Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 66 (2006) 1455 – 1471
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

A tight bound on remote reference time complexity of mutual exclusion in the read-modify-write model[☆]

Sheng-Hsiung Chen*, Ting-Lu Huang

Department of Computer Science and Information Engineering, National Chiao Tung University, EC635R 1001 Ta Hsueh Rd., Hsinchu 300, Taiwan

Received 9 September 2005; received in revised form 12 April 2006; accepted 31 July 2006
Available online 15 September 2006

Abstract

In distributed shared memory multiprocessors, remote memory references generate processor-to-memory traffic, which may result in a bottleneck. It is therefore important to design algorithms that minimize the number of remote memory references. We establish a lower bound of three on remote reference time complexity for mutual exclusion algorithms in a model where processes communicate by means of a general read-modify-write primitive that accesses at most one shared variable in one instruction. Since the general read-modify-write primitive is a generalization of a variety of atomic primitives that have been implemented in multiprocessor systems, our lower bound holds for all mutual exclusion algorithms that use such primitives. Furthermore, this lower bound is shown to be tight by presenting an algorithm with the matching upper bound.

Keywords: Mutual exclusion; Atomic instructions; Shared memory systems; Time complexity; Tight bounds

1. Introduction

The mutual exclusion problem is fundamental in multiprocessing systems for managing accesses to a single indivisible resource. The problem is to design an algorithm guaranteeing that at most one process at a time is permitted to access the resource within a distinct part of code called its *critical region*.

In shared memory systems, since all processes communicate through the shared memory, each competing process may test certain shared variable(s) repeatedly while it is waiting to enter its critical region. Such repeated testing may produce a large amount of processor-to-memory traffic in shared memory systems, heavily degrading the system performance. This problem can be avoided in two architectural paradigms of shared memory systems: distributed shared memory (DSM) systems, in which each process has a local portion of shared memory, and cache coherent (CC) systems, in which each process has a local cache [23]. In DSM systems, a memory reference to a shared variable will not cause interconnect traffic if the variable is stored in the local portion of shared memory. In CC systems, whether a memory reference causes interconnect traffic depends on the caching protocol. Generally speaking, the first reference (be it read, write, or both) to a shared variable will cause interconnect traffic and establish a cached copy. Subsequent references, however, will not cause traffic until the cached copy of the shared variable is updated or invalidated. In general, a memory reference is regarded as *local* if it does not cause any interconnect traffic; otherwise, it is *remote*. Recent work on the mutual exclusion problem has focused on the design of *local-spin* algorithms, which reduce the number of remote memory reference (RMR) steps by busy waiting only on locally accessible shared variables. A number of performance studies [6,4,17,20,23,24] have shown that synchronization algorithms minimizing the number of RMR steps have the best performance.

To evaluate mutual exclusion algorithms, the conventional time complexity, which counts all steps for one process in the worst case, might be inappropriate. This is because in any

[☆] The tight bound contains two parts: the lower bound and the upper bound. The former appears in the *Proceedings of the International Computer Symposium*, Taiwan, 2004, pp. 1352–1357, [11]. The latter appears in the *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999, pp. 224–231, [18].

* Corresponding author. Fax: +886 3 5724176.
E-mail address: chenss@csie.nctu.edu.tw (S.-H. Chen).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.07.002

algorithm in which a process enters a busy-waiting loop when its critical region is unavailable, the worst case number of steps taken by one waiting process is unbounded. In other words, the conventional time complexity yields no useful information concerning the performance of such algorithms. Since the number of RMR steps significantly reflects the performance of an algorithm, Anderson and Yang [5] were the first to propose the number of RMR steps as a time complexity metric. To be more specific, the time complexity of a mutual exclusion algorithm is the worst case number of RMR steps taken by any single process to enter and exit its critical region once. One may consider the amortized number of RMR steps instead of the worst case number as the time complexity of an algorithm. But, as Anderson and Yang did, we adopt the worst case number rather than the amortized one because of the following reasons:

1. The worst case time complexity of an algorithm can be easily analyzed by just inspecting the algorithm.
2. To achieve low amortized time complexity, an algorithm may assign some process to service other processes. However, such a process is not equally treated. This unfairness will be revealed if we consider the worst case number.

Throughout the rest of the paper, time complexity means the worst case time complexity.

In the literature, with some read-modify-write (RMW) primitives in addition to atomic *read* and *write*, many mutual exclusion algorithms of constant time complexity are proposed; see Anderson et al.'s survey paper [3]. Because of these constant time algorithms, the asymptotic tight bound on time complexity is $\Theta(1)$. From a theoretical point of view, constant time is the best an algorithm can achieve in time complexity. Nevertheless, some researchers such as Fu and Tzeng [16,19] continue to strive for minimizing the number of RMR steps. We consider it worthwhile to reduce the number as much as possible. In practice, RMRs are orders of magnitude slower than references to the local memory. And mutual exclusion is a basic synchronization mechanism frequently used in multiprocessing systems both at the operating system kernel level and the users' application level [23]. Consequently, minimizing the number of RMR steps yields considerable performance improvement.

The primary result of this paper is a tight bound on the number of RMR steps needed to solve the mutual exclusion problem in DSM systems. We first present an algorithm whose time complexity is three in DSM systems, and then prove three is a lower bound on time complexity. This bound is therefore tight.

Our algorithm is inspired by the mutual exclusion algorithm proposed by Mellor-Crummey and Scott [23], also known as the MCS lock, and the one by Fu and Tzeng [16,19]. Fu and Tzeng tried to improve the MCS lock, whose time complexity is four, and obtained a better algorithm in terms of amortized time complexity. But, in Fu and Tzeng's algorithm, some process in its exit region (i.e., the code fragment after executing its critical region) may take an unbounded number of RMR steps for the purpose of scheduling other competing processes. Thus, the worst case number of RMR steps taken by some process is unbounded, i.e., the time complexity is unbounded. We follow the line of their algorithm but eliminate the above drawback.

We prove the time bound in an asynchronous DSM model where processes communicate by means of a general RMW primitive. The general RMW primitive atomically accesses one shared variable, reading the value of the variable and writing back a new value according to the submitted function. Let V be the set of all possible values for the variable. The submitted function can be any function $f : V \to V$. Hence, the general RMW primitive is a generalization of all atomic primitives that access at most one shared variable, and therefore our lower bound holds for any set of such primitives. In practice, almost all commonly available primitives implemented in multiprocessor systems—such as *read/write*, *test&set*, *compare&swap*, *fetch&add*, *fetch&increment*, *fetch&store*, *fetch-and-*ϕ—access one shared variable. Thus, the general RMW primitive can be used to model these primitives. For instance, a *read* primitive is equivalent to the general RMW primitive with the identity function (write the same value as that returned by the read), and a *write* primitive is equivalent to the general RMW primitive with the constant function that always maps to the new value (write the new value and discard the returned value). Two more examples appear in Section 3.1. Formally, the general RMW primitive is defined below, where v is the shared variable and f is the submitted function.

```
RMW (variable $v$, function $f$)
    $previous := v$
    $v := f(v)$
    return $previous$
```

Related lower bounds: Several related lower bounds have been proved in the literature. All of these bounds are asymptotic. Anderson and Yang [5] first initiated a series of studies of lower bounds on time complexity. They established a trade-off between the amount of contention, which was defined by Dwork et al. [15], and time complexity. The amount of contention of an algorithm is the maximum number of processes that are enabled to access the same shared variable simultaneously. Since our aim is minimizing the number of RMR steps, we focus on the time complexity when contention may equal the number of all processes. Applying their result to the model with the general RMW primitive, we have that $\Omega(\log_c n)$ RMR steps are required in both DSM and CC systems, where c is the amount of contention and n is the number of processes. Thus, the lower bound on time complexity is $\Omega(1)$, a trivial bound, when contention is n. Then, Cypher [13] showed a lower bound of $\Omega(\log\log n/\log\log\log n)$ on time complexity in DSM and CC systems with only atomic *read* and *write* primitives. This result implies that there is no constant time mutual exclusion algorithm if only *read* and *write* are available. He went on to show that the lower bound holds even if comparison primitives (e.g., *test&set* and *compare&swap*) are available in addition to *read* and *write*. In a later work, Anderson and Kim [2] improved Cypher's lower bound to $\Omega(\log n/\log\log n)$. Cypher's lower bound and the improved bound by Anderson and Kim hold for *read*, *write* and comparison primitives, whereas ours holds for all commonly available primitives that access at most one shared variable in an instruction.

In addition, Kim and Anderson [21] provided a time complexity lower bound for *adaptive* mutual exclusion algorithms in which time complexity is a function of the number of contending processes. They showed that for any k, there exists some n such that, for any n-process mutual exclusion algorithm based on *read*, *write* or comparison primitives, there exists an execution involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ RMR steps to enter and exit its critical region. The result applies to both DSM and CC systems. In another paper [1], Anderson and Kim showed that for any n-process mutual exclusion based on *non-atomic read* and *write*, there exists an execution involving only one process in which that process performs $\Omega(\log n/\log\log n)$ RMR steps in DSM systems to enter its critical region. Moreover, these RMR steps must access $\Omega(\sqrt{\log n/\log\log n})$ distinct remote shared variables, which implies that the process performs $\Omega(\sqrt{\log n/\log\log n})$ RMR steps in CC systems to enter its critical region.

Unlike the researchers who provided related lower bounds, we establish a lower bound only for DSM systems; the lower bound proof herein is not applicable to CC systems. As a result, a problem left open by the paper is what lower bounds are obtainable for CC systems.

Contribution: We improve the tight bound of mutual exclusion algorithms on time complexity from $\Theta(1)$ to three in DSM systems. From the complexity-theoretic point of view, it may not be so surprising. But, this result is of importance for algorithm designers. Focus of mutual exclusion algorithms for shared memory systems for the last 15 years has been on minimizing the number of RMRs [12,16,18,19,23]. The tight bound shows that it is impossible to obtain any better algorithm than ours in terms of minimizing the number.

The rest of the paper is organized as follows. Section 2 provides the system model and definitions. Section 3 presents an optimal algorithm establishing our upper bound for the mutual exclusion problem. Section 4 proves the lower bound. Finally, Section 5 concludes the paper.

2. System model and definitions

In this section, we first describe a model of asynchronous DSM systems. The salient features of the model are that:

1. each process has a segment of shared memory that is local to it, and
2. processes communicate by means of RMW primitives which atomically access one shared variable.

We adopt the definition of a RMR step proposed by Anderson and Yang [5], and thus use the number of RMR steps as our time complexity metric. Next, we define an indistinguishability relation on system states. Finally, we give a formal definition of the mutual exclusion problem, which is similar to the definition proposed by Burns et al. in [8].

2.1. Distributed RMW shared memory model

An algorithm in a distributed RMW shared memory system is modelled as a triple $(\mathcal{P}, \mathcal{V}, \delta)$, where $\mathcal{P}$ is a non-empty finite set of processes, $\mathcal{V}$ is a non-empty finite set of shared variables, and δ is a transition relation for the entire system.

