A Tight Bound on Time Complexity of Mutual Exclusion

Sheng-Hsiung Chen and Ting-Lu Huang
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.
{chenss,tlhuang}@csie.nctu.edu.tw

Abstract

In distributed shared memory multiprocessors, remote memory accesses 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 accesses. We establish a lower bound of 3 on remote access time complexity for mutual exclusion algorithms in a model where processes communicate by means of a general read-modify-write primitive. Since a general read-modify-write primitive is a generalization of all atomic primitives that access at most one shared variable, our lower bound holds for any set of such primitives. Furthermore, this lower bound is tight because it matches the upper bound of Huang’s algorithm proposed in 1999.

Keywords: mutual exclusion, atomic instructions, shared-memory systems, time complexity, tight bounds

1 Introduction

The mutual exclusion problem is fundamental in asynchronous shared-memory 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.

A mutual exclusion algorithm may produce large amount of processor-to-memory traffic in shared-memory systems, heavily degrading the system performance. Since all processes communicate through the shared memory, each competing process may test certain shared variables repeatedly while it is waiting to enter its critical region. This problem is not inherent in multiprocessor systems in which each processor has a local portion of shared memory (i.e., distributed shared-memory systems (DSM)) or has a local cache (i.e., cache

*This work is supported by National Science Council, Republic of China, under Grant NSC 93-2213-E-009-116.
coherent systems (CC)) [23]. In DSM systems, a memory access step 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 access step causes interconnect traffic depends on the various cache protocols. Generally speaking, the first access (be it matter read, write, or both) to a shared variable will cause interconnect traffic and establish a cached copy. But the subsequent reads will not cause traffic unless the cached copy of the shared variable is invalidated. In general, a memory access step is described as *local* if it doesn’t cause any interconnect traffic; otherwise, it is *remote*. Recent work on the mutual exclusion problem has focused on the design of *local-spin* algorithms that reduce the number of remote memory access (RMA) steps by busy waiting only on locally-accessible shared variables. A number of performance studies [6, 14, 23, 27, 4, 18] have shown that synchronization algorithms minimizing the number of RMA steps have the best performance.

Since the number of RMA steps accurately reflects the performance of an algorithm, Anderson and Yang [5] first defined this number as the time complexity measure. To be more specific, the time complexity of a mutual exclusion algorithm is the maximum number of RMA steps required by one process to go through its critical region once. Based on this definition, a mutual exclusion algorithm is local-spin if its time complexity is bounded [18].

Many local-spin mutual exclusion algorithms have been proposed in the literature. Using some read-modify-write primitives in addition to atomic *read* and *write*, many mutual exclusion algorithms are of constant time complexity. For example, Anderson [6] and Graunke and Thakkar [14] proposed a constant time algorithm for CC systems using *fetch-and-increment* and *fetch-and-store*, respectively. Mellor-Crummey and Scott [23] first proposed two constant time algorithms (referred to as MCS lock in the literature) for both CC and DSM systems, one using *fetch-and-store* and *compare-and-swap* and the other using *fetch-and-store* only. Craig [10], Magnusson et al. [22], and Huang and Lin [16] independently proposed the same constant time algorithm with *fetch-and-store*. Craig presented variants of the algorithm for both CC and DSM systems; while the other two considered only CC systems. In recent work, Anderson and Kim [3] presented a genetic constant time algorithm for both CC and DSM systems, using *fetch-and-ϕ*.

Even though there are several mutual exclusion algorithms of constant time complexity, some other researchers aimed to minimize the number of RMA steps. For instance, Fu and Tzeng [12, 17] improved MCS lock by using circular waiting list to eliminate RMA steps needed in MCS lock to re-direct an address link for each privilege passing during resource busy period. Unfortunately, the algorithm of Fu and Tzeng suffers from blocking in its exit region, the code fragment after executing its critical region. Then, Huang [15] presented
algorithms that follow the line of the algorithm of Fu and Tzeng but eliminate the above drawback.

Although improving algorithms of constant time yields no asymptotic improvement in performance, we consider it worthwhile to reduce the number of RMA steps as many as possible. Mutual exclusion is a basic synchronization facility frequently used in multiprocessor systems both in operating system kernel level and in users’ application level [23]. Thus, minimizing the number of RMA steps yields considerable performance improvement, as shown in the simulation results of Fu and Tzeng [12].

The main focus of this paper is to investigate what is the exact lower bound on time complexity. Huang’s algorithm [15] is of time complexity 3 in DSM systems using fetch-and-store and compare-and-swap. An intriguing question is that whether there is any possible algorithm with fewer time complexity than 3.

Related lower bounds. Anderson and Yang [5] first initiated a series of studies of lower bounds on time complexity. They proved a trade-o between the amounts of contention and time complexity. The amounts of contention of an algorithm is the maximum number of processes that are enabled to access the same shared variable simultaneously. They provide the meaningful lower bound on time complexity only if the amounts of contention is limited.

Cypher [11] was the first to study the lower bound on time complexity with arbitrary contention. He showed that there is no constant time mutual exclusion algorithm that uses only atomic read and write primitives in DSM or in CC systems by proving a lower bound of $\Omega(\log n/\log \log \log n)$ on time complexity, where $n$ is the number of processes. His lower bound holds even if comparison primitives (e.g., test-and-set and compare-and-swap) are available in addition to read and write primitives. In a later work, Anderson and Kim [1] improved Cypher’s lower bound to $\Omega(\log n/\log \log n)$. In addition, Anderson and Kim [2] proved a lower bound of $\Omega(\log n/\log \log n)$ with nonatomic read and write.

The above lower bounds hold only for special sets of primitives. In fact, many atomic primitives other than read, write, and comparison primitives have been implemented in multiprocessor systems such as:

- **fetch-and-add**: BBN Butterfly [25], NYU Ultracomputer [13], IBM RP3 [24];
- **fetch-and-increment**: Cray T3E [26];
- **fetch-and-store**: Cray T3E [26], BBN Butterfly [25], Sequent Symmetry [20];
- **fetch-and-$\phi$**: SGI Origin 2000 [19].
The asymptotic lower bound for such primitives is $\Theta(1)$ because several algorithms of $O(1)$ time complexity have been proposed using read-modify-write primitives such as.fetch-and-store and fetch-and-add. However, the exact lower bound on time complexity is still unknown.

**Contribution.** We prove 3 is a lower bound on time complexity in DSM systems for mutual exclusion algorithms using a general read-modify-write (RMW) primitive. A 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. It is a generalization of most commonly-available primitives which access at most one shared variable in shared memory systems. For instance, a read primitive is a special case of a general RMW primitive such that the submitted function must be the identical function. Hence, our lower bound holds for any set of primitives that involve at most one shared variable atomically. Formally, a general RMW primitive is defined below, where $v$ is the shared variable it involved, and $f$ is any function mapping the value set of $v$ into the same set.

$$\text{RMW (variable } v, \text{ function } f)$$

$$\text{previous} := v$$
$$v := f(v)$$
$$\text{return previous}$$

Our lower bound matches Huang’s algorithm and therefore is tight in DSM systems.

Unlike the proofs of related lower bounds [11, 5, 1, 2], the proof of our bound herein is applicable to DSM systems only. This is because our lower bound is exact rather than asymptotic. Since the previous lower bounds are asymptotic, a constant factor of the number of RMA steps can be ignored. For a exact lower bound, however, each inherent RMA step is critical. But a inherent RMA step in DSM systems may be no longer remote in CC systems, depending on the various cache protocols supported by hardware. To simplify our formal model, we concern only DSM systems in this paper.

