up to the **last** point where process i covers x in E. At this point, the last nontrivial operation of i in the solo execution is enabled. (That is, i can write a value to x in its next step such that x has the same value as that at system state s'.) Then, α_1 is extended to α_2 by allowing process j, which is in R at the end of α_1, to enter T and finish its wait-free doorway, and then allowing process i to overwrite x. Let the final system states of α_1 and α_2 be s_1 and s_2, respectively. Object x has the same value at s' and s_2 because the last

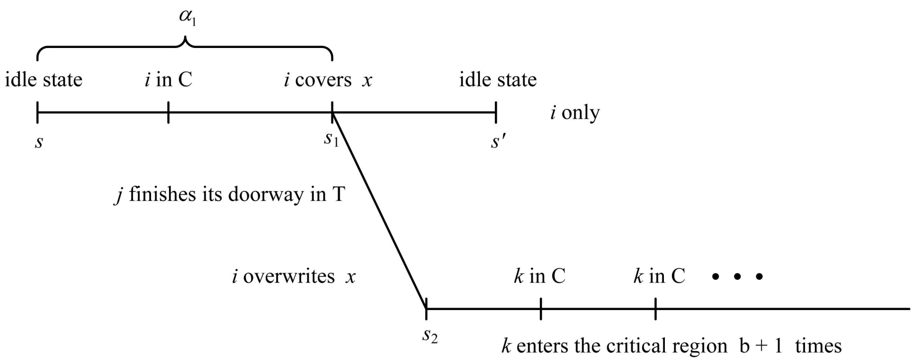

Fig. 5. The execution for the proof of Theorem 8.

nontrivial operation in the execution leading to s' is the same as that in the execution leading to s_2. Therefore, $s' \overset{k}{\sim} s_2$ for every $k \neq i$ and j.

Consider any process k that is different from i and j. (k exists since $n > 2$.) Since $s' \overset{k}{\sim} s_2$, and s' is an idle system state, the progress condition implies that process k can enter the critical region an arbitrary number of times on its own, starting from s_2. Additionally, process j must remain in the trying region at s_2, to avoid violating the mutual exclusion condition.

A counterexample execution α is constructed as follows: It begins with α_2 and then continues by allowing k to enter the critical region $b+1$ times, as if process j had never entered its trying region. Execution α violates the bounded-bypass condition, because process j, which has passed through its doorway, is bypassed more than b times by k. □

6 Conclusions

We have provided two fair and space-efficient algorithms for systems under time and memory constraints. The first algorithm is 2-bounded-bypass; the second is an FIFO algorithm based on the first algorithm. Each algorithm adopts the commonly available operations *fetch&store* and *read/write*.

Each algorithm utilizes only two shared variables, one for arranging requests and the other for communicating messages. The shared variable for arranging requests requires $n+1$ distinct values in either algorithm, where n is the number of processes. In contrast, to improve the fairness from the bounded-bypass condition to FIFO, the FIFO algorithm increases the number of values taken on by the other shared variable from $(n+1)^2$ to $2(n+1)^3$. That is, the size of the shared variable for communicating messages is increased from $2\log_2(n+1)$ bits to $1+3\log_2(n+1)$ bits. The best choice of algorithm thus depends on the size of the shared variables in the underlined system.

Furthermore, we have shown that any bounded-bypass algorithm using the same set of operations must utilize at least two shared variables, regardless of the size of the variables. This lower bound is proven by showing a more general result that two objects are necessary to solve the bounded-bypass mutual exclusion problem when using only historyless objects. Since shared variables associated with *fetch&store* and *read/write* belong to the class of historyless objects, the more general result applies to our model, implying that both the algorithms are optimal with respect to the number of shared variables. The proof technique is derived from that of Burns and Lynch [4].

One disadvantage of the algorithms is that the hot spot contention [7] can be up to n. The hot spot contention is the maximum number of pending operations for any individual shared variable in any execution, and this number is one of the principal determiners of the system performance. Because each algorithm utilizes only a constant number of shared variables to meet the memory constraint, $\Omega(n)$ hot spot contention is inevitable.