$\mathcal{V}$ is the set of all shared variables every process can access. $\mathcal{V}$ is partitioned into disjoint non-empty subsets $\mathcal{V}_i$ for each $i \in \mathcal{P}$. In other words, each variable belongs to a segment of shared memory that is local to a single process. This captures the essence of DSM systems. $\mathcal{V}_i$ denotes the set of all shared variables located at process i. To a process i, a shared variable v is *remote* if $v \notin \mathcal{V}_i$; otherwise, it is *local*. In addition, let I_v, a subset of the value set of v, denote the possible initial values of v.

Each process $i \in \mathcal{P}$ is associated with a kind of state machine consisting of the following components:

- Σ_i: a (possibly infinite) set of states;
- I_i: a subset of Σ_i, indicating the initial states;
- Π_i : $\{(v, f)_i \mid v \in \mathcal{V}$ and f is a function mapping from the value set of v to the same set$\}$. Informally, Π_i specifies the steps that i may execute. Each step $(v, f)_i$ is a RMW operation that atomically reads the current value of v, say *old*, and writes back $f(old)$ to the same variable v. That is, step $(v, f)_i$ means that process i accesses v by executing RMW(v, f).

For a step $(v, f)_i \in \Pi_i$, we say that this step of process i *accesses* the shared variable v. It is an RMR *step from* i if $v \notin \mathcal{V}_i$. That is, the step accesses a shared variable located at some other process. *An RMR step to* j is an RMR step from $i \neq j$ that accesses a shared variable $v \in \mathcal{V}_j$.

A *system state* is a tuple consisting of the state of each process in $\mathcal{P}$ and the value of each shared variable in $\mathcal{V}$. System states will be denoted by s and t with subscripts and superscripts. For a system state s, we write $s(i)$, $i \in \mathcal{P}$, to denote the state of process i at s, and $s(v)$, $v \in \mathcal{V}$, to denote the value of shared variable v at s. An *initial system state* is a system state s at which $s(i) \in I_i$ for each process $i \in \mathcal{P}$ and $s(v) \in I_v$ for each shared variable $v \in \mathcal{V}$.

The transition relation δ is a set of (s, e, s') triples, where s and s' are system states, and e is a step of some process. We assume that δ satisfies the following assumptions:

Localized update: Suppose $(s, (v, f)_i, s')$ is a transition in δ, where $(v, f)_i$ is a step of process i.

1. *Suppose* $(t, (v, f)_i, t')$ *is an arbitrary transition in* δ, *with the same step of* i. *If* $s(i) = t(i)$ *and* $s(v) = t(v)$, *then* $s'(i) = t'(i)$.
   Informally, the present state of i and the present value of v uniquely determine the state of i after i takes step $(v, f)_i$.
2. $s'(v) = f(s(v))$.
   The new value of v is determined by the function f and the current value of v.

3. *$s'(j) = s(j)$ for all $j \in \mathcal{P}\backslash\{i\}$, and*
   $s'(u) = s(u)$ for all $u \in \mathcal{V}\backslash\{v\}$.
   Only the state of process i and the value of variable v can be affected.

Localized enabling: *If $(s, (v, f)_i, s') \in \delta$, then for any system state t at which $t(i) = s(i)$ holds, there exists a system state t' such that $(t, (v, f)_i, t') \in \delta$.*

We say that a step $e = (v, f)_i$ is *enabled* at system state s if there exists a system state s' such that $(s, e, s') \in \delta$. "Localized enabling" means that whether or not a step of a process is enabled at a system state depends only on the state of the process. Namely, if a step of process i is enabled at system state s, then the step is also enabled at any system state t at which $t(i) = s(i)$ holds.

Determinism: *For any process at any system state, there is at most one step of that process enabled.*

If a step $e = (v, f)_i$ is enabled at system state s, the resulting system state after i takes the step is unique since the new state of i and the new value of v are uniquely determined in the model. Therefore, we write $e(s)$ to denote the resulting system state.

An *execution fragment* is a finite or infinite sequence of steps. Several notations regarding execution fragments will be used in the sequel. Let α and α' be execution fragments.

- $|\alpha|$: The length of α (if α is a finite fragment).
- $\alpha|i$: The subsequence of α consisting of all steps of process i in α.
- $Pro(\alpha)$: The set of processes that take at least one step in α.
- $Var(\alpha)$: The set of shared variables accessed by any step in α.
- $\alpha \circ \alpha'$: The execution fragment obtained by concatenating α and α', provided that α is finite.

In addition, we say that α is a *P-execution fragment* if all processes involved in α are included in P (i.e., $Pro(\alpha) \subseteq P$), where P is a subset of $\mathcal{P}$. When $P = \{i\}$ we write i-execution fragment instead of $\{i\}$-execution fragment.

A finite execution fragment $e_1 e_2 \dots e_n$ is *executable from a system state s* if for all i, $n \geqslant i \geqslant 1$, e_i is enabled at s_{i-1} where $s_0 = s$ and $s_i = e_i(s_{i-1})$. Likewise, an infinite execution fragment $e_1 e_2 \dots$ is executable from a system state s if for all $i \geqslant 1$, e_i is enabled at s_{i-1} where $s_0 = s$ and $s_i = e_i(s_{i-1})$. If α is a finite execution fragment executable from s, we write $\alpha(s)$ to denote the system state after performing α from s. An *execution* is an execution fragment that is executable from an initial system state. A system state s is said to be *reachable* if there exists a finite execution such that the resulting system state is s.

Indistinguishability: Variants of the notion of indistinguishability are frequently used to prove impossibility results in distributed systems [22]. Here, we first define an equivalence relation among system states, and then propose several ways to manipulate execution fragments.

Let P be a subset of $\mathcal{P}$ and V a subset of $\mathcal{V}$. System states s and t are said to be *indistinguishable to P with respect to V*, denoted by $s \underset{V}{\overset{P}{\sim}} t$, if

1. $s(i) = t(i)$ for each $i \in P$, and
2. $s(v) = t(v)$ for each $v \in V$.

Informally, for system states s and t with $s \underset{V}{\overset{P}{\sim}} t$, s and t are indistinguishable to those processes in P consulting only shared variables in V. When $P = \{i\}$, we write $s \underset{V}{\overset{i}{\sim}} t$ instead of $s \underset{V}{\overset{\{i\}}{\sim}} t$.

Our definition is a generalization of the indistinguishability relation defined by Lynch [22]: when $V = \mathcal{V}$, the two indistinguishability relations become equal. The generalized version of indistinguishability makes it easier to define a weaker condition imposed on two system states such that an execution fragment executable from one system state is also executable from the other. Intuitively, it is enough to consider the set of all shared variables accessed in the execution fragment rather than the whole set $\mathcal{V}$. Furthermore, for a shared memory model whose memory has locality, this definition is useful in characterizing properties related to local shared memory, as we will see in Lemma 2 below and Lemma 9 in Section 4.1.

Now, we present two lemmas about ways to manipulate execution fragments based on the indistinguishability relation defined above. These lemmas can be easily proved by the localized update and localized enabling assumptions.

Suppose that execution fragment α is executable from system state s. Let $P = Pro(\alpha)$ and $V = Var(\alpha)$. If $s \underset{V}{\overset{P}{\sim}} t$, Lemma 1 says that α is also executable from system state t. This is because each process and each shared variable involved in α have the same state and the same value, respectively, in s and t. By the localized update and localized enabling assumptions, an induction on each prefix of α can show that α is also executable from system state t. If, in addition, α is finite, the resulting system states $\alpha(s)$ and $\alpha(t)$ will be also indistinguishable to P with respect to V, i.e., $\alpha(s) \underset{V}{\overset{P}{\sim}} \alpha(t)$.

Lemma 1. *Let s and t be system states. Suppose that α is an execution fragment executable from s. Let $P = Pro(\alpha)$ and $V = Var(\alpha)$. If $s \underset{V}{\overset{P}{\sim}} t$, then α is also executable from t. If, in addition, α is finite, then $\alpha(s) \underset{V}{\overset{P}{\sim}} \alpha(t)$.*

Proof. Suppose that $s \underset{V}{\overset{P}{\sim}} t$, that is, each process and each shared variable involved in α have the same state and the same value, respectively, in s and t. According to the localized update and localized enabling assumptions, a straightforward induction proves that for each prefix α' of α, α' is also executable from t and furthermore at the resulting system states $\alpha'(s)$ and $\alpha'(t)$, the states of all processes in P and the values of all shared variables in V are the same. □

Lemma 2 is for system states s and t that are indistinguishable to a process i consulting only shared variables in $\mathcal{V}_i$. Informally, if an execution fragment α executable from system state s contains neither RMR steps from i nor RMR steps to i, no communication between i and any other process can occur in α. Lemma 2 says that $\alpha|i$ is also executable from all system states t at which $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$ holds. If, in addition, α is finite, then the resulting system states $\alpha(s)$ and $(\alpha|i)(t)$ will be also indistinguishable to process i with respect to $\mathcal{V}_i$.

Lemma 2. *Let s and t be system states and i a process. Suppose α is an execution fragment that is executable from s and contains neither RMR steps from i nor RMR steps to i. If $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$, then $\alpha|i$ is also executable from t. If, in addition, α is finite, then $\alpha(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(t)$.*

Proof. Since α contains neither RMR steps from i nor RMR steps to i, i does not access any remote shared variable and no other process accesses any shared variable located at i in α. Thus, when α is executed from s, the state of i and the values of all shared variables located at i depend only on $\alpha|i$. Therefore, $\alpha|i$ is also executable from s and if, in addition, α is finite, $\alpha(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(s)$.

Suppose that $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$. We show that $\alpha|i$ is also executable from t. Since $\alpha|i$ is an i-execution fragment and i does not access any remote shared variable in $\alpha|i$ (i.e., $Var(\alpha|i) \subseteq \mathcal{V}_i$), $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$ implies $s \underset{V}{\overset{P}{\sim}} t$ where $P = Pro(\alpha|i) = \{i\}$ and $V = Var(\alpha|i)$. Hence, by Lemma 1, $\alpha|i$ is also executable from t and if, in addition, α is finite, $(\alpha|i)(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(t)$.

If α is finite, since $\alpha(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(s)$ and $(\alpha|i)(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(t)$, we have $\alpha(s) \underset{\mathcal{V}_i}{\overset{i}{\sim}} (\alpha|i)(t)$. $\square$

When α ending with an RMR step from i satisfies the assumptions on α in Lemma 2 except the last step, the following corollary says that $\alpha|i$ is also executable from t. Let α' be the prefix of α, just excluding the last step of α. By Lemma 2, $\alpha'|i$ is also executable from t and the states of i at $\alpha'(s)$ and $(\alpha'|i)(t)$ are the same. Thus, the RMR step from i at the end of α is also enabled at $(\alpha'|i)(t)$. Namely, the execution fragment $\alpha|i$ ($\alpha|i = \alpha'|i \circ$ the RMR step from i) is also executable from t. However, since the last step from i is an RMR step, the state of i at $\alpha(s)$ might be different from that at $(\alpha|i)(t)$.

Corollary 3. *Let s and t be system states and i a process. Suppose α is a finite execution fragment that is executable from s, ends with an RMR step from i, and contains neither RMR steps from i nor RMR steps to i except the last step. If $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$, then $\alpha|i$ is also executable from t.*

2.2. The mutual exclusion problem

So far, we have described a DSM model for all algorithms in general. For mutual exclusion algorithms in particular, we now specify requirements to capture the mutual exclusion behavior among a set of processes.