The proof strategy of our lower bound is different to those in previous lower bounds [11, 1, 2] focused on restricted sets of primitives. Their proof strategies are similar to each other and described roughly below. In shared-memory systems, communication between two processes is through shared variables, one reading the value written by the other. We can arrange a sequence of accesses to a variable to eliminate possible communication. For instance, if a sequence of read and write are enabled, let all read primitives execute before all write primitives. Then, no communication occurs; furthermore, each value written by a write primitive is overwritten except the last one. For any mutual exclusion algorithm, if
no communication occurs among competing processes, then each competing process must enable another access to a shared variable, otherwise we can construct an execution such that more than one process is in the critical region. Thus, by arranging accesses to shared variables to eliminate possible communication, the previous lower bounds are derived by inductively constructing longer and longer executions.

Since we consider the general RMW primitive, the previous method is not suitable. For instance, if each competing process communicates through a shared variable using \textit{fetch-and-store}, communication will not be eliminated by arranging primitives. Our method is briefly described as follows. Since non-local-spin algorithms may produce unbounded number of RMA steps, we consider only local-spin algorithms without loss of generality. We construct an execution in which some process performs at least 3 RMA steps based on a simple fact of local-spin mutual exclusion algorithms: when a process leaves its critical region, if there is some process busy waiting on one or more local shared variables, the process leaving the critical region must enable at least one RMA step to wake up the busy-waiting process in its exit region.

The rest of the paper is organized as follows. Section 2 provides the system model and definitions. Section 3 presents the lower bound on time complexity. Section 4 is the conclusion.

\section{System model and Definitions}

First, we will describe a model of asynchronous distributed shared memory system. The salient features of our model are that:

1. shared memory is distributed to each process, and

2. processes communicate by means of read-modify-write operations which atomically access one shared variable.

We adopt the definition of a remote memory access step proposed by Anderson and Yang [5], and also define the number of remote memory access steps as the time complexity measurement. Next, we define two indistinguishability relations on system states that will be used in our lower bound proof. Finally, we give a formal definition of mutual exclusion which is similar to the definition in [9].
2.1 Distributed Read-Modify-Write Shared Memory Model

An algorithm in a distributed read-modify-write shared memory system is modelled as a triple \((\mathcal{P}, \mathcal{V}, \delta)\), where \(\mathcal{P}\) is a nonempty finite set of processes, \(\mathcal{V}\) is a nonempty finite set of shared variables, and \(\delta\) 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 nonempty subsets \(\mathcal{V}_i\) for each \(i \in \mathcal{P}\). Intuitively, each shared variable \(v\) is located at a unique process, capturing the essence of a distributed shared memory system. \(\mathcal{V}_i\) denotes the set of all shared variables located at process \(i\). For a process \(i\), a shared variable \(v\) is remote if \(v \not\in \mathcal{V}_i\); otherwise, it is local. In addition, let \(I_v\), a subset of the value set of shared variable \(v\), denote the possible initial values of shared variable \(v\).

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

- \(\Sigma_i\): a (possibly infinite) set of states;
- \(I_i\): a subset of \(\Sigma_i\), indicating the initial states;
- \(\Pi_i : \{(v, f) \mid v \in \mathcal{V} \text{ and } f \text{ is a function mapping from the value set of } v \text{ to the same set}\}\).

Informally, \(\Pi_i\) specifies the steps that \(i\) may execute. Each step \((v, f)_i\) is a read-modify-write operation which atomically reads a value \(old\) from variable \(v\) and writes back \(f(old)\) to the same variable \(v\).

For a step \((v, f)_i \in \Pi_i\), we say that this step accesses the shared variable \(v\). It is a remote memory access (RMA) step if \(v \not\in \mathcal{V}_i\). That is, the step accesses a shared variable located at some other process. An RMA step to \(j\) is an RMA step that accesses a share variable \(v \in \mathcal{V}_j\).

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

The transition relation \(\delta\) 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 \(\delta\) satisfies the following assumptions.

**Localized update:** Suppose \((s, (v, f)_i, s')\) is a transition in \(\delta\), where \((v, f)_i\) is a step of process \(i\).
1. Suppose \((s_1, (v,f)_i, s'_1)\) is an arbitrary transition in \(\delta\), with the same step of \(i\). If \(s(i) = s_1(i)\) and \(s(v) = s_1(v)\), then \(s'(i) = s'_1(i)\).

   Informally, the new state of process \(i\) depends only on the current state of \(i\) and the value of variable \(v\).

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 \(j \in P \setminus \{i\}\), and
   \(s'(u) = s(u)\) for all \(u \in V \setminus \{v\}\).

   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 all system state \(s_1\) with \(s_1(i) = s(i)\), there exists a system state \(s'_1\) such that \((s_1, (v,f)_i, s'_1) \in \delta\).

   We say that a step \(e = (v,f)_i\) is locally enabled in system state \(s\) if there exists a system state \(s'\) such that \((s, e, s') \in \delta\). “Localized enabling” means that whether a step of a process is locally enabled in a system state or not depends only on the process state. If a step of process \(i\) is locally enabled in system state \(s\), then the step is also locally enabled in any other system state \(s_1\) with \(s_1(i) = s(i)\). For brevity, we write “enabled” instead of “locally enabled” throughout this paper.

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

   More precisely, for all \(i \in P\), for any two steps \((v_1, f_1)_i, (v_2, f_2)_i \in \Pi_i\), and for all system state \(s\), if \((v_1, f_1)_i, (v_2, f_2)_i\) are enabled in \(s\), then \((v_1, f_1)_i = (v_2, f_2)_i\).

These three assumptions correspond to normal models of shared memory systems in the literature [8, 21, 7].

If a step \(e = (v,f)_i\) is enabled in system state \(s\), due to the localized update assumption the resulting system state is unique after performing \(e\) in \(s\). (If \((s, (v,f)_i, s')\) and \((s, (v,f)_i, s'')\) are in \(\delta\), we have \(s' = s''\) according to the localized update assumption.) Therefore, we write \(e(s)\) to denote the resulting system state.

An execution fragment is a finite or infinite sequence of steps \(e_1 e_2 \ldots\). Several definitions of execution fragments are given as follows. Let \(\alpha\) and \(\alpha'\) be execution fragments.

- \(|\alpha|\): the length of \(\alpha\).
- \(\alpha[i]\): the subsequence of \(\alpha\) containing all steps of process \(i\) in \(\alpha\).
- \(Pro(\alpha)\): the set of processes that take at least one step in \(\alpha\).
- $\text{Var}(\alpha)$: the set of shared variables accessed in $\alpha$.

- $\alpha \circ \alpha'$: the execution fragment obtained by concatenating $\alpha$ and $\alpha'$, provided that $\alpha$ is finite.

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

An execution fragment $e_1e_2\ldots$ is enabled in a system state $s$ if for all $i \geq 1$, $e_i$ is enabled in $s_{i-1}$ where $s_0 = s$ and $s_i = e_i(s_{i-1})$. If $\alpha$ is a finite execution fragment enabled in $s$, we use $\alpha(s)$ to denote the system state after performing $\alpha$ from $s$. A system state $s'$ is reachable from system state $s$ if there exists a finite execution fragment $\alpha$ such that $\alpha$ is enabled in $s$ and $\alpha(s) = s'$. An execution is an execution fragment that is enabled in an initial system state.

**Indistinguishabilities.** Variants of the notion of indistinguishability are frequently used to prove impossibility results in distributed systems [21]. Here, we define two equivalence relations among system states, and then show several ways to “splice” execution fragments between system states for constructing a needed execution in a proof. Such proof method is inspired by Lynch [21].