Additionally, each competing process in the algorithms repeatedly tests a shared variable while it is waiting to enter its critical region. Such repeated testing may generate much traffic on the interconnection network between the process and the memory, heavily degrading the system performance. A complexity metric that counts the number of *remote memory references* (RMRs) is widely used to evaluate mutual exclusion algorithms in systems with hardware support for cache coherence or distributed shared memory [2]. In cache-coherent systems, both the algorithms have $O(n)$ RMR complexity. Since the algorithms are bounded-bypass and each process performs a constant number of steps to modify shared variables in its trying and exit regions, the cached copies of these shared variables are updated $O(n)$ times during a process' life cycle. Thus, a process takes $O(n)$ RMRs to pass through its critical region once. In distributed shared memory systems, a process in the algorithms, however, may take an unbounded number of RMRs in a busy-waiting loop. The problem can be alleviated by using a collision avoidance technique such as *exponential backoff*. A contending process increases its delay time before testing again after a failed attempt to obtain the required value from a remote shared variable.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work was supported in part by the National Science Council of Taiwan under Grant NSC-92-2213-E-009-064. A preliminary version of this work was presented at the 11th IEEE International Conference on Parallel and Distributed Systems, July 2005.

References

[1] *ARM1136JF-S and ARM1136J-S Technical Reference Manual,* ARM, Inc., http://www.arm.com/documentation/ARMProcessor_Cores/, 2005.

[2] J.H. Anderson, Y.-J. Kim, and T. Herman, "Shared-Memory Mutual Exclusion: Major Research Trends Since 1986," *Distributed Computing,* vol. 16, nos. 2/3, pp. 75-110, Sept. 2003.

[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," *J. ACM,* vol. 29, no. 1, pp. 183-205, Jan. 1982.

[4] J.E. Burns and N.A. Lynch, "Bounds on Shared Memory for Mutual Exclusion," *Information and Computation,* vol. 107, no. 2, pp. 171-184, Dec. 1993.

[5] S.-H. Chen and T.-L Huang, "A Fair and Space-Efficient Mutual Exclusion," *Proc. 11th Int'l Conf. Parallel and Distributed Systems,* pp. 467-473, July 2005.

[6] E.W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Comm. ACM,* vol. 8, no. 9, p. 569, Sept. 1965.

[7] C. Dwork, M. Herlihy, and O. Waarts, "Contention in Shared Memory Algorithms," *J. ACM,* vol. 44, no. 6, pp. 779-805, Nov. 1997.

[8] F. Fich, D. Hendler, and N. Shavit, "On the Inherent Weakness of Conditional Primitives," *Distributed Computing,* vol. 18, no. 4, pp. 267-277, Mar. 2006.

[9] F. Fich, M. Herlihy, and N. Shavit, "On the Space Complexity of Randomized Synchronization," *J. ACM,* vol. 45, no. 5, pp. 843-862, Sept. 1998.

[10] M.J. Fischer, N.A. Lynch, J.E. Burns, and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Space," *ACM Trans. Programming Languages and Systems,* vol. 11, no. 1, pp. 90-114, Jan. 1989.

[11] S.S. Fu and N.-F. Tzeng, "A Circular List-Based Mutual Exclusion Scheme for Large Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 8, no. 6, pp. 628-639, June 1997.

[12] J.G. Ganssle, *The Art of Programming Embedded Systems.* Academic Press, 1991.

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

[14] J.M. Mellor-Crummey and M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors," *ACM Trans. Computer Systems,* vol. 9, no. 1, pp. 21-65, Feb. 1991.

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

[16] I. Rhee, "Optimizing a FIFO, Scalable Spin Lock Using Consistent Memory," *Proc. 17th IEEE Real-Time Systems Symp.,* pp. 106-114, Dec. 1996.

[17] K.M. Zuberi and K.G. Shin, "An Efficient Semaphore Implementation Scheme for Small-Memory Embedded Systems," *Proc. Third IEEE Real-Time Technology and Applications Symp.,* pp. 25-34, June 1997.

[18] K.M. Zuberi and K.G. Shin, "EMERALDS: A Small-Memory Real-Time Microkernel," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 909-927, Oct. 2001.

Sheng-Hsiung Chen received the BS degree in computer science and information engineering and the PhD degree in computer science from the National Chiao Tung University, Taiwan, in 1998 and 2008, respectively. Since 2006, he has been with the Physical Design Group, SpringSoft, Inc., Taiwan. His main research interests are in distributed algorithms, particularly, in fault tolerance and synchronization.

Ting-Lu Huang received the BS degree in physics from Tung-hai University, Taiwan, in 1976, the MS degree in computer science from the University of Texas at Arlington, in 1981, and the PhD degreee in computer science from Northwestern University in 1989. He is currently an associate professor in the Department of Computer Science at National Chiao Tung University. He is engaged in research on distributed algorithms and distributed systems.