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 critical region. In order to gain admission to the critical region, a process executes its *trying region* code, and when a process leaves its critical region, it executes the *exit region* code for synchronization purposes and then returns to the rest of its code, called the *remainder region*.

For each process i, Σ_i is partitioned into non-empty disjoint subsets R_i, T_i, C_i and E_i. We say that *a process i is in its 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. In each initial system state, we assume that each process is in its region R. In addition, we assume that the transition relation δ for a mutual exclusion algorithm satisfies the following *well-formedness* conditions:

- If $(s, (v, f)_i, s') \in \delta$ and $s(i) \in R_i$, then $s'(i) \in R_i \cup T_i$.
- If $(s, (v, f)_i, s') \in \delta$ and $s(i) \in T_i$, then $s'(i) \in T_i \cup C_i$.
- If $(s, (v, f)_i, s') \in \delta$ and $s(i) \in C_i$, then $s'(i) \in C_i \cup E_i$.
- If $(s, (v, f)_i, s') \in \delta$ and $s(i) \in E_i$, then $s'(i) \in E_i \cup R_i$.

That is, each process cycles through its remainder, trying, critical and exit regions, in that order.

For all steps, we assume that a step enabled in the region R or C never accesses a shared variable that may be accessed by a step enabled in the region T or E. Thus, a step taken in the region R or C will not affect the processes in their regions T and E. Since all RMR steps of interest in this paper are those that are taken in the regions T and E, we assume, without loss of generality, that a process in its region R or C will not take any RMR step.

In addition, an algorithm that solves the mutual exclusion problem must meet the two basic conditions below.

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

The next condition depends on an assumption about the scheduling of processes in executions: no process "halts" anywhere except possibly in the region R. Executions with this property are said to be *admissible*. Let α be an execution executable from an initial system state s. Formally, α is *admissible* from s if for every process $i \in \mathcal{P}$ that takes only finitely many steps in α, i's final state belongs to R_i.

Progress: Let α be an admissible execution executable from an initial system state s and α_1 be any finite prefix of α. At system state $\alpha_1(s)$,

- if at least one process is in the region T and no process is in the region C, then there exists a finite prefix α_2 of α, $|\alpha_2| > |\alpha_1|$, such that some process enters its region C at $\alpha_2(s)$;
- if at least one process is in the region E, then there exists a finite prefix α_2 of α, $|\alpha_2| > |\alpha_1|$, such that some process enters its region R at $\alpha_2(s)$.

This condition is necessary for the system to make any progress at all. However, an algorithm satisfying the condition does not guarantee that the critical region is granted fairly to each individual process. A situation in which some process is denied indefinitely access to its region *C* (known as lockout, or starvation) may occur. Thus, it is often desirable to have some level of fairness of granting the region *C*.

An algorithm is *lockout-free* provided that it guarantees, assuming that no process stays in the region *C* indefinitely and the execution is admissible, no process can be kept waiting indefinitely either for the region *C* or for the region *R*. It is intuitively clear that a lockout-free algorithm is also an algorithm satisfying the progress condition.

An algorithm is *bounded-bypass* if it is *b-bounded-bypass* for some constant *b*. We say that an algorithm is *b-bounded-bypass* if after a process *i* has performed a step in its region *T*, process *i* cannot be bypassed more than *b* times by any particular process in competing for the region *C*.

In the lower bound proof (see Section 4), we assume only the two basic conditions of the mutual exclusion problem. Thus, the lower bound proof also works when we consider lockout-freedom and bounded bypass. Besides, our algorithm with the matching lower bound also satisfies these two conditions. Hence, the same tight bound holds for lockout-free mutual exclusion and bounded-bypass mutual exclusion.

Time complexity: The time complexity of a mutual exclusion algorithm is the worst case number of RMR steps taken by any single process in its region *T* and the following region *E* if the process enters and then leaves its region *C*, i.e., the worst case number of RMR steps for any single process to enter and then exit its region *C* once.

Then, a local-spin mutual exclusion algorithm can be formally defined as follows. This definition has been used implicitly or explicitly in related work about local-spin algorithms [3].

Definition 1. A mutual exclusion algorithm is *local-spin* if its time complexity is bounded, that is, a constant c exists such that its time complexity is less than or equal to c.

3. A time complexity upper bound

This section presents an algorithm whose time complexity is three. The key to minimizing the number of RMR steps is encoding different messages into an RMR step. Additionally, the mutual exclusion algorithm is bounded-bypass and lockout-free.

3.1. The primitives

Besides atomic *read* and *write*, two RMW primitives are used in our algorithm: (1) *fetch&store*(v, new), which atomically writes value *new* to shared variable v and returns the old value; and (2) *compare&swap*(v, old, new), which atomically writes value *new* to shared variable v exactly if its old value equals *old*, and returns the old value regardless of what happens in the comparison.

It is not hard to show that these two RMW primitives are special cases of the general RMW primitive. Primitive *fetch&store*(v, new) is equivalent to RMW(v, f) where f is a constant function that always maps to value *new*, and *compare&swap*(v, old, new) is equivalent to RMW(v, f) where f is a function defined as follows: let x be any value in the value set of v

$$f(x) = \begin{cases} new & \text{if } x = old, \\ x & \text{otherwise.} \end{cases}$$

Since all primitives used in the algorithm can be replaced by the general RMW primitive, the algorithm is indeed an upper bound result in our model.

3.2. The algorithm

We prove the following theorem by presenting our algorithm shown in Fig. 3. Fig. 4 is an example to help explain how the algorithm works. For ease of explanation, we let each process have several private variables. These private variables are part of the process's state and are unaccessible to other processes.

Theorem 4. *There is a mutual exclusion algorithm whose time complexity is three.*

Before presenting the algorithm, we explain how the MCS lock [23] schedules requests to the critical region in an orderly way using *fetch&store* and *compare&swap*. This inspires our algorithm. As shown in Fig. 1, the MCS lock uses a *fetch&store* on a lock to chain competing processes as a list. Each process desiring to enter its critical region executes *fetch&store* on the shared variable *L* (i.e., the lock), announcing its identity and obtaining the identity of its predecessor if there is one ($T1$). If the returned value is *nil*, i.e., the requesting process is the head of the list, then it immediately enters its critical region. Otherwise, if it has a predecessor, it first writes a value to its predecessor's *Next* variable, notifying its predecessor to refer back to its identity ($T3$). It then starts to spin on a locally accessible shared variable until it is awakened ($T4$).

In the region *E*, a process *i* passes the permission to its successor if there is one. If *Next*$(i) \neq \perp$, i.e., *i*'s successor has updated *Next*(i), then *i* updates its successor's spin variable ($E8$). Otherwise, two cases are possible: (1) *i* has no successor, or (2) *i* does have a successor, but the successor has not yet updated *Next*(i). Primitive *compare&swap* in $E2$ enables *i* to determine which case is true. If the returned value of *compare&swap* is not *i*, i.e., *i* indeed has a successor, *i* waits until its successor updates *Next*(i) ($E3$), and then wakes up its successor ($E5$). Otherwise, if the returned value of *compare&swap* is *i*, i.e., *i* has no successor, then *compare&swap* has modified *L*'s value to *nil*, setting the system state to the starting state.

Fig. 2 illustrates a simple execution of the MCS lock. Process 3 first executes *fetch&store* in $T1$ and gets *nil* from *L*, so it enters its region *C* immediately. While process 3 is in its region *C*, processes 1, 5 and 4 execute $T1$ in turn. Each of processes 1, 5 and 4 updates its predecessor's *Next* variable and then starts to wait. The permission is conveyed from 3 to 1, then from 1 to 5, and then from 5 to 4. After process 4 leaves its region *C*, if there is no other request, process 4 modifies *L*'s value to *nil*; otherwise, it passes the permission to its successor.

```
Shared variables:
$L \in \{nil, 0, 1, \ldots, n-1\}$, initially $nil$                ▷ $L$ can be located at any process
for every $i \in \{0, \ldots, n-1\}$:
  $Spin(i) \in \{true, false\}$, initially $true$
  $Next(i) \in \{\perp, 0, \ldots, n-1\}$, initially $\perp$
                                        ▷ $Spin(i)$ and $Next(i)$ are located at process $i$
Process $i$ :   $(i \in \{0, \ldots, n-1\})$

Private variables of $i$:
$pred, suc \in \{nil, 0, 1, \ldots, n-1\}$, initially arbitrary


     while $true$ do
R:       Remainder region
T1:      $pred := fetch\&store(L, i)$;
T2:      if $pred \neq nil$ then
T3:          $Next(pred) := i$;
T4:          await $\neg Spin(i)$; fi              ▷ locally spin until $Spin(i) = false$
C:       Critical region
E1:      if $Next(i) = \perp$ then
E2:          if $compare\&swap(L, i, nil) \neq i$ then
E3:              await $Next(i) \neq \perp$;      ▷ locally spin until $Next(i)$ is updated
E4:              $suc := Next(i)$;
E5:              $Spin(suc) := false$; fi                ▷ wake up its successor
E6:      else
E7:          $suc := Next(i)$;
E8:          $Spin(suc) := false$;                       ▷ wake up its successor
E9:      fi
E10:     $Spin(i) := true$;                              ▷ set $Spin(i)$ to $true$
E11:     $Next(i) := \perp$;                             ▷ set $Next(i)$ to $\perp$
     od
```

Fig. 1. The MCS lock.

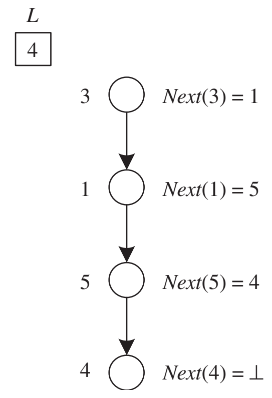

Fig. 2. An execution of the MCS lock. An arrow from node p to note q indicates that process q has updated process p's *Next* variable so that p knows the identity of its successor.

As in the MCS lock, we use a *fetch&store* on a lock to link competing processes, but, as in Fu and Tzeng's algorithm [16], we eliminate the RMRs needed in the MCS lock to notify its predecessor to re-direct the link for each process in a list. With this modification, Fu and Tzeng devised a way to pass the lock among processes. However, their way suffers from blocking in the exit region. To eliminate this drawback, we provide a new way to convey the lock.

We first give an informal description of the algorithm and then describe it in more detail. In the algorithm for n processes, each process $i \in \mathcal{P} = \{0, \ldots, n-1\}$ has two identities, i and $n+i$. For brevity, let $\bar{i}$ denote $n+i$. Each process uses different identities in any two consecutive life cycles to avoid a subtle situation. We defer the explanation of the subtlety until we have presented the algorithm (Fig. 3).

We now explain the key idea of the algorithm. Each requesting process executes *fetch&store* on the shared variable L (i.e., the lock) to announce its identity and obtain its predecessor's identity if there is one. If the returned value is *nil*, the critical region is available and the requesting process enters the critical region immediately; otherwise, it waits by repeatedly testing its local-spin variable. Since each process makes a request by executing *fetch&store* on the same variable L, a waiting list will be formed if some process has been in its region C. For instance, in Fig. 4(a), as process 3 is in its region C, all competing processes (1, 5, and 4) form a waiting list.