First, we introduce the indistinguishability relations, the first relation with respect to an execution fragment, and the second relation with respect to a process.

**Definition 1** Let $s$ and $s'$ be system states and $\alpha$ an execution fragment. System states $s$ and $s'$ are indistinguishable to execution fragment $\alpha$, written as $s \approx s'$, if

1. $s(i) = s'(i)$ for each $i \in \text{Pro}(\alpha)$, and
2. $s(v) = s'(v)$ for each $v \in \text{Var}(\alpha)$.

**Definition 2** Two system states $s$ and $s'$ are locally indistinguishable to process $i$, written as $s \overset{i}{\approx} s'$, if

1. $s(i) = s'(i)$, and
2. $s(v) = s'(v)$ for each $v \in \mathcal{V}_i$.

Informally, we say that two system states $s$ and $s'$ are indistinguishable to execution fragment $\alpha$ if each process and each shared variable involved in $\alpha$ have the same state and the same value in $s$ and $s'$, respectively. We say that $s$ and $s'$ are locally indistinguishable to process
if the state of process \( i \) and the values of all shared variables located at \( i \) are the same in system states \( s \) and \( s' \).

According to these two definitions, if execution fragment \( \alpha \) is an \( i \)-execution fragment and all shared variables accessed in \( \alpha \) are located at \( i \), then \( s \sim i s' \) implies \( s \overset{\alpha}{\sim} s' \) since \( \text{Pro}(\alpha) \subseteq \{i\} \) and \( \text{Var}(\alpha) \subseteq \mathcal{V}_i \). Thus, we have:

**Lemma 1** If \( \alpha \) is an \( i \)-execution fragment such that \( \text{Var}(\alpha) \subseteq \mathcal{V}_i \), then \( s \sim i s' \) implies \( s \overset{\alpha}{\sim} s' \).

Next, we present two lemmas showing ways to splice execution fragment based on the indistinguishability relations defined above. Informally, these lemmas are easily extended from the localized update and localized enabling assumptions of our model. Thus, we leave the proofs of these two lemmas in the appendix.

If execution fragment \( \alpha \) is enabled in system state \( s \) and \( s \overset{\alpha}{\sim} s' \), it is easy to show that \( \alpha \) is also enabled in system state \( s' \) because all involved in \( \alpha \) are the same in \( s \) and \( s' \). Additionally, if \( \alpha \) is finite, the state of each process and the value of each shared variable involved in \( \alpha \) are also the same in the resulting system states.

**Lemma 2** Let \( s \) and \( s' \) be system states. Suppose that \( \alpha \) is an execution fragment enabled in \( s \). If \( s \overset{\alpha}{\sim} s' \), then \( \alpha \) is also enabled in \( s' \). Furthermore, if \( \alpha \) is finite, at the resulting system states \( \alpha(s) \) and \( \alpha(s') \), \( \alpha(s) \overset{\alpha}{\sim} \alpha(s') \).

Lemma 3 is for system states that are locally indistinguishable to a process \( i \). If an execution fragment \( \alpha \) enabled in system state \( s \) contains neither RMA steps from \( i \) nor RMA steps to \( i \), no communication between \( i \) and any other process occurs in \( \alpha \). Thus, \( \alpha|i \) can be enabled in \( s \) and furthermore, in all system states \( s' \) such that \( s \sim i s' \). Similar to Lemma 2, if \( \alpha \) is finite, the resulting system states \( \alpha(s) \) and \( \alpha|\alpha(s')(s') \) are still locally indistinguishable to process \( i \).

**Lemma 3** Let \( s \) and \( s' \) be two system states and \( i \) a process. Suppose that \( \alpha \) is an execution fragment that is enabled in \( s \) and contains neither RMA steps from \( i \) nor RMA steps to \( i \). If \( s \sim i s' \), then \( \alpha|i \) is also enabled in \( s' \). Furthermore, if \( \alpha \) is finite, at the resulting system states \( \alpha(s) \) and \( \alpha(i)|\alpha(s') \), \( \alpha(s) \overset{\alpha}{\sim} \alpha(i)|\alpha(s') \).

The following corollary of Lemma 3 shows that \( \alpha|i \) is also enabled in \( s' \) even if \( \alpha \) ends with an RMA step from \( i \) (\( \alpha \) is finite). Let \( \alpha' \) be the prefix of \( \alpha \), excluding the last step of \( \alpha \). By Lemma 3, \( \alpha'|i \) is also enabled in \( s' \) and the process states of \( i \) are the same in \( \alpha'(s) \) and \( \alpha'(i)|\alpha(s') \). Thus, the RMA step from \( i \) at the end of \( \alpha \) is also enabled in \( \alpha'|\alpha(s')(s') \). Namely, the execution fragment \( \alpha|i \) (\( \alpha|i = \alpha'|i \circ \text{RMA step from } i \)) is also enabled in
s'. However, since the last step from i is an RMA step, the process states of i in α(s) and (α|i)(s') might be different.

**Corollary 4** Let s and s' be two system states and i a process. Suppose that α is a finite execution fragment that is enabled in s, ends with an RMA step from i, and contains neither RMA steps from i nor RMA steps to i except the last one. If s ≈ s', then α|i is also enabled in s'.

### 2.2 Mutual Exclusion Problem

So far, we have described a distributed shared memory model for all algorithms in general. For mutual exclusion algorithms in particular, we need to make some assumptions to capture the desired exclusion behavior of a set of processes.

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

For each process i, Σ_i is partitioned into nonempty disjoint subsets R_i, T_i, C_i and E_i, indicating that process i is in the remainder region, trying region, critical region and exit region, respectively. In each initial system state, we assume that each process is in the remainder region. 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') ∈ δ and s(i) ∈ R_i, then s'(i) ∈ R_i ∪ T_i.
- If (s, (v, f)_i, s') ∈ δ and s(i) ∈ T_i, then s'(i) ∈ T_i ∪ C_i.
- If (s, (v, f)_i, s') ∈ δ and s(i) ∈ C_i, then s'(i) ∈ C_i ∪ E_i.
- If (s, (v, f)_i, s') ∈ δ and s(i) ∈ E_i, then s'(i) ∈ E_i ∪ R_i.

That is, each process obeys a loop of life cycle: remainder region, trying region, critical region and exit region.

For all steps, we assume that a step in the remainder region or critical region never accesses a shared variable that may be accessed by a step in the trying region or exit region. More precisely, for any two transitions (s_1, (v_1, f_1)_i, s'_1) and (s_2, (v_2, f_2)_j, s'_2) in δ, if s_1(i) ∈ R_i ∪ C_i and s_2(j) ∈ T_j ∪ E_j, then v_1 ≠ v_2.

In addition, a mutual exclusion algorithm must meet the conditions below.
**Mutual Exclusion:** There is no reachable system state from an initial system state in which more than one process is in the critical region.

The next condition depends on a fairness assumption for executions. An execution $\alpha$ from initial system state $s$ is *admissible* if for each process $i$ that contains only finite steps in $\alpha$, the state of process $i$ after performing the last step of $i$ belongs to $R_i$. Namely, a process halts in an admissible execution only if it is in its remainder region.

**Progress:** Let $\alpha$ be an admissible execution from an initial system state $s$ and $\alpha_1$ be any finite prefix of $\alpha$. In system state $\alpha_1(s)$,

- if at least one process is in the trying region and no process is in the critical region, then there exists a finite prefix $\alpha_2$ of $\alpha$, $|\alpha_2| > |\alpha_1|$, such that some process enters the critical region in $\alpha_2(s)$;

- if at least one process is in the exit region, then there exists a finite prefix $\alpha_2$ of $\alpha$, $|\alpha_2| > |\alpha_1|$, such that some process enters the remainder region in $\alpha_2(s)$.

**Time Complexity.** The time complexity of a mutual exclusion algorithm is the maximum number of RMA steps required by one process in its trying region and the following exit region to go through the critical region once.

Then, a local-spin mutual exclusion algorithm can be formally defined. A mutual exclusion algorithm is *local-spin* if its time complexity is bounded, that is, a constant $c$ exists so that its time complexity $\leq c$.

**3 Lower Bound on Time Complexity**

In this section, we show that the time complexity of any mutual exclusion algorithm is at least 3.

**Theorem 5** Suppose that an algorithm $A$ solves the mutual exclusion problem for $n > 3$ processes. Then the time complexity of $A$ must be greater than or equal to 3.

For each mutual exclusion algorithm, our objective is to show that there exists an execution such that some process performs at least 3 RMA steps in its trying region and exit region to go through its critical region once.

We first make a simplifying restriction on the mutual exclusion algorithms. Next, we propose several properties of local-spin algorithms. These properties show that starting from certain reachable system states, at least one RMA step must be taken to wake up a
process that is waiting to enter its critical region. We will use these properties to construct a desired execution in our lower bound proof. Finally, before showing the detail proof of the lower bound, we present the outline of our proof.

For simplicity, and without loss of generality, we make the following assumption on the mutual exclusion algorithms. We consider only local-spin mutual exclusion algorithms because the time complexity of a non-local-spin algorithm is unbounded and must be greater than 3.

3.1 Basic Properties of Local-spin Algorithms

Two lemmas of local-spin algorithms are presented. Since these lemmas are somewhat intuitive, we skip the proofs of these lemmas here and leave them in the appendix.

First, we need a definition. Since our model is asynchronous, a process can be in the critical region for arbitrarily long time. Thus, for a local-spin mutual exclusion algorithm, because the time complexity is bounded, a process will not enable RMA steps anymore after some point in the trying region while some other process is in its critical region. We say that the process is locally spinning in its trying region.

**Definition 3** In a system state $s$, a process $i$ in $T$ is locally spinning if for all finite $i$-execution fragment $\alpha$ enabled in $s$, $\alpha$ contains no RMA step and $i$ is still in $T$ at $\alpha(s)$.

Informally, a process $i$ locally spinning in $T$ means that process $i$ is busy waiting at certain local shared variables. For any local-spin mutual exclusion algorithm, we can easily construct an execution such that some competing process is locally spinning in $T$. As the following lemma shows, starting from a state in which some process $i$ is in $C$ and running another process $j$ alone to enter the trying region, there must be a reachable system state such that $j$ is locally spinning in $T$, otherwise the number of RMA steps executed by $j$ is unbounded, violating the local spin condition.

**Lemma 6** Suppose $A$ is a local-spin mutual exclusion algorithm for $n > 1$ processes. Let $s$ be a system state reachable from an initial system state such that process $i$ is in $C$ and process $j$ is in $R$. Then there exists a finite $j$-execution fragment $\alpha$ enabled in $s$ such that $j$ is locally spinning in $T$ at system state $\alpha(s)$.

Intuitively, if process $j$ is locally spinning at some point and enters its critical region at later point, then there exists at least one RMA step by some other process to wake up $j$. As shown in the inherent cost lemma below, if there is no RMA step to $j$, then $j$ will continue to wait in its trying region.
Lemma 7 (inherent cost) Suppose $A$ is a local-spin mutual exclusion algorithm for $n > 1$ processes. Let $s$ be a system state in which process $i$ is in $C$ and process $j$ is locally spinning in $T$. Suppose that process $j$ reaches $C$ in a finite $\{i,j\}$-execution fragment $\alpha$ enabled in $s$. Then, $\alpha$ must contain at least one RMA step from $i$ to $j$.

3.2 Proof Outline

To prove this lower bound, it suffices to show that for any local-spin mutual exclusion algorithm there exists an execution such that some process takes at least 3 remote memory accesses. Suppose $A = (P, V, \delta)$ is a local-spin mutual exclusion algorithm for $n$ processes. We will construct a desired execution of $A$ in which some process takes at least 3 RMA steps in its trying and exit regions. Let $s$ be any initial system state of $A$. To construct a desired execution, we start by defining $n$ solo executions of $A$, one per process, each starting from the initial system state $s$ and involving its steps only until it has just reached its critical region. Then, with a case analysis on the number of RMA steps taken by every process in its solo execution, the proof proceeds by extending a solo execution for each case until the desired lower bound is attained.

For each process $i \in P$, let $\alpha_i$ denote the solo execution of process $i$. The progress condition implies that $\alpha_i$ exists and is finite. Since $A$ is deterministic, $\alpha_i$ is unique.

Consider each solo execution of $A$ from $s$. We get a desired execution for the following two cases.

Case 1. There exists some $\alpha_i$ such that $i$ takes at least 2 RMA steps in its trying region.

Case 2. There exists no $\alpha_i$ such that $i$ takes at least 2 RMA steps in its trying region.

That is, for each solo execution $\alpha_i$, process $i$ takes at most one RMA step in its trying region.

Case 1.

Assume that $A$ is in this case. Let $\alpha_i$ be a solo execution in which process $i$ takes at least 2 RMA steps in its trying region. If we extend $\alpha_i$ to obtain an extension such that $i$ takes at least one RMA step in its exit region, we get a desired execution since $i$ totally takes at least 3 RMA steps. The inherent cost lemma shows that to wake up a process that is locally spinning in $T$, at least one RMA step to the process must be enabled by some other process. Thus, at the end of $\alpha_i$, we let another process $j$ enter its trying region and take its steps only until $j$ is locally spinning in $T$. (Lemma 6 implies that $j$ will eventually locally spin.) Then, let $i$ leave its critical region first and run steps of $i$
and $j$ only until $j$ enters its critical region. In the resulting execution, process $i$ takes at least one RMA step to $j$ in its exit region, according to the inherent cost lemma.

**Case 2.**

Before constructing a desired execution, we introduce a property (Property 1 in the next subsection) among these solo executions if algorithm $A$ is in this case: there is one shared variable, say variable $v$, that is accessed in every $\alpha_i$. Since every process takes at most one RMA step, this property shows that, except one process, say process $m$, at which variable $v$ is located, each process $i$ takes exactly one RMA step in $\alpha_i$ and this step is to access $v$.

We now continue to construct a desired execution. For each $\alpha_i$ and each process $j$ such that $i \neq j$, we extend $\alpha_i$ to $\alpha_{ij}$ by running $j$ only until $j$ has just entered a state in which $j$ is locally spinning. We consider all $\alpha_{ij}$, $i, j \in P$ and $i \neq j$. With a case analysis on the number of RMA steps taken by process $j$ in each $\alpha_{ij}$, we get a desired execution extended from some $\alpha_{ij}$ for each case:

**Case 2.1.** There exists a $\alpha_{ij}$ in which $j$ takes at least 2 RMA steps.

**Case 2.2.** There exists no $\alpha_{ij}$ in which $j$ takes at least 2 RMA steps, i.e., for each $\alpha_{ij}$, process $j$ takes at most one RMA step in $\alpha_{ij}$.

**Case 2.1.**

By a similar way in **Case 1**, we can obtain a desired execution in which $j$ takes at least 1 RMA step to wake up some other process that is locally spinning in $T$, and therefore $j$ totally takes at least 3 RMA steps.