When a process leaves its region C, it takes an RMR step to write a value, called the permission word, to the spin variable of some waiting process. Since the waiting process is testing its spin variable repeatedly, the permission word in effect serves as a wake-up signal. In order to minimize the number of RMRs, the permission word not only serves as permission to enter the region C, but also carries enough information for processes to arrange among themselves the order to enter the region C, without using any other control word.

The permission will be conveyed in the following way. First, any process that succeeded in acquiring *nil* from L enters the region C. When such a process leaves its region C, it conveys the permission to the tail of the current waiting list. Then, the permission will be transmitted along the list from the tail to the

```
Shared variables:
$L \in \{nil, 0, 1, \dots, 2n-1\}$, initially $nil$              ▷ $L$ can be located at any process
for every $i \in \{0, \dots, n-1\}$:
    $Spin(i) \in \{(head, tail) \mid head, tail \in$
        $\{\perp, 0, 1, \dots, 2n-1\}\}$, initially $(\perp, \perp)$      ▷ $Spin(i)$ is located at process $i$

Process $i$ :   $(i \in \{0, \dots, n-1\})$

Private variables of $i$:
$id \in \{i, n+i\}$, initially $i$
$pred \in \{nil, 0, 1, \dots, 2n-1\}$, initially arbitrary
$h, t \in \{\perp, 0, 1, \dots, 2n-1\}$, initially arbitrary

   while true do
R:      Remainder region
T1:     $pred := fetch\&store(L, id)$;
T2:     if $pred \neq nil$ then
T3:         await $Spin(i) \neq (\perp, \perp)$; fi
C:      Critical region
E1:     $(h, t) := Spin(i)$;
E2:     if $pred = nil$ or $pred = h$ then                    ▷ as a controller
E3:         if $pred = nil$ then                     ▷ E3–E8 encode the permission word
E4:             $h := id$;
E5:         else
E6:             $h := t$;
E7:         fi
E8:         $t := compare\&swap(L, h, nil)$;
E9:         if $t \neq h$ then                      ▷ wake up the tail of the waiting list
E10:            $Spin(t \bmod n) := (h, t)$; fi
E11:    else                                                 ▷ as a non-controller
E12:        $Spin(pred \bmod n) := (h, t)$;                  ▷ wake up its predecessor
E13:    fi
E14:    $Spin(i) := (\perp, \perp)$;                  ▷ set the spin variable to $(\perp, \perp)$
E15:    $id := (id + n) \bmod 2n$;                              ▷ change the identity
   od
```

Fig. 3. Our algorithm.

head, allowing every process in the list to enter its region C in an orderly way. While the permission is being transmitted, all subsequent requesting processes form a new waiting list appending to the tail of the old list. Once the head of the old list leaves its region C, i.e., all processes in the list have finished their regions C, the permission will be redirected to the tail of the new waiting list. Similarly, the permission will be conveyed along the new list. We call a process that redirects the permission to the tail of a new waiting list a *controller*. Namely, a process is a *controller* if it gets *nil* from L or it is the head of a waiting list. In addition, a controller has the responsibility to encode some information into the permission so that each process in a new list can check whether it is the head of the list and if so, it should take the role of a new controller. If there is no new waiting list when a controller tries to redirect the permission, the controller modifies L's value to *nil*, thus properly setting the system to the starting state. Using *compare&swap*, a controller can atomically check whether there is a new waiting list and if not, modify L's value to *nil*, avoiding any interleaving with processes that make requests about the same time.

For example, in Fig. 4(a), when process 3 (the controller at the time) leaves its region C, it conveys the permission to process 4, the tail of the current waiting list, called list 1. Pair (3,4) serves as the permission, where 3 is used for each process receiving the permission to check whether it is the head of list 1, and 4 indicates the tail of the list and will be used to encode the next permission. The permission will be transmitted along list 1. In Fig. 4(b), when process 1 in list 1 leaves its region C, i.e., all processes in the list have finished their regions C, process 1 knows that it is the head of list 1 by checking whether its predecessor is 3. Process 1 encodes new information into the permission and redirects it to the tail of the current waiting list, called list 2.

We now describe the algorithm in more detail. The algorithm uses $n+1$ shared variables: L and $Spin(i)$ for each $i \in \mathcal{P}$. L can be located at any process; in contrast, $Spin(i)$ must be located at process i. $Spin(i)$ is the spin variable of process i. Whenever busy-waiting is necessary, process i repeatedly checks its spin variable without causing any RMR. Each spin variable consists of two parts, (*head*, *tail*), each being the identity of a process or $\perp$. Initially, L is set to *nil* and each spin variable is set to $(\perp, \perp)$.

In the region T, a process executes *fetch&store* on L ($T1$). If the returned value of the primitive is *nil*, the requesting process enters its critical region immediately; otherwise, it waits by repeatedly testing its spin variable until the value is not equal to $(\perp, \perp)$ ($T3$).

In the region E, a process will identify itself as a controller if the result of checking $E2$ is "yes"—that is, *pred* is equal to *nil* or *head* of its spin variable. If the process is not a controller,

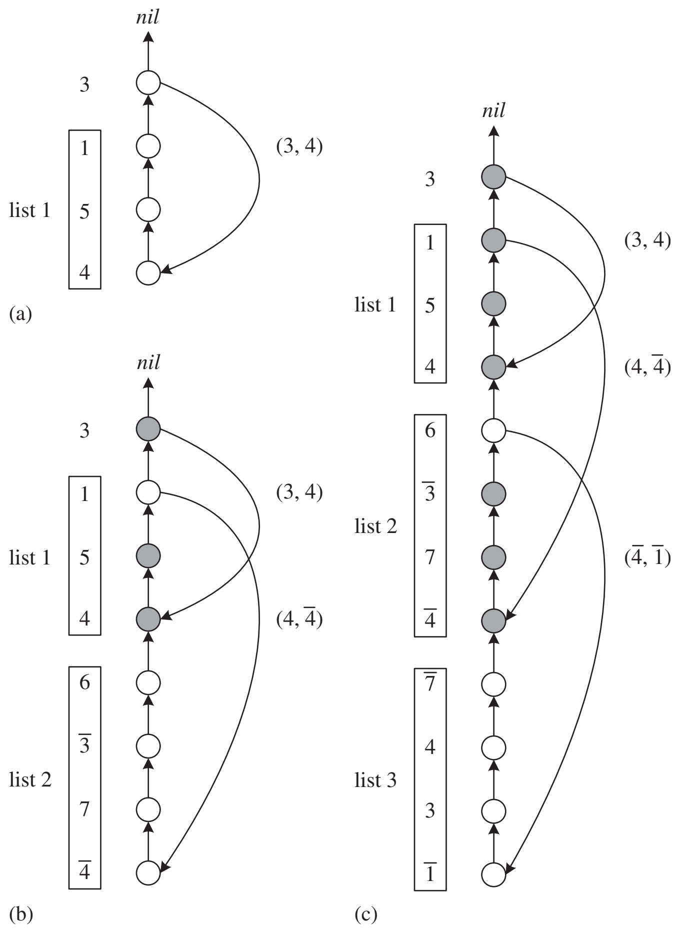

Fig. 4. An execution of our algorithm. A gray node indicates a process that has finished one life cycle. An upward arrow from a process points to the process's predecessor, and a downward arrow from a process, which must be a controller, points to the tail of the waiting list to which the process is responsible. The label of a downward arrow from a process represents the permission word conveyed to the tail by the process.

it just transmits the permission to its predecessor by executing $E12$. Otherwise, it first encodes new control information into the permission word $(head, tail)$ by executing steps $E3$–$E8$. Steps $E3$–$E7$ set the new value of *head*: if the controller gets *nil* from *L*, *head* is set to its current identity; otherwise, *head* is set to the value of *tail* in the old permission word. This is because the value of *head* will be used by processes in the new waiting list to check whether it is the head of the list. Step $E8$ sets *tail* to the returned value of *compare&swap* on *L*, which is the identity of the tail of the new waiting list if there is one. If there is no new waiting list, $E8$ atomically modifies *L*'s value to *nil*. Otherwise, the controller redirects the permission to the tail of the new list by executing $E10$.

We have presented the algorithm. It remains to explain the reason why each process uses different identities in any two consecutive life cycles. Each process alternately uses one of its identities to avoid a subtle situation. Although a process cannot appear more than once in a waiting list, it may appear in two neighboring lists. A process's identity in one life cycle is different from that in the next cycle since a process always changes its current identity in $E15$. Therefore, no two identities of the same process in any two consecutive lists are the same. This is important for a process to determine whether it should act as the controller for the next waiting list. For example, in Fig. 4(c), process 3 in list 3 would not be able to tell the difference between 4 in list 3 and $\bar{4}$ in list 2 if process 4 uses the same identity. With the different identities, process 3 should pass the permission to process 4, rather than taking up the role of a controller. The situation occurs whenever a process at the tail of a waiting list, after having been given permission to enter its region *C*, quickly makes a new request in the next waiting list. Fortunately, the subtlety needs to be resolved only between two neighboring waiting lists, thus two identities for each process suffice.

Time complexity: Inspecting the algorithm, it is easy to find that the worst case number of RMR steps taken by any single process in its regions *T* and *E* is three (steps $T1$, $E8$ and $E10$).

3.3. A correctness argument

3.3.1. Mutual exclusion

In the algorithm, a process *i* has permission to enter its region *C* exactly if it obtains *nil* from *L* when executing $T1$ (i.e., $pred = nil$) or $Spin(i) \neq (\perp, \perp)$. Since a process that obtains *nil* when executing $T1$ writes its identity, a non-*nil* value, to L in the same step, a *nil* in L permits at most one process to gain permission. Initially, L is set to *nil* and $Spin(i) = (\bot, \bot)$ for each process *i*. Thus, at most one process may enter the region C initially.

To prove mutual exclusion, we focus on steps that may cause some process to gain permission, that is, on steps that may set L to *nil* or modify some process's spin variable. Inspection of the algorithm clearly indicates that only steps $E8$, $E10$, and $E12$ need to be considered.

- Step $E8$ ($t :=$ *compare&swap*(L, h, nil)) assigns the current value of L to *t*, and modifies L's value to *nil* only if $L = h$. If the step indeed modifies L's value, it is regarded as *successful*. A successful $E8$ allows at most one process to gain permission.
- Each of $E10$ and $E12$ modifies some process's spin variable. Since the spin variables of any two processes are distinct, each of the two steps allows at most one process to gain permission.

According to the algorithm, a process that executes a successful $E8$ bypasses $E10$ since $t = h$. Hence, a process in its region E executes exactly one of the following steps: successful $E8$, $E10$, or $E12$. That is, a process in its region E passes its permission to at most one process.

Since at most one process may gain permission initially and each process having permission passes its permission to at most one process, the following lemma holds.

Lemma 5. *The algorithm guarantees mutual exclusion.*

3.3.2. Lockout-freedom

We now show that the algorithm is lockout-fee. This also implies that the algorithm satisfies progress.