**Case 2.2.**

This case is the heart of the lower bound proof.

In this case, not only $i$ but also $j$ take at most one RMA step in each $\alpha_{ij}$. Except process $m$, we have known that each process $i$ takes exactly one RMA step and this RMA step is to access $v$ in $\alpha_i$. Furthermore, we will show that for each $\alpha_{ij}$ such that $i$ and $j$ are different to $m$, process $j$ also takes exactly one RMA step and this step is also to access $v$ (Property 2 in the next subsection). Now, fix a $\alpha_{ij}$ such that $i$ and $j$ are different to $m$. We know that $i$ and $j$ take exactly one RMA step and this step is to access $v$ in $\alpha_{ij}$, respectively. It follows that communication between $i$ and $j$ in $\alpha_{ij}$ is through shared variable $v$ which is remote for both $i$ and $j$. Hence, starting from system state $\alpha_{ij}(s)$, $i$ does not know that $j$ is locally spinning before executing any RMA steps. Based on this, we show that $i$ will perform at least 2
RMA steps in its exit region, i.e., totally at least 3 RMA steps, in some extension from $\alpha_{ij}$.

Such extension from $\alpha_{ij}$ is easily constructed as follows. We extend $\alpha_{ij}$ by letting process $i$ leave its critical region first and then running processes $i$ and $j$ only until $j$ reaches its critical region. Informally, process $i$ must take at least 2 RMA steps in its exit region. Since process $j$ is locally spinning at the end of $\alpha_{ij}$, process $i$ must take at least one RMA step to wake up process $j$. In addition, since $i$ does not know that $j$ is locally spinning, $i$ must take at least one RMA step to check which process (if has one) it should wake up before waking up $j$. As a result, process $i$ takes at least 2 RMA steps in its exit region. The proof of the lower bound is completed.

### 3.3 Detail Proofs

We begin by giving some definitions and several lemmas that will be used in the lower bound proof.

For convenience, we restate several definitions mentioned in the previous subsection. Let $A = (\mathcal{P}, \mathcal{V}, \delta)$ be a local-spin algorithm for $n$ processes and $s$ an initial system state of $A$. For each process $i \in \mathcal{P}$, let $\alpha_i$ denote the solo execution of process $i$, starting from the initial system $s$ and involving its steps only until $i$ has just reached its critical region (i.e., $i$ is in $C$ at $\alpha_i(s)$, and for every prefix $\alpha'$ of $\alpha_i$ such that $\alpha' \neq \alpha_i$, $i$ is not in $C$ at $\alpha'(s)$). Such $\alpha_i$ exists and is finite according to the progress condition. For each $\alpha_i$ and each process $j$ such that $j \neq i$, we extend $\alpha_i$ to $\alpha_{ij}$ by running $j$ only until $j$ has just entered a state in which $j$ is locally spinning in $T$. Such $\alpha_{ij}$ exists and is finite according to Lemma 6. Since $A$ is deterministic, each $\alpha_i$ and $\alpha_{ij}$ are unique. In addition, we need a definition to present properties for solo executions: for each shared variable $v$, let $\mathcal{P}_v$ denote the subset of $\mathcal{P}$ such that each process in $\mathcal{P}_v$ accesses variable $v$ in its solo execution. More precisely, for each shared variable $v$ in $\mathcal{V}$,

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

**Basic Facts.** We now present four basic facts (Lemma 8–11) of all $\alpha_i$ and $\alpha_{ij}$. As mentioned in the subsection Proof Outline, we need to show the following two properties to prove the main case, Case 2.2, in our lower bound proof.

Property 1. If there exists no $\alpha_i$ such that $i$ takes at least 2 RMA steps in its trying region, i.e., each process $i$ takes at most one RMA step, then there is one shared variable, say variable $v$, that is accessed in all $\alpha_i$ (This is implied by Lemma 9).
Property 2. If in addition there exists no $\alpha_{ij}$ in which $j$ takes at least 2 RMA steps in $\alpha_{ij}$, then for each $\alpha_{ij}$ such that $i$ and $j$ are different to the process at which $v$ is located, $j$ takes exactly one RMA step and this step is also to access $v$ (This is implied by Lemma 11).

Lemma 9 shows that if each process accesses at most one remote shared variable in its solo execution, then there exists one shared variable accessed in every solo execution. Property 1 is a special case of Lemma 9 because a process taking at most one RMA steps accesses at most one remote shared variable. Similarly, Property 2 is implied by Lemma 11.

To prove Lemma 9 and Lemma 11, we need two more lemmas: Lemma 8 is a property for any two solo executions, and is used in the proof of Lemma 9; while, Lemma 10 is a property for each $\alpha_{ij}$ and is used in the proof of Lemma 11. Note that although Lemma 8 and Lemma 10 hold for any $A$ with $n > 1$ processes, to prove Lemma 9 and Lemma 11 we need to assume that $A$ is with $n > 3$ processes. Thus, our lower bound holds for $A$ with $n > 3$ processes.

First, Lemma 8, the pairwise common lemma, shows that for any two different solo executions $\alpha_i$ and $\alpha_j$, there exists at least one shared variable that is accessed in both $\alpha_i$ and $\alpha_j$. Although $\alpha_i$ and $\alpha_j$ are two independent executions, processes $i$ and $j$ should access at least one common shared variable, respectively, for purposes of synchronization, otherwise we can easily construct an execution such that both $i$ and $j$ are in $C$ simultaneously by concatenating $\alpha_i$ and $\alpha_j$.

**Lemma 8 (pairwise common)** Suppose $A$ is for $n > 1$ processes. For any two solo executions $\alpha_i$ and $\alpha_j$, $i \neq j$, there exists one shared variable accessed in both $\alpha_i$ and $\alpha_j$. 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 variables accessed in both $\alpha_i$ and $\alpha_j$, i.e., $i$ and $j$ access disjoint shared variables in $\alpha_i$ and $\alpha_j$, respectively. Thus, each shared variable accessed in $\alpha_j$ has the same value in system state $s$ and $\alpha_i(s)$. In addition, process $j$ has the same state in $s$ and $\alpha_i(s)$, and therefore $s \approx \alpha_i(s)$. Hence, according to Lemma 2, $\alpha_j$ is also enabled in $\alpha_i(s)$. This violates the mutual exclusion condition, because both $i$ and $j$ are in $C$ at the end of $\alpha_i \circ \alpha_j$ starting from $s$. \qed

Since a shared variable is local for one process and remote for all other processes, a shared variable accessed in both $\alpha_i$ and $\alpha_j$ is remote for at least one of processes $i$ and $j$. That is, at least one of $i$ and $j$ accesses a remote shared variable.

In addition, suppose that $n > 3$ and each process accesses at most one remote shared variable. The pairwise common lemma can be generalized for all solo executions as shown
in Lemma 9: there exists one shared variable accessed in every solo execution \( \alpha_i, i \in P \). We can easily conclude that the number of such shared variables must be one: if there are more than one such shared variable, since \( n > 3 \), there exists one process that accesses more than one remote shared variable, violating the assumption that each process accesses at most one remote shared variable. Let variable \( v \) be the shared variable accessed in every solo execution. Since each process accesses at most one remote shared variable, we make sure that each process, except the process at which \( v \) is located, accesses exactly one shared variable and this shared variable is \( v \) in its solo execution.

The basic idea of the proof of Lemma 9 is described as follows. By the pairwise common lemma, we have known that for any two different solo executions, there exists one shared variable accessed in both solo executions. If each process \( i \) can access at most one remote shared variable in \( \alpha_i \) and \( n > 3 \), to satisfy the pairwise lemma for any two solo executions it must be the case that each process accesses one common shared variable, otherwise some process will access more than one remote shared variable as shown in the following proof.