Before proving the lockout-freedom condition, we present several definitions that intend to organize all requests in an execution. First, a *busy period* is an execution fragment that starts with a step $T1$ that succeeds in acquiring *nil* from L, and ends with the following successful $E8$, which modifies L's value to *nil*. Since $L = nil$ initially, all occurrences of $T1$ (i.e., all requests) in an execution can be divided into busy period(s). In a busy period, each requesting process except the first one has the identity of its predecessor because each process makes a request by executing $T1$ on the same shared variable L.

Next, we try to divide all requests in a busy period into *lists*. A *list* in a busy period is a sequence of processes that execute $T1$ between the first $T1$, which obtains *nil* from L, and the following unsuccessful $E8$, or between an unsuccessful $E8$ and the next unsuccessful one. 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. A process that executes $E8$ is called a *controller*. If a controller executes an unsuccessful $E8$, it defines a new list and is also called the controller of the new list. Otherwise, if a controller executes a successful $E8$, it ends the busy period.

Lemma 6. *The algorithm guarantees lockout-freedom.*

Proof. The argument for the exit region is simple. Since no loop occurs in the exit region, each process in its region E eventually enters its region R.

The lockout-freedom condition for the trying region is now considered. We argue that each requesting process in any busy period of an admissible execution eventually enters its region C.

In a busy period, the first $T1$ obtains *nil* from L and thus the first requesting process eventually enters its region C. When leaving its region C, the process identifies itself as a controller since $pred = nil$. After executing $E4$ to assign its current identity to *h*, it executes $E8$. When it executes $E8$, if $L = h$ (i.e., no other request exists), it modifies L's value to *nil* in the same step and ends the busy period. Otherwise, it defines the first list and is the controller of the list. We need to prove that each requesting process in the first list and all possible subsequent lists eventually enters its region C.

We show that each requesting process in the *i*th list, called list *i*, eventually enters its region C, and only the head of the list is selected as a new controller by induction on *i*.

Basis: $i = 1$. List 1 contains all processes that make requests between the first $T1$ in the busy period and the unsuccessful $E8$ executed by the controller of list 1. Through the returned value of *compare&swap* in $E8$, the controller has the identity of the tail of the list. Lemma 5 implies that at most one process has permission at any system state. Thus, before the controller passes the permission to the tail, all requesting processes will be blocked at $T3$. The controller then executes $E10$ to pass the permission to the tail by writing pair (h, t) to the tail's *Spin* variable, where *h* is the controller's identity and *t* is the tail's identity. In list 1, each process except the head will not be selected as a new controller since $pred \neq nil$ and $pred \neq h$, and will pass the permission to its predecessor by executing $E12$. Thus, the permission will be conveyed along the whole list from the tail to the head so that each process in list 1 eventually enters its region C. When the head of list 1 leaves its region C, it identifies itself as a new controller since its *pred* is equal to the previous controller's identity (i.e., $pred = h$).

Inductive step: Assume that each process in list *i* eventually enters its region C and only the head of the list is selected as a new controller. While the permission is conveyed along list *i*, all subsequent requesting processes, including those that are in list *i* and make requests again, will be blocked at $T3$.

According to the induction hypothesis, the head of list *i* identifies itself as a new controller after leaving its region C. The new controller executes $E6$ to assign the identity of the tail of list *i* to its *h*. Since a process always switches its identity in $E15$, if the process at the tail of list *i* makes a request, after having been given permission, it has a different identity. Thus, if $L = h$ when the new controller executes $E8$, no other request exists. The new controller modifies L's value to *nil* in the same step and ends the busy period. Otherwise, it defines list $i+1$ and is the controller of list $i+1$. List $i+1$ contains all requesting processes that make requests between the previous unsuccessful $E8$, which defines list *i*, and the unsuccessful $E8$ executed by the controller of list $i+1$. The controller then passes the permission to the tail of list $i+1$ by writing pair

(h, t) to the tail's *Spin* variable, where h is the identity of the tail of list i and t is the identity of the tail of list $i + 1$.

It remains to show that the permission will be conveyed along the whole list. Although the process at the tail of list i may be a member of list $i + 1$, it has a different identity when it appears in list $i + 1$. Therefore, in list $i + 1$, only the head will identify itself as a new controller when checking whether its predecessor's identity equals the identity of the tail of list i. The permission will be conveyed along the whole list so that each process in list $i + 1$ eventually enters its region C. □

3.3.3. *Bounded bypass*

In a busy period of an execution, since a list does not receive the permission until each process in the previous list has left its region C, the algorithm satisfies bounded bypass.

Lemma 7. *The algorithm guarantees bounded bypass.*

4. A time complexity lower bound

In this section we show that, under the system model and definitions in Section 2, the time complexity of any mutual exclusion algorithm with at least four processes is at least three.

Theorem 8. *Let A be a mutual exclusion algorithm for $n > 3$ processes. Then the time complexity of A must be three or more.*

Theorems 4 and 8 together imply the tight bound of three on time complexity.

This section is organized as follows. We first make a simplifying restriction on the mutual exclusion algorithms. Next, we present several properties of a process that is busy waiting only at certain local shared variable(s) in its region T, i.e., a process that is locally spinning in its region T. These properties will be used in our lower bound proof. Finally, we present the outline of the lower bound proof, and then show the detailed proof.

For simplicity, we make the following restriction on mutual exclusion algorithms: we only consider local-spin mutual exclusion algorithms. This entails no loss of generality, because the time complexity of a non-local-spin algorithm is unbounded.

4.1. *Basic properties*

We present three lemmas about a process that is locally spinning and show that for any local-spin mutual exclusion algorithm, there exists a reachable system state at which some process is locally spinning.

First, a definition is needed to describe a system state at which some process is locally spinning in its region T. Informally, a process i locally spinning in its region T at a system state s has two features: by running i alone from s, (1) i will not perform any RMR step; and (2) i will never change regions. The definition below tries to capture this notion.

Definition 2. Let s be a system state of a mutual exclusion algorithm. We say that process i is locally spinning in its region T at s if

1. i is in its region T at s, and
2. for any finite i-execution fragment α executable from s, α contains no RMR step and i remains in its region T from s to $\alpha(s)$.

The following lemma says that whether a process is locally spinning at a system state depends on the state of the process and the values of its local shared variables.

Lemma 9. *Let s and t be system states of a mutual exclusion algorithm such that $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$ for process i. Then i is locally spinning in its region T at s if and only if i is locally spinning in its region T at t.*

Proof.

1. ($\rightarrow$) Suppose i is locally spinning in its region T at s. Since i is in its region T at s and $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$, i is in its region T at t. It remains to show that for any finite i-execution fragment α executable from t, α contains no RMR step and i remains in its region T from t to $\alpha(t)$. By way of contradiction, suppose that α is a finite i-execution fragment executable from t such that α contains an RMR step or i changes regions in α. Let α' be the prefix of α including and ending with the first step that is either an RMR step or an operation that makes i change regions. Since $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} t$, α' is also executable from s. (If α' ends with an RMR step, this follows from Corollary 3; otherwise, this follows from Lemma 2.) This contradicts the assumption that i is locally spinning in its region T at s.
2. ($\leftarrow$) The other direction follows from symmetry. □

The next lemma says that starting from a system state at which process i is locally spinning, i will not perform any RMR step before any other process takes an RMR step to i.

Lemma 10. *Let s be a system state of a mutual exclusion algorithm at which process i is locally spinning in its region T. In any execution fragment executable from s, no RMR step from i exists before the first RMR step to i occurs.*

Proof. Suppose for the sake of contradiction that α is an execution fragment executable from s in which an RMR step from i exists before the first RMR step to i occurs. We construct an i-execution fragment that is executable from s but ends with an RMR step from i. This contradicts the assumption that i is locally spinning at s.

Let α' be the prefix of α including and ending with the first RMR step from i. Note that the assumption on α implies that α' contains no RMR step to i. We show that $\alpha'|i$ is also executable from s. This is the needed contradiction because $\alpha'|i$ ends with an RMR step from i. By the definition of α', it is executable from s, ends with an RMR step from i, and contains neither RMR steps from i nor RMR steps to i except the last step. Also, it is trivial that $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} s$. By Corollary 3, $\alpha'|i$ is also executable from s. □

Intuitively, if a process i is locally spinning in its region T at some point and i enters its region C at a later point, then some other process must have taken at least one RMR step to wake up i. The next lemma, also called the inherent cost lemma, formalizes this intuition. A similar observation in a message-passing model can be found in Chandy and Misra's work about "knowledge" among processes [10].

Lemma 11 (*inherent cost*). *Let s be a system state of a mutual exclusion algorithm at which process i is locally spinning in its region T. Suppose that process i reaches its region C in a finite execution fragment α executable from s. Then, α must contain at least one RMR step to i.*

Proof. By way of contradiction, suppose that α contains no RMR step to i. We construct an i-execution fragment that is executable from s but violates the assumption that i is locally spinning in its region T at s.

By Lemma 10, α contains no RMR step from i and therefore it contains neither RMR steps from i nor RMR steps to i. In addition, it is clear that $s \underset{\mathcal{V}_i}{\overset{i}{\sim}} s$. Thus, by Lemma 2, $\alpha|i$ is also executable from s and process i has the same state at $\alpha(s)$ and $(\alpha|i)(s)$. Since i is in its region C at $\alpha(s)$, i is also in its region C at $(\alpha|i)(s)$. Thus, $\alpha|i$ is the needed execution fragment because i changes regions in $\alpha|i$. □

Finally, the following lemma says that for any local-spin mutual exclusion algorithm, if some process has been in its region C at a system state, then running another requesting process i alone eventually leads to a system state at which i is locally spinning.

Lemma 12. *Let A be a local-spin mutual exclusion algorithm for $n > 1$ processes. Let s be a reachable system state of A at which process i is in its region R and some other process is in its region C. Then there exists a finite i-execution fragment α executable from s such that i is locally spinning in its region T at system state α(s).*

Proof. Starting from s, let i enter its region T and continue to run i alone. This must lead to a system state at which i is locally spinning since otherwise the time complexity would be unbounded or i would change regions. The former violates the assumption that A is a local-spin mutual exclusion algorithm; while, the latter violates the mutual exclusion condition. □

4.2. Proof outline

Throughout the rest of this paper, we let $A = (\mathcal{P}, \mathcal{V}, \delta)$ be an arbitrary local-spin algorithm for $n > 3$ processes and let s_{init} be an initial system state of A. To prove the lower bound of three on time complexity, our objective is to construct an

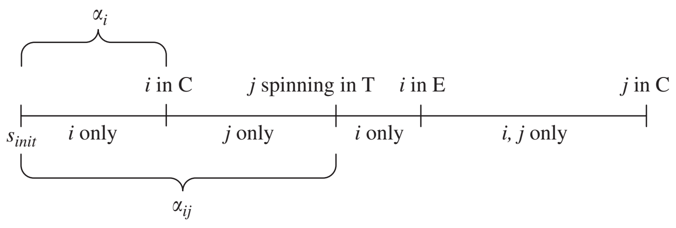

Fig. 5. A goal execution extended from α_{ij} in which $time(i, \alpha_{ij}) \geqslant 2$.

execution of A from s_{init} in which some process takes at least three RMR steps to enter and exit its region C once. We call such an execution a *goal execution*.

A goal execution will be constructed in the following way. We start by constructing n solo executions, one per process, each starting from s_{init} and involving its steps only until it has just entered its region C. (The progress condition implies that this is possible.) For each $i \in \mathcal{P}$, let α_i denote the solo execution of i. Next, for each α_i and each process $j \neq i$, we extend α_i to what is denoted by α_{ij} by running j alone until j has just entered a system state at which j is locally spinning in its region T. Execution α_{ij} exists according to Lemma 12. Our lower bound proof focuses on the set of all α_{ij}'s, called set $\mathcal{E}$. More precisely, we define

$$\mathcal{E} = \{\alpha_{ij} \mid i, j \in \mathcal{P} \text{ and } i \neq j\}.$$

We show that a goal execution can be constructed by extending some execution in $\mathcal{E}$.

Consider the number of RMR steps that have been taken by each process in each execution in $\mathcal{E}$. For brevity, let $time(i, \alpha_{ij})$ and $time(j, \alpha_{ij})$, respectively, denote the number of RMR steps taken by i and j in their regions T in α_{ij}. Then two cases are discussed.

Case 1: $\exists\, \alpha_{ij} \in \mathcal{E} : time(i, \alpha_{ij}) \geqslant 2$ or $time(j, \alpha_{ij}) \geqslant 2$. Let $\alpha_{ij} \in \mathcal{E}$ be such an execution. A goal execution can be extended from α_{ij} by applying the inherent cost lemma.

If $time(i, \alpha_{ij}) \geqslant 2$, we extend α_{ij} to an execution in which i takes at least one RMR step in its corresponding region E, so that i takes at least three RMR steps in total. The way to extend α_{ij} is described as follows and is illustrated in Fig. 5. From the end of α_{ij}, we let i leave its region C first and then let i and j take enabled steps alternately until j enters its region C. Since i and j take enabled steps alternately, this execution is admissible. Thus, the progress condition implies that j eventually enters its region C. By the inherent cost lemma, there exists at least one RMR step to j, which must be taken by i because only processes i and j are involved, in the portion of the resulting execution after α_{ij}.

If $time(i, \alpha_{ij}) \geqslant 2$ does not hold, it must be the case that $time(j, \alpha_{ij}) \geqslant 2$ holds. Similarly, by the inherent cost lemma, we extend α_{ij} to an execution in which j instead of i takes at least one RMR step in its corresponding region E. The construction will be given in the detailed proof.

Case 2: $\forall\, \alpha_{ij} \in \mathcal{E} : time(i, \alpha_{ij}) < 2$ and $time(j, \alpha_{ij}) < 2$. This case is the core of the lower bound proof. We construct a goal execution in which some process takes one RMR step in its

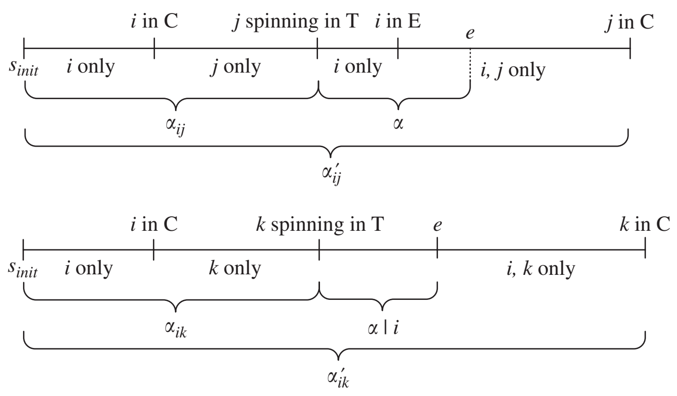

Fig. 6. A goal execution extended from either α_{ij} or α_{ik}. We write e to denote the RMR step from i to j.

region T and takes at least two RMR steps in its corresponding region E.

We first use the following property, called the *rendezvous property*, which says that in most executions in $\mathcal{E}$, processes communicate through the same remote shared variable: (The property will be proved in the next subsection.)

Suppose that for all $\alpha_{ij} \in \mathcal{E}$, processes i and j each access at most one remote shared variable in α_{ij}. Then there exists a shared variable v such that for all $\alpha_{ij} \in \mathcal{E}$ that v is remote to both i and j, both i and j must access v in α_{ij}. More precisely,

$$\exists\ v \in \mathcal{V}, \forall \alpha_{ij} \in \mathcal{E}, v \neq \mathcal{V}_i \text{ and } v \notin \mathcal{V}_j : \text{ both } i \text{ and } j \text{ must access } v \text{ in } \alpha_{ij}.$$

Since for all $\alpha_{ij} \in \mathcal{E}$, $time(i, \alpha_{ij}) < 2$ and $time(j, \alpha_{ij}) < 2$, i and j each access at most one remote shared variable in α_{ij}. Therefore, the above property guarantees the existence of such a shared variable v. Let m be the process to which v is local. We conclude that for each α_{ij} with $i \neq m$ and $j \neq m$, we always have $time(i, \alpha_{ij}) = 1$ and $time(j, \alpha_{ij}) = 1$. Furthermore, both i and j must access the same remote shared variable v.

Take any three distinct processes i, j and k that are different from m. Processes i, j and k exist since $n > 3$. Consider α_{ij} and α_{ik}. By the conclusion above, i and j each take exactly one RMR step in α_{ij}, and they access the shared variable v, which is located at process m. Likewise, i and k each take exactly one RMR step in α_{ik}, and they access v. A goal execution can be constructed by extending either α_{ij} or α_{ik}, in which i takes at least two more RMR steps in addition to the one it has taken in α_{ij} or α_{ik}. The construction is illustrated in Fig. 6.

First, we extend α_{ij} to α'_{ij} in the same way as that shown in Fig. 5, i.e., by letting i leave its region C and then running i and j until j reaches its region C. In the suffix of α'_{ij} after α_{ij}, if i takes at least two RMR steps, α'_{ij} is already a goal execution. Otherwise, the inherent cost lemma implies that i takes exactly one RMR step, which is from i to j. Based on this implication, a goal execution α'_{ik} is constructed below.

Execution α'_{ik} begins with α_{ik}, in which i has taken one RMR step. It then continues by letting process i run alone until it takes the RMR step to j as it does in the suffix of α'_{ij} after α_{ij}. This is possible mainly because $\alpha_{ij}(s_{\text{init}}) \overset{i}{\underset{\mathcal{V}_i}{\sim}} \alpha_{ik}(s_{\text{init}})$. (A precise argument will be given in the detailed proof.) It finishes by running processes i and k until k enters its region C; along the way, the inherent cost lemma guarantees that i must take at least one RMR step to k. Thus, i takes at least three RMR steps in α'_{ik}, and thereby α'_{ik} is a goal execution.

4.3. Detailed proof

We begin by proving the rendezvous property (Lemma 15) and then provide the detailed lower bound proof.

In order to prove the rendezvous property, we first present two lemmas for all α_i of A, $i \in \mathcal{P}$. The first, Lemma 13, says that for any two distinct solo executions α_i and α_j, there exists at least one shared variable that is accessed in both α_i and α_j. That is, processes i and j must access at least one common shared variable in their respective solo executions. The other, Lemma 14, says that if every $i \in \mathcal{P}$ accesses at most one remote shared variable in its α_i, then there is exactly one shared variable, say v, that is accessed in all α_i, $i \in \mathcal{P}$. That is, every process i must access v in its α_i. Note that unlike Lemma 14, Lemma 13 holds without any assumption on the number of remote shared variables accessed in each α_i.

For presenting Lemmas 13 and 14, we need a definition: for every shared variable v, let $\mathcal{P}_v$ denote the set of all processes that access v in their respective solo executions. That is, for every v in $\mathcal{V}$, define

$$\mathcal{P}_v = \{i \in \mathcal{P} \mid i \text{ accesses } v \text{ in } \alpha_i\}.$$

First, we prove Lemma 13, also called the pairwise common lemma. Informally, although α_i and α_j are two independent executions, processes i and j should access at least one common shared variable for synchronization purposes. For otherwise, it is easy to yield an execution in which both i and j are in their regions C simultaneously by concatenating α_i and α_j. This violates the mutual exclusion condition.

Lemma 13 (*pairwise common*). *For any two solo executions α_i and α_j, $i \neq j$, there exists at least one shared variable accessed in both α_i and α_j. More precisely, $\forall\, i, j \in \mathcal{P}, i \neq j, \exists\, v \in \mathcal{V} : \{i, j\} \subseteq \mathcal{P}_v$.*

Proof. By way of contradiction, suppose that there exists no shared variable accessed in both α_i and α_j. Thus, each shared variabl accessed in α_j has the same value at system states s_{init} and $\alpha_i(s_{\text{init}})$. In addition, process j has the same state at s_{init} and $\alpha_i(s_{\text{init}})$, and therefore we have $s_{\text{init}} \overset{P}{\underset{V}{\sim}} \alpha_i(s_{\text{init}})$, where $P = Pro(\alpha_j) = \{j\}$ and $V = Var(\alpha_j)$. Hence, according to Lemma 1, α_j is also executable from $\alpha_i(s_{\text{init}})$. This violates the mutual exclusion condition because both i and j are in their regions C at $(\alpha_i \circ \alpha_j)(s_{\text{init}})$. □

Since a shared variable is local to one process and remote to all other processes, a shared variable accessed in both α_i and α_j is remote to either i or j, or to both. That is, at least one of i and j accesses a remote shared variable.

Next, we prove Lemma 14. Suppose that every process $i \in \mathcal{P}$ accesses at most one remote shared variable in α_i. (Note that i may access many local shared variables in α_i.) Based on the pairwise common lemma, we show that all processes must access one common shared variable, say v, in their respective solo executions. This implies that for all $i \in \mathcal{P}$, except the process to which v is local, i accesses exactly one remote shared variable in α_i and this shared variable is v.

Lemma 14. *Suppose that for all α_i, $i \in \mathcal{P}$, i accesses at most one remote shared variable in α_i. Then there exists exactly one shared variable v such that for all α_i, $i \in \mathcal{P}$, i accesses v in α_i. More precisely, there exists exactly one shared variable v such that $|\mathcal{P}_v| = n$.*

Proof. If there exists one shared variable that is accessed in every α_i, it is easy to show that the number of such shared variables must be exactly one. Suppose not, that is, there is more than one such shared variable. Since $n > 3$ (A is for $n > 3$ processes), there exists one process that accesses more than one remote shared variable, violating the assumption that each process accesses at most one. Thus, all we need to show is that there exists one such shared variable. More precisely, $\exists\, v \in \mathcal{V} : |\mathcal{P}_v| = n$. We first show the following weaker claim.

Claim 14.1. *Suppose that for all α_i, $i \in \mathcal{P}$, i accesses at most one remote shared variable in α_i. Then, $\exists\, v \in \mathcal{V} : |\mathcal{P}_v| > 2$.*

Proof. By way of contradiction, suppose that $|\mathcal{P}_v| \leqslant 2$ for all $v \in \mathcal{V}$. We show that some process accesses more than one remote shared variable. This contradicts the assumption that each process accesses at most one.