**Lemma 9** Suppose \( A \) is for \( n > 3 \) processes and each process \( i \) accesses at most one remote shared variable in \( \alpha_i \). Then there exists one shared variable that is accessed in every \( \alpha_i \), \( i \in P \). More precisely, \( \exists v \in V : |P_v| = n \).

**Proof.** We first show the following weaker claim, and then prove this lemma.

**Claim.** Suppose \( A \) is for \( n > 3 \) processes and each process \( i \) accesses at most one remote shared variable in \( \alpha_i \). Then, \( \exists v \in V : |P_v| > 2 \).

**Proof.** By way of contradiction, suppose that \( |P_v| \leq 2 \) for each shared variable \( v \). We show that some process accesses more than one remote shared variable, contradicting the assumption that each process accesses at most one.

Consider processes \( i, j, k \) and \( l \). (Processes \( i, j, k \) and \( l \) exists because \( n > 3 \).) For processes \( i \) and \( j \), there exists one shared variable accessed in both \( \alpha_i \) and \( \alpha_j \) by the pairwise common lemma. Let variable \( u \) be a such variable, i.e., \( \{i, j\} \subseteq P_u \). Since \( |P_v| \leq 2 \) for each shared variable \( v \), \( P_u = \{i, j\} \). Similarly, let variables \( v, w \) and \( x \) be shared variables such that \( P_v = \{j, k\} \), \( P_w = \{k, l\} \) and \( P_x = \{j, l\} \), respectively. (See Figure 1.) Clearly, \( u, v, w \) and \( x \) are four different shared variables. We show that \( j \) or \( l \) accesses more than one remote shared variable as follows.

Since a shared variable is local for only one process, variable \( u \) is remote for at least one of processes \( i \) and \( j \). Assume, without loss of generality, \( u \) is remote for \( j \). Since \( j \) accesses at most one remote shared variable and it has accessed \( u \), variable \( v \) must be local for \( j \), and therefore \( v \) is remote for process \( k \). Similarly, \( w \) is remote for process
Then, we prove this lemma as follows. Again, by way of contradiction, suppose that $|\mathcal{P}_v| < n$ for each shared variable $v$. We also show that some process accesses more than one remote shared variable. By the claim above, we conclude that there exists one shared variable $v$ such that $n > |\mathcal{P}_v| > 2$. Fix this variable $v$. Since $n > |\mathcal{P}_v| > 2$, assume $\{i, j, k\} \subseteq \mathcal{P}_v$ and $\{l\} \not\subseteq \mathcal{P}_v$. For processes $i, j$ and $k$, variable $v$ is remote for at least two of them. Without loss of generality, assume $v$ is remote for $j$ and $k$.

Now we show that some process accesses more than one remote shared variable. Consider processes $j$ and $l$. By the pairwise common lemma, there exists a variable accessed in both $\alpha_j$ and $\alpha_l$. Let variable $w$ be a such variable, i.e., $\{j, l\} \subseteq \mathcal{P}_w$. Similarly, for processes $k$ and $l$, let variable $x$ be a variable such that $\{k, l\} \subseteq \mathcal{P}_x$. Clearly, both $w$ and $x$ are variables different from variable $v$ since $\{l\} \subseteq \mathcal{P}_w$, $\{l\} \subseteq \mathcal{P}_x$, but $\{l\} \not\subseteq \mathcal{P}_v$.

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

Otherwise—$w$ and $x$ are two different shared variables, since $j$ has accessed remote shared variable $v$, variable $w$ is local for $j$ and therefore $w$ is remote for $l$. Notice that we have known that $k$ accesses remote shared variable $v$, and $l$ accesses remote shared variable $w$. However, because $x$ is remote for at least one of $k$ and $l$, at least one of $k$ and $l$ accesses more than one shared variable. \qed

Next, Lemma 10 shows that for each $\alpha_{ij}$, $j$ must access some shared variable that has been accessed by $i$. Informally, process $j$ must read some information left by process $i$ which is in $C$ before starting to wait.

**Lemma 10** Suppose $A$ is for $n > 1$ processes. In each $\alpha_{ij}$, there exists one shared variable accessed by both $i$ and $j$.

**Proof.** By way of contradiction, suppose that processes $i$ and $j$ access disjoint shared
variables in $\alpha_{ij}$. Let $\alpha$ be the subsequence of steps executed by $j$, that is, the suffix of $\alpha_{ij}$ starting from the end of $\alpha_i$. Since processes $i$ and $j$ don’t access any common shared variables, $s \approx \alpha_i(s)$. According to Lemma 2, $\alpha$ is also enabled in $s$ and $j$ is also locally spinning in $T$ at $\alpha(s)$.

But this easily yields a $j$-execution $\alpha'$ violating the progress condition. Starting from $s$, execution $\alpha'$ begins with $\alpha$ and then continues to take steps of $j$ only. Since $j$ is locally spinning in $T$ at $\alpha(s)$, no finite $j$-execution fragment enabled in $\alpha(s)$ will lead $j$ to a system state in which $j$ is in $C$. This violates the progress condition. □

Follow the same assumption in Lemma 9: each process $i$ accesses at most one remote shared variable in $\alpha_i$. Lemma 9 implies that there is exactly one shared variable that is accessed in every $\alpha_i$. Assume that the shared variable, say $v$, is located at $m$. Moreover, if in all $\alpha_{ij}$, $i \neq m$ and $j \neq m$, $j$ also accesses at most one remote shared variable, the next lemma shows that $j$ accesses exactly one remote shared variable and this variable is also $v$. Consequently, for a $\alpha_{ij}$ such that $i \neq m$ and $j \neq m$, we know that both $i$ and $j$ access the same remote shared variable.

**Lemma 11** Suppose $A$ is for $n > 3$ processes and each process $i$ accesses at most one remote shared variable in $\alpha_i$. By Lemma 9, there is exactly one shared variable, say $v$, that is accessed in every $\alpha_i$. Assume that $v$ is located at process $m$. In addition, suppose that in each $\alpha_{ij}$, $i \neq m$ and $j \neq m$, $j$ also accesses at most one remote shared variable. Then, in each $\alpha_{ij}$, $i \neq m$ and $j \neq m$, $j$ accesses exactly one remote shared variable and this variable is $v$.

**Proof.** By way of contradiction, there exists a $\alpha_{ij}$, $i \neq m$ and $j \neq m$, such that $j$ does not access the remote shared variable $v$. Fix this $\alpha_{ij}$. We construct an execution that violates Lemma 10 as follows.

Lemma 10 implies that $i$ and $j$ must access some common shared variable. The possible shared variables accessed by $i$ in $\alpha_{ij}$ are $v$ and the shared variables located at $i$. 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 $\alpha_{ij}$, $j$ must access exactly one remote shared variable and this shared variable is located at $i$. Let $\alpha$ be the execution fragment in $\alpha_{ij}$ starting from the end of $\alpha_i$ (not include the end of $\alpha_i$) until $j$ just has finished its first RMA step. This RMA step accesses a shared variable located at $i$. Note that there is no RMA step in $\alpha$ except the last one. We now consider another $\alpha_k$ such that $k \neq m, i, j$ ($\alpha_k$ exists since $n > 3$). Since processes $i$ and $k$ don’t access any shared variables located at $j$ in $\alpha_i$ and $\alpha_k$, respectively, we have $\alpha_i(s) \not\approx \alpha_k(s)$. We can extend $\alpha_k$ by concatenating $\alpha|j (= \alpha)$ first (by Corollary 4) and
then running \( j \) alone until \( j \) just has entered a system state in which it is locally spinning in \( T \). Since the algorithm is deterministic, the resulting execution is \( \alpha_{kj} \).