Consider four distinct processes i, j, k and l. (Processes i, j, k and l exist because $n > 3$.) By the pairwise common lemma, for α_i and α_j, there exists one shared variable accessed in both α_i and α_j. Let variable w be such a variable, i.e., $\{i, j\} \subseteq \mathcal{P}_w$. Since $|\mathcal{P}_v| \leqslant 2$ for all $v \in \mathcal{V}$, $\mathcal{P}_w = \{i, j\}$. Likewise, we conclude that $\mathcal{P}_x = \{j, k\}$ for some variable x, $\mathcal{P}_y = \{k, l\}$ for some variable y and $\mathcal{P}_z = \{j, l\}$ for some variable z. Since $\mathcal{P}_w = \{i, j\} \neq \mathcal{P}_x = \{j, k\}$, w and x must be two different shared variables. Similarly, we conclude that w, x, y and z are four different shared variables. (See Fig. 7.)

Since a shared variable is local to only one process, variable w is remote to at least one of processes i and j. Assume, without loss of generality, w is remote to j. Since j accesses at most one remote shared variable in α_j and it has accessed w, variable x must be local to j and therefore x is remote to process k. Similarly, y is remote to process l. Hence, we know that j accesses remote shared variable w in α_j and l accesses remote shared variable y in α_l. However, variable z, which is accessed in both α_j and α_l, is remote to at least one of j and l. Thus, at least one of j and l accesses more than one remote shared variable, which is the needed contradiction. □

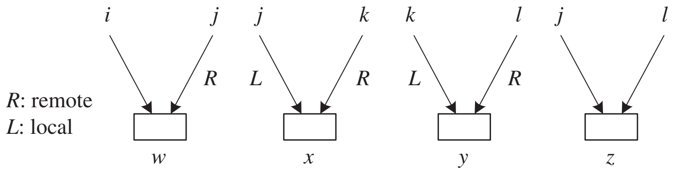

Fig. 7. Shared variables for the proof of Claim 14.1.

Next, we prove that $\exists\, v \in \mathcal{V} : |\mathcal{P}_v| = n$. Again, by way of contradiction, suppose that $|\mathcal{P}_v| < n$ for all $v \in \mathcal{V}$. We will show that some process accesses more than one remote shared variable, which contradicts the assumption that each process accesses at most one. By Claim 14.1, we conclude that there exists one shared variable v such that $n > |\mathcal{P}_v| > 2$. Let variable w be such a shared variable. Since $n > |\mathcal{P}_w| > 2$, assume $\{i, j, k\} \subseteq \mathcal{P}_w$ and $\{l\} \nsubseteq \mathcal{P}_w$. For processes i, j and k, variable w is remote to at least two of them. Without loss of generality, assume w is remote to j and k. Namely, j and k each access remote shared variable w in α_j and α_k.

We now show that some process accesses more than one remote shared variable. Consider α_j and α_l. By the pairwise common lemma, there exists a variable accessed in both α_j and α_l. Let variable x be such a variable, i.e., $\{j, l\} \subseteq \mathcal{P}_x$. Similarly, for α_k and α_l, let variable y be a variable that $\{k, l\} \subseteq \mathcal{P}_y$. Clearly, both x and y are variables different from variable w since $\{l\} \subseteq \mathcal{P}_x$, $\{l\} \subseteq \mathcal{P}_y$, but $\{l\} \nsubseteq \mathcal{P}_w$.

If x and y are the same shared variable, i.e., $\{j, k, l\} \subseteq \mathcal{P}_x = \mathcal{P}_y$, since x is remote to at least one of j and k, at least one of j and k accesses more than one remote shared variable: variables w and x.

Otherwise, if x and y are two different shared variables, since j has accessed remote shared variable w, variable x is local to j and therefore x is remote to l. Thus, for processes k and l, we know that k accesses remote shared variable w in α_k, and l accesses remote shared variable x in α_l. However, because y, which is accessed in both α_k and α_l, is remote to at least one of k and l, at least one of k and l accesses more than one remote shared variable. □

In the following, we complete the proof of the rendezvous property.

Lemma 15 (*rendezvous property*). *Suppose that for all $\alpha_{ij} \in \mathcal{E}$, processes i and j each access at most one remote shared variable in α_{ij}. Then there exists a shared variable v such that for all $\alpha_{ij} \in \mathcal{E}$ that v is remote to both i and j, both i and j must access v in α_{ij}. More precisely,*

$$\exists\, v \in \mathcal{V}, \forall\, \alpha_{ij} \in \mathcal{E}, v \notin \mathcal{V}_i \text{ and } v \notin \mathcal{V}_j : \textit{ both } i \textit{ and } j \textit{ must access } v \textit{ in } \alpha_{ij}.$$

Proof. Since for all $\alpha_{ij} \in \mathcal{E}$, processes i and j each access at most one remote shared variable in α_{ij}, it is true, *a fortiori*, that for all α_i, $i \in \mathcal{P}$, process i accesses at most one remote shared variable in α_i. From Lemma 14, there exists exactly one common shared variable, say v, that is accessed in all α_i, $i \in \mathcal{P}$.

Let m be the process to which v is local. We prove that j also accesses v in every $\alpha_{ij} \in \mathcal{E}$ with $i \neq m$ and $j \neq m$, which completes the proof because v is the needed shared variable.

We first show the following claim.

Claim 15.1. *In every $\alpha_{ij} \in \mathcal{E}$ with $i \neq m$ and $j \neq m$, process j must access some shared variable that has been accessed by i.*

Proof. Suppose not, that is, there exists an α_{ij} with $i \neq m$ and $j \neq m$ in which j does not access any shared variable that has been accessed by i. Let α_{ij} be such an execution. We construct an execution violating the progress condition.

Let α be the subsequence of α_{ij} containing all steps executed by j, that is, the suffix of α_{ij} after α_i ($\alpha_{ij} = \alpha_i \circ \alpha$). We will show that α is executable from s_{init} and $\alpha_{ij}(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha(s_{\text{init}})$. By Lemma 9, and because $\alpha_{ij}(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha(s_{\text{init}})$ and j is locally spinning in its region T at $\alpha_{ij}(s_{\text{init}})$, this implies that j is also locally spinning in its region T at $\alpha(s_{\text{init}})$. But this easily yields a j-execution α' violating the progress condition. Starting from s_{init}, execution α' begins with α. It then continues by running j alone. Since j is locally spinning in its region T at $\alpha(s_{\text{init}})$, no finite j-execution fragment executable from $\alpha(s_{\text{init}})$ will lead j to its region C. This violates the progress condition.

It remains only to show that α is executable from s_{init} and $\alpha_{ij}(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha(s_{\text{init}})$. Since j does not access any shared variable that has been accessed by i, we have $\alpha_i(s_{\text{init}}) \underset{V}{\overset{j}{\sim}} s_{\text{init}}$, where $V = \mathit{Var}(\alpha)$. By the definition of α, $\mathit{Pro}(\alpha) = \{j\}$ and α is executable from $\alpha_i(s_{\text{init}})$. Thus, by Lemma 1, α is also executable from s_{init} and $\alpha_{ij}(s_{\text{init}}) \underset{V}{\overset{j}{\sim}} \alpha(s_{\text{init}})$. In addition, since v is the only remote shared variable accessed by i in α_{ij}, i does not access any shared variable located at j. We now show that $\alpha_{ij}(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha(s_{\text{init}})$ holds. Let w be any variable in $\mathcal{V}_j$. If w is in V, it has the same value at $\alpha_{ij}(s_{\text{init}})$ and $\alpha(s_{\text{init}})$ because we have proved that $\alpha_{ij}(s_{\text{init}}) \underset{V}{\overset{j}{\sim}} \alpha(s_{\text{init}})$. Otherwise, if w is not accessed by j in α, because i does not access any shared variable located at j, the value of w is never changed in α_{ij} and α. Hence, $\alpha_{ij}(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha(s_{\text{init}})$. □

Next, we prove that j also accesses v in every α_{ij} with $i \neq m$ and $j \neq m$ by contradiction. Assume that there exists an α_{ij} with $i \neq m$ and $j \neq m$ in which j does not access v. Let α_{ij} be such an execution.

In α_{ij}, the possible shared variables accessed by i are v and the shared variables located at i. By Claim 15.1, since j does not access v, j must access some shared variable located at i. Since j accesses at most one remote shared variable in α_{ij}, j must access exactly one remote shared variable and this shared variable is located at i.

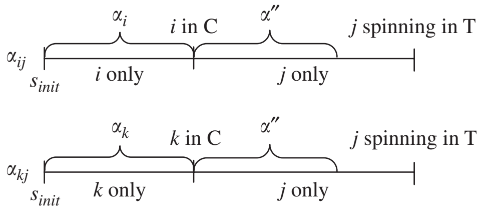

Fig. 8. Executions α_{ij} and α_{kj}. Execution fragment α'' ends with the first RMR step from j to i.

Consider another α_k, $k \neq m, i, j$. We show that in α_{kj}, j does not access any shared variable that has been accessed by k in α_{kj}, contradicting Claim 15.1. As shown in Fig. 8, let α'' be the subsequence of α_{ij} starting from the end of α_i (not including the end of α_i) until j has just finished its first RMR step, which is from j to i. In α_i and α_k, since i and k do not access any shared variable located at process j (variable v, which is located at m, is the only remote shared variable accessed by i and k), we have $\alpha_i(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha_k(s_{\text{init}})$. Since $\alpha_i(s_{\text{init}}) \underset{\mathcal{V}_j}{\overset{j}{\sim}} \alpha_k(s_{\text{init}})$, j enables the same step at $\alpha_i(s_{\text{init}})$ and $\alpha_k(s_{\text{init}})$ by the determinism and the localized enabling assumptions of the model. Furthermore, if the step is not remote, the resulting system states are also indistinguishable to j with respect to $\mathcal{V}_j$ by the localized update assumption. Using such an argument repeatedly, it is easy to see that j also performs α'' in α_{kj} after α_k as it does in α_{ij}. Thus, j also accesses a shared variable located at i in α_{kj}.

Since process j accesses at most one remote shared variable in α_{kj} by the assumption on every execution in $\mathcal{E}$, j accesses exactly one remote shared variable and this shared variable is located at i. Therefore, j does not access any shared variable that has been accessed by k in α_{kj}. (Note that the possible shared variables accessed by k in α_{kj} are v and the shared variables located at k.) This contradicts Claim 15.1. □

The main lemma, rendezvous property, has been proven. To finish the lower bound proof, it remains to provide the details that are skipped in the proof outline in Section 4.2.

Proof (of Theorem 8). We show that there exists an execution of A in which some process performs at least three RMR steps to enter and exit its region C once. We complete the proof with a case analysis on $\mathcal{E}$, getting a goal execution for each possibility.

Case 1: $\exists \alpha_{ij} \in \mathcal{E} : \mathit{time}(i, \alpha_{ij}) \geqslant 2$ or $\mathit{time}(j, \alpha_{ij}) \geqslant 2$. Let $\alpha_{ij} \in \mathcal{E}$ be such an execution.

If $\mathit{time}(i, \alpha_{ij}) \geqslant 2$, we have presented the construction of a goal execution in Section 4.2. If $\mathit{time}(i, \alpha_{ij}) \geqslant 2$ does not hold, it must be the case that $\mathit{time}(j, \alpha_{ij}) \geqslant 2$ holds. It remains to construct a goal execution in this case. We extend α_{ij} to an execution in which j must take at least one RMR step in its region E.

As shown in Fig. 9, we extend α_{ij}, in which j has taken at least two RMR steps, to α_1 by letting i leave its region C first and then alternately executing enabled steps of i and j until i enters its region R and j enters its region C. This follows from the progress condition. Then we extend α_1 to α_2 by running a

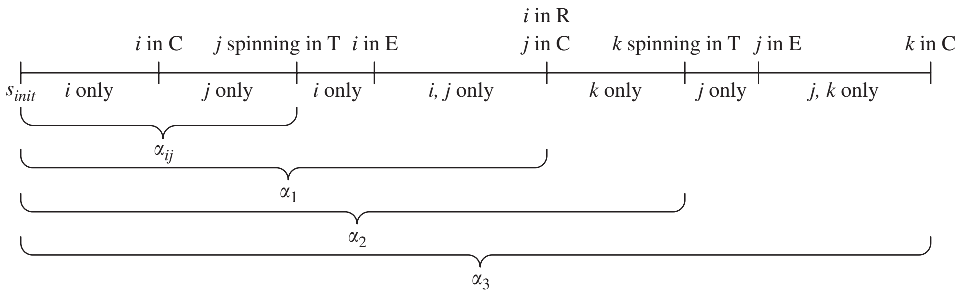

Fig. 9. Executions in Case 1.

new competing process k alone until k is locally spinning in its region T. This follows from Lemma 12. Finally, we extend α_2 to α_3 by letting j leave its region C first and then alternately executing enabled steps of j and k until k enters its region C; along the way, by the inherent cost lemma, process j must take at least one RMR step to k. In total, j takes at least three RMR steps to enter and exit its region C once in α_3.

Case 2: $\forall \alpha_{ij} \in \mathcal{E} : \mathit{time}(i, \alpha_{ij}) < 2$ and $\mathit{time}(j, \alpha_{ij}) < 2$. By the rendezvous property, there exists a shared variable v such that for any distinct processes i and j to which v is remote, both i and j must access v in α_{ij}.

Let m be the process to which v is local. Take any three distinct processes i, j and k that are different from m. As shown in Section 4.2, we first extend α_{ij} to α'_{ij} by letting i leave its region C and then running i and j until j reaches its region C. If process i takes at least two RMR steps in the portion of α'_{ij} after α_{ij}, execution α'_{ij} is already a goal execution. Otherwise, the inherent cost lemma implies that i takes exactly one RMR step, which is from i to j. We now construct a goal execution α'_{ik}. Let α be the subsequence of α'_{ij} starting from the end of α_{ij} (not including the end of α_{ij}) until i has just finished its RMR step, say step e. The precise construction of α'_{ik} is given below.

Execution α'_{ik} begins with α_{ik}, in which i has taken one RMR step. Then it is concatenated by $\alpha|i$, which ends with an RMR step from i to j. It finishes by letting processes i and k alternately execute enabled steps until k enters its region C; along the way, i must take at least one RMR step to k by the inherent cost lemma. In total, i takes at least three RMR steps in α'_{ik}.

It remains to show that it is legitimate in our construction to concatenate α_{ik} by $\alpha|i$. This follows from Corollary 3. To apply the corollary, we need to show the following properties: $\alpha_{ij}(s_{\mathrm{init}}) \underset{\mathcal{V}_i}{\overset{i}{\sim}} \alpha_{ik}(s_{\mathrm{init}})$; α is executable from $\alpha_{ij}(s_{\mathrm{init}})$ and it ends with an RMR step from i; and α contains neither RMR steps from i nor RMR steps to i except the last step. In α_{ij} and α_{ik}, since i performs the same sequence of steps (i.e., α_i), and j and k do not access any shared variable located at i (v, which is located at m, is the only remote shared variable accessed by j and k), we have $\alpha_{ij}(s_{\mathrm{init}}) \underset{\mathcal{V}_i}{\overset{i}{\sim}} \alpha_{ik}(s_{\mathrm{init}})$. By the definition of α, it is executable from $\alpha_{ij}(s_{\mathrm{init}})$ and it ends with an RMR step from i. Since e is the only RMR step from i in the portion of α'_{ij} after α_{ij}, α contains no RMR step from i except the last one. In addition, by Lemma 10, α contains no RMR step from j and, *a fortiori*, α contains no RMR step from j to i. Thus, α, which is an $\{i, j\}$-execution fragment, contains neither RMR steps from i nor RMR steps to i except the last step. Thus, by Corollary 3, $\alpha|i$ is executable from $\alpha_{ik}(s_{\mathrm{init}})$. $\square$

5. Conclusion

5.1. Summary of results

We have proved that the remote reference time complexity of any mutual exclusion algorithm with at least four processes is at least three in DSM systems, and provided an algorithm with the matching upper bound. The bound is therefore tight. The lower bound is proved by constructing an execution in which some process takes at least three RMRs to enter and exit its critical region once. In the course of proving the lower bound, we need to formalize the notion of a process "entering a local-spin loop." Danek and Hadzilacos [14] and we [11] independently proposed a similar formal definition at about the same time. Based on the definition, we also present several properties of local-spin mutual exclusion algorithms.

The tight bound remains unchanged when we consider lockout-freedom and bounded bypass. Because we only assume the basic conditions of the mutual exclusion problem in the proof of the lower bound, this bound also holds for lockout-free mutual exclusion and bounded-bypass mutual exclusion. Additionally, we have shown that the algorithm also satisfies these two fairness properties. Consequently, the time complexity of mutual exclusion in our model is not sensitive to the properties.

5.2. Other open questions

One disadvantage of our algorithm is that it uses two primitives, *compare&swap* and *fetch&store*, besides *read/write*. Since Cypher [13] showed that there is no constant time algorithm using comparison primitives (e.g., *test&set* and *compare&swap*) and *read/write*, a non-comparison primitive is needed to implement an algorithm with the matching upper bound. An open question is whether such an algorithm is obtainable using only one non-comparison primitive such as *fetch&store* in addition to *read/write*.

The algorithm satisfies lockout-freedom and bounded bypass. However, it does not satisfy the first-come-first-served (FCFS) property that if process i performs a step in its region

T before process j, then j does not enter its region C before i. Hence, the tight bound for the FCFS mutual exclusion problem remains to be solved. The tight bound must be either three or four, because the MCS lock [23] satisfies the FCFS property and its time complexity is four, and our lower bound of three also holds for the problem.

In this paper, we focus only on DSM systems. The lower bound proof herein is not applicable to CC systems. A problem left open is what exact lower bounds are obtainable for CC systems.

Acknowledgments

The authors are grateful to the anonymous referees for many helpful comments. This work was supported by National Science Council, Taiwan, under Grant NSC 93-2213-E-009-116.

References

[1] J.H. Anderson, Y.-J. Kim, Nonatomic mutual exclusion with local spinning, in: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing, 2002, pp. 3–12.
[2] J.H. Anderson, Y.-J. Kim, An improved lower bound for the time complexity of mutual exclusion, Distributed Comput. 15 (4) (2002) 221–253.
[3] J.H. Anderson, Y.-J. Kim, T. Herman, Shared-memory mutual exclusion: major research trends since 1986, Distributed Comput. 16 (2–3) (2003) 75–110.
[4] J.H. Anderson, M. Moir, Using local-spin k-exclusion algorithms to improve wait-free object implementations, Distributed Comput. 11 (1) (1997) 1–20.
[5] J.H. Anderson, J.-H. Yang, Time/contention trade-offs for multiprocessor synchronization, Inform. Comput. 124 (1) (1996) 68–84.
[6] T.E. Anderson, The performance of spin lock alternatives for shared-memory multiprocessors, IEEE Trans. Parallel Distributed Systems 1 (1) (1990) 6–16.
[8] J.E. Burns, P. Jackson, N.A. Lynch, M.J. Fischer, G.L. Peterson, Data requirements for implementation of n-process mutual exclusion using a single shared variable, J. ACM 29 (1) (1982) 183–205.
[10] K.M. Chandy, J. Misra, How processes learn, Distributed Comput. 1 (1986) 40–52.
[11] S.-H. Chen, T.-L. Huang, A tight bound on time complexity of mutual exclusion, in: Proceedings of the International Computer Symposium, Taipei, Taiwan, 2004, pp. 1352–1357.
[12] T.S. Craig, Queuing spin lock algorithms to support timing predictability, in: Proceedings of the 14th IEEE Real-Time Systems Symposium, 1993, pp. 148–157.
[13] R. Cypher, The communication requirements of mutual exclusion, in: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures, 1995, pp. 147–156.
[14] R. Danek, V. Hadzilacos, Local-spin group mutual exclusion algorithms, in: Proceedings of the 18th International Symposium on Distributed Computing, 2004.
[15] C. Dwork, M. Herlihy, O. Waarts, Contention in shared memory algorithms, J. ACM 44 (6) (1997) 779–805.
[16] S.S. Fu, N.-F. Tzeng, A circular list-based mutual exclusion scheme for large shared-memory multiprocessors, IEEE Trans. Parallel Distributed Systems 8 (6) (1997) 628–639.
[17] G. Graunke, S. Thakkar, Synchronization algorithms for shared-memory multiprocessors, IEEE Comput. 23 (6) (1990) 60–69.
[18] T.-L. Huang, Fast and fair mutual exclusion for shared memory systems, in: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999, pp. 224–231.
[19] T.-L. Huang, C.-H. Shann, A comment on "A circular list-based mutual exclusion scheme for large shared-memory multiprocessors", IEEE Trans. Parallel Distributed Systems 9 (4) (1998) 414–415.
[20] P. Keane, M. Moir, A simple local-spin group mutual exclusion algorithm, IEEE Trans. Parallel Distributed Systems 12 (7) (2001) 673–685.
[21] Y.-J. Kim, J.H. Anderson, A time complexity bound for adaptive mutual exclusion, in: Proceedings of the 15th International Symposium on Distributed Computing, 2001, pp. 1–15.
[22] N.A. Lynch, Distributed in Algorithms, Morgan Kaufmann, Los Altos, CA, 1996.
[23] J.M. Mellor-Crummey, M.L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM Trans. Comput. Systems 9 (1) (1991) 21–65.
[24] J.-H. Yang, J.H. Anderson, A fast, scalable mutual exclusion algorithm, Distributed Comput. 9 (1) (1995) 51–60.

Sheng-Hsiung Chen received the B.S. degree in Computer Science and Information Engineering from National Chiao Tung University, Taiwan, in 1998, where he is currently pursuing the Ph.D. degree in Computer Science. His main research interests are in distributed algorithms; particularly, in fault tolerance and synchronization.

Ting-Lu Huang studied at Tunghai University (B.S., 1976), University of Texas at Arlington (M.S., 1981), and Northwestern (Ph.D., 1989). He is currently an associate professor in the Department of Computer Science at National Chiao Tung University. He works on distributed algorithms and distributed systems.