By the assumption, process \( j \) accesses at most one remote shared variable in \( \alpha_{kj} \). Hence, \( j \) accesses exactly one remote shared variable and this shared variable is located at \( i \), and therefore \( k \) and \( j \) don’t access any common shared variable in \( \alpha_{kj} \). This violates Lemma 10. \( \square \)

Now Theorem 5 follows:

**Proof (of Theorem 5).** We show that for any local-spin mutual exclusion algorithm there exists an execution such that some process performs at least 3 RMA steps. We complete the proof with a nested case analysis, getting a desired execution for each possibility.

Consider each solo execution \( \alpha_i, i \in \mathcal{P}, \) of \( A \) from \( s \).

**Case 1.** There exists some \( \alpha_i \) in which process \( i \) takes at least 2 RMA steps in its trying region.

Fix this \( \alpha_i \). We will construct an extension from \( \alpha_i \) such that \( i \) must take at least one RMA step in its exit region. Namely, process \( i \) totally takes at least 3 RMA steps in this
execution.

Starting with a $\alpha_{ij}$, we first let process $i$ leave its critical region. Then, continue to execute enabled steps of $i$ and $j$ alternatively until $j$ enters its critical region. Since enabled steps of $i$ and $j$ are executed alternatively, the resulting execution is fair to $i$ and $j$. Thus, process $j$ will eventually enter its critical region by the progress condition. Since $i$ is in $C$ and $j$ is locally spinning in $T$ at the end of $\alpha_{ij}$, process $i$ must take at least one RMA step to $j$ in its exit region according to the inherent cost lemma. Consequently, $i$ takes at least 3 RMA step in the resulting execution.

See Figure 2(a).

**Case 2.** There exists no $\alpha_i$ in which $i$ takes at least 2 RMA steps in its trying region, i.e., for each $\alpha_i$, $i$ takes at most one RMA step.

Now consider all $\alpha_{ij}$. Note that a $\alpha_{ij}$ is extended from $\alpha_i$, that is, $\alpha_i$ is a prefix of $\alpha_{ij}$.

**Case 2.1.** There exists a $\alpha_{ij}$ in which $j$ takes at least 2 RMA steps.

Fix this $\alpha_{ij}$. Similar to **Case 1**, we will construct an extension from $\alpha_{ij}$ such that $j$ must take at least one RMA step in its exit region.

We extend $\alpha_{ij}$ to $\alpha_1$ by letting $i$ leave its critical region first and then alternatively executing enabled steps of $i$ and $j$ only until $i$ is in $R$ and $j$ is in $C$. This follows from the progress condition. Then we extend $\alpha_1$ to $\alpha_2$ by running a new competing process $k$ only until $k$ is locally spinning in $T$. This follows from Lemma 6. Finally, we extend $\alpha_2$ to $\alpha_3$ by letting $j$ leave its critical region first and then alternatively executing enabled steps of $j$ and $k$ only until $k$ is in $C$. By the inherent cost lemma, process $j$ must take at least one RMA step to $k$ in its exit region.

See Figure 2(b).

**Case 2.2.** There exists no $\alpha_{ij}$ in which $j$ takes at least 2 RMA steps, i.e., for each $\alpha_{ij}$, process $j$ takes at most one RMA step in $\alpha_{ij}$.

In this case, not only $i$ but also $j$ take at most one RMA step in each $\alpha_{ij}$. Since each process $i$ takes at most one RMA step in $\alpha_i$ and a step accesses one shared variable in our model, $i$ accesses at most one remote shared variable in $\alpha_i$. Thus, Lemma 9 implies that there is exactly one shared variable, say $v$, that is accessed in every $\alpha_i$. Without loss of generality, suppose $v$ is located at process $m$. For each process $i \neq m$, because $i$ must access $v$ in $\alpha_i$, $i$ takes at least one RMA step to $m$ in $\alpha_i$. Since $i$ takes at most one RMA step in $\alpha_i$, $i$ takes exactly one RMA step and this step is to access $v$ in $\alpha_i$. 

21
Furthermore, for each $\alpha_{ij}$, process $j$ accesses at most more remote shared variable. Obviously, for each $\alpha_{ij}$ such that $i \neq m$ and $j \neq m$, process $j$ accesses at most one remote shared variable. Thus, Lemma 11 implies that in each $\alpha_{ij}$ such that $i \neq m$ and $j \neq m$, $j$ must also access $v$. Similarly, $j$ takes exactly one RMA step and this step is to access $v$ in $\alpha_{ij}$.

In summary, we have known that processes $i$ and $j$ take exactly one RMA step in $\alpha_{ij}$, respectively, provided $i \neq m$ and $j \neq m$. If we can construct an extension from such $\alpha_{ij}$ so that $i$ or $j$ takes additional 2 RMA steps, the proof is done. We show that there exists one desired extension in which $i$ takes at least 2 RMA steps in its exit region.

We extend a $\alpha_{ij}$, $i \neq m$ and $j \neq m$, to $\alpha'_{ij}$ by letting $i$ leave its critical region first and then alternatively executing enabled steps of $i$ and $j$ only until $i$ is in $R$ and $j$ is in $C$. Let $\alpha$ be the execution fragment in $\alpha'_{ij}$ starting from the end of $\alpha_{ij}$ (not include the end of $\alpha_{ij}$) until $i$ just has finished its first RMA step, say step $e$. (At least one RMA step of $i$ exists due to the inherent cost lemma.) Consider (i) $e$ is an RMA step from $i$ to some process other than $j$, and (ii) $e$ is an RMA step from $i$ to $j$. For either case, we show a desired execution as follows.

Case (i) is easy. Execution $\alpha'_{ij}$ is just a desired execution since by the inherent cost lemma there exists at least one another RMA step from $i$ to $j$.

For case (ii), consider another process $k$ such that $k$ is different to $i$, $j$, and $m$. (Process $k$ exists since $n > 3$.) We construct a desired execution extended from $\alpha_{ik}$. At system states $\alpha_{ij}(s)$ and $\alpha_{ik}(s)$, because processes $j$ and $k$ execute exactly one RMA step, respectively, and these RMA steps access the shared variable $v$ located at process $m$, we have $\alpha_{ij}(s) \prec \alpha_{ik}(s)$. Since $j$ is locally spinning at $\alpha_{ij}(s)$, no RMA step occurs in $\alpha$ except the last step of $\alpha$. Thus, we can extend $\alpha_{ik}$ to $\alpha'_{ik}$ by concatenating $\alpha|i$ first (by Corollary 4) and then alternatively executing enabled steps of $i$ and $k$ until $k$ is in $C$. By the inherent cost lemma, there is at least one another RMA step from $i$ to $k$. Totally, $i$ takes at least 2 RMA steps in its exit region (one from $i$ to $j$ and one from $i$ to $k$). See Figure 2(c).

\[\Box\]

4 Conclusion

We have shown that the remote access time complexity of any mutual exclusion algorithm is at least 3 in distributed shared memory systems. In the course of proving the lower
bound, we need to formalize the notion of a process “entering a local-spin loop.” As a minor contribution, the notion is given a definition in a formal model for the first time.

Our lower bound holds for any set of atomic primitives that access at most one shared variable. Essentially, we showed that, for any mutual exclusion algorithm, there exists an execution of the algorithm such that at least one process takes at least 3 RMA steps to go through its critical region once. As a byproduct of the execution construction in the lower bound proof, we found that even if the atomic primitive being considered accesses more than one local shared variable (but at most one remote variable) the lower bound result still holds.

Our result improves the tight bound of mutual exclusion algorithms on time complexity from $\Theta(1)$ to 3. From the theoretical point of view, it may not be so surprising. But, our 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 remote memory accesses [23, 10, 12, 17, 15]. Our tight bound shows that it is impossible to obtain better algorithms than Huang’s [15] in terms of minimizing the number.

References


**Appendix**

The proofs of Lemmas 2, 3, 6 and 7 are presented in the following.

**Lemma 2** Let $s$ and $s'$ be system states. Suppose that $\alpha$ is an execution fragment enabled in $s$. If $s \preceq s'$, then $\alpha$ is also enabled in $s'$. Furthermore, if $\alpha$ is finite, at the resulting system states $\alpha(s)$ and $\alpha(s')$, $\alpha(s) \preceq \alpha(s')$.  

25
Proof. Assume that $s \approx s'$, that is, each process and each shared variable involved in $\alpha$ have the same state and the same value in $s$ and $s'$, respectively. According to the localized update and localized enabling assumptions, a straightforward induction proves that, for each prefix $\alpha'$ of $\alpha$, $\alpha'$ is also enabled in $s'$, and furthermore at the resulting system states $\alpha'(s)$ and $\alpha'(s')$, the states of all processes in $\text{Pro}(\alpha)$ and the values of all shared variables in $\text{Var}(\alpha)$ are the same. \hfill $\square$

Lemma 3 Let $s$ and $s'$ be two system states and $i$ a process. Suppose that $\alpha$ is an execution fragment that is enabled in $s$ and contains neither RMA steps from $i$ nor RMA steps to $i$. If $s \stackrel{i}{\sim} s'$, then $\alpha|i$ is also enabled in $s'$. Furthermore, if $\alpha$ is finite, at the resulting system states $\alpha(s)$ and $(\alpha|i)(s')$, $\alpha(s) \stackrel{i}{\sim} (\alpha|i)(s')$.

Proof. Since $\alpha$ contains no RMA steps from $i$ and no RMA steps to $i$, $i$ does not access any remote shared variables and no other process accesses any shared variable located at $i$ in $\alpha$. That is, while executing $\alpha$ from $s$, the state of $i$ and the values of all shared variables located at $i$ depend only on $\alpha|i$. Thus, $\alpha|i$ is also enabled in $s$. In addition, if $\alpha$ is finite, at system states $\alpha(s)$ and $(\alpha|i)(s)$, the state of process $i$ and the values of all shared variables in $\mathcal{V}_i$ are the same, i.e., $\alpha(s) \stackrel{i}{\sim} (\alpha|i)(s)$.

Then, we show that $\alpha|i$ is also enabled in $s'$. Since $\alpha|i$ is an $i$-execution fragment and $i$ does not access any remote shared variable in $\alpha|i$, $s \stackrel{i}{\sim} s'$ implies $s \stackrel{\alpha|i}{\approx} s'$ according to Lemma 1. Hence, by Lemma 2, $\alpha|i$ is also enabled in $s'$; and if $\alpha$ is finite, at system states $(\alpha|i)(s)$ and $(\alpha|i)(s')$, the state of $i$ and the values of all shared variables in $\mathcal{V}_i$ are the same, i.e., $(\alpha|i)(s) \stackrel{i}{\sim} (\alpha|i)(s')$. If $\alpha$ is finite, since $\alpha(s) \stackrel{i}{\sim} (\alpha|i)(s)$ and $(\alpha|i)(s) \stackrel{i}{\sim} (\alpha|i)(s')$, we have $\alpha(s) \stackrel{i}{\sim} (\alpha|i)(s')$. \hfill $\square$.

Lemma 6 Suppose $A$ is a local-spin mutual exclusion algorithm for $n > 1$ processes. Let $s$ be a system state reachable from an initial system state such that process $i$ is in $C$ and process $j$ is in $R$. Then there exists a finite $j$-execution fragment $\alpha$ enabled in $s$ such that $j$ is locally spinning in $T$ at system state $\alpha(s)$.

Proof. First, let process $j$ enter its trying region from system state $s$. Let $s'$ be the system state after $j$ enters its trying region. Then, we show that there exists a reachable system state from $s'$ by running $j$ only such that $j$ is locally spinning in $T$. Since $i$ is in $C$ at $s'$, $j$ is still in $T$ at any reachable system state from $s'$ by executing steps of $j$ only. Hence, all we need to prove is that $j$ will reach a system state $s''$ such that any finite $j$-execution fragment enabled in $s''$ contains no RMA steps. By way of contradiction, assume that for each reachable system state $s''$ from $s'$ by running $j$ alone, there exists a finite $j$-execution
fragment from $s''$ containing at least one RMA step. Since $A$ is a local-spin algorithm, a constant $c$ exists such that the time complexity of $A$ is less than or equal to $c$. We construct an execution fragment from $s'$ that violates the time bound $c$ as follows.

By the assumption, there is a $j$-execution fragment $\alpha_1$ from $s'$ containing at least one RMA step. By the assumption again, there is a $j$-execution fragment enabled in $\alpha_1(s')$ containing at least one RMA step. We extend $\alpha_1$ to $\alpha_2$ by concatenating this $j$-execution fragment at the end of $\alpha_1$. Repeated use of the assumption, we construct a $j$-execution fragment $\alpha_{c+1}$ containing at least $c+1$ RMA steps. This contradicts the time bound $c$. □

**Lemma 7 (inherent cost)** Suppose $A$ is a local-spin mutual exclusion algorithm for $n > 1$ processes. Let $s$ be a system state in which process $i$ is in $C$ and process $j$ is locally spinning in $T$. Suppose that process $j$ reaches $C$ in a finite $\{i, j\}$-execution fragment $\alpha$ enabled in $s$. Then, $\alpha$ must contain at least one RMA step from $i$ to $j$.

**Proof.** By way of contradiction, suppose that $\alpha$ contains no RMA steps from $i$ to $j$. If we show that $\alpha$ also contains no RMA steps from $j$, the $\{i, j\}$-execution fragment $\alpha$ contains neither RMA steps from $j$ nor RMA steps to $j$. Therefore $\alpha|j$ is also enabled in $s$ and process $j$ has the same state at $\alpha(s)$ and $(\alpha|j)(s)$ by Lemma 3. Since $j$ is in $C$ at $\alpha(s)$, $j$ is also in $C$ at $(\alpha|j)(s)$. Since $i$ is in $C$ at $s$ and takes no steps in $\alpha|j$, $i$ is still in $C$ at $(\alpha|j)(s)$. This violates the mutual exclusion condition, because both $i$ and $j$ are in $C$ at $(\alpha|j)(s)$.

Now, we show that $\alpha$ also contains no RMA steps from $j$. Suppose for the sake of contradiction that $\alpha$ contains at least one RMA step from $j$. Let $\alpha'$ be the prefix of $\alpha$ ending with the first RMA step from $j$. Since $\alpha$ contains no RMA steps to $j$, $\alpha'$ contains no RMA steps to $j$. Thus, $\alpha'$ contains neither RMA steps to $j$ nor RMA steps from $j$ except the last step. By Corollary 4, $\alpha'|j$ is also enabled in $s$. But $j$ is locally spinning in $T$ at $s$, that is, no finite $j$-execution fragment enabled in $s$ will contain an RMA step from $j$. This is a contradiction. □