A Tight Bound on Time Complexity of Mutual Exclusion*

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

Abstract

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 coherent systems (CC)) [14]. 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 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 [14, 15, 1, 11] 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 [2] 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 [11].

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, Mellor-Crummey and Scott [14] 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 [6], Magnusson et al. [13], and Huang and Lin [9] 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.

*This work is supported by National Science Council, Republic of China, under Grant NSC 93-2213-E-009-116.

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 [7, 10] 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 [8] 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 [14]. Thus, minimizing the number of RMA steps yields considerable performance improvement, as shown in the simulation results of Fu and Tzeng [7].

The main focus of this paper is to investigate what is the exact lower bound on time complexity. Huang's algorithm [8] 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.

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.

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

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

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.

2 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 [2], and also define the number of remote memory access steps as the time complexity measurement. Next, we give a formal definition of mutual exclusion which is similar to the definition in [5].

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 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 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 $v \notin \mathcal{V}_i$; 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:

- Σ_i: a (possibly infinite) set of states;
- I_i: a subset of Σ_i, indicating the initial states;
- $\Pi_i : \{(v, f)_i \mid v \in \mathcal{V}$ 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 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 $v \notin \mathcal{V}_i$. 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 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 $(s_1, (v, f)_i, s'_1)$ is an arbitrary transition in δ, 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 $j \in \mathcal{P}\backslash\{i\}$, 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 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 $i \in \mathcal{P}$, 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 [4, 12, 3].

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 δ, 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 $e_1 e_2 \ldots$. Execution fragment α is a P-execution fragment if all processes involved in α are included in 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 $e_1 e_2 \ldots$ 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 α is a finite execution fragment enabled in s, we use $\alpha(s)$ to denote the system state after performing α from s. A system state s' is *reachable* from system state s if there exists a finite execution fragment α such that α is enabled in s and $\alpha(s) = s'$. An *execution* is an execution fragment that is enabled in an initial system state.

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. We assume that 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 $s_1(i) \in R_i \cup C_i$ and $s_2(j) \in T_j \cup E_j$, then $v_1 \neq v_2$.

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 α from initial system state s is *admissible* if for each process i that

contains only finite steps in α, 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 α be an admissible execution from an initial system state s and α_1 be any finite prefix of α. 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 α_2 of α, $|\alpha_2| > |\alpha_1|$, 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 α_2 of α, $|\alpha_2| > |\alpha_1|$, 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 1 *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, we present the outline of our lower bound proof. Due to the space limitation, the detail proof of the lower bound is given in the full version of this paper.[1]

[1]The full version of this paper can be found at http://www.csie.nctu.edu.tw/∼chenss/papers/ICS2004Full.pdf.

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 full paper.

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 1 *In a system state s, a process i in T is locally spinning if for all finite i-execution fragment α enabled in s, α 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 2 *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 α 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 3 (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 α enabled in s. Then, α 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 $A = (\mathcal{P}, \mathcal{V}, \delta)$ 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 $i \in \mathcal{P}$, let α_i denote the solo execution of process i. The progress condition implies that α_i exists and is finite. Since A is deterministic, α_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 α_i such that i takes at least 2 RMA steps in its trying region.

Case 2. There exists no α_i such that i takes at least 2 RMA steps in its trying region. That is, for each solo execution α_i, process i takes at most one RMA step in its trying region.

Case 1.

Assume that A is in this case. Let α_i be a solo execution in which process i takes at least 2 RMA steps in its trying region. If we extend α_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 α_i, we let another process j enter its trying region and take its steps only until j is locally spinning in T. (Lemma 2 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 Section 3.3 of the full paper) among these solo executions if A is in this case: there is one shared variable, say variable v, that is accessed in every α_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 α_i and this step is to access v.

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

Case 2.1. There exists a α_{ij} in which j takes at least 2 RMA steps.

Case 2.2. There exists no α_{ij} in which j takes at least 2 RMA steps, i.e., for each α_{ij}, process j takes at most one RMA step in α_{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 α_{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 α_i. Furthermore, we will show that for each α_{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 Section 3.3 of the full paper). Now, fix a α_{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 α_{ij}, respectively. It follows that communication between i and j in α_{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 α_{ij}.

Such extension from α_{ij} is easily constructed as follows. We extend α_{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 α_{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.

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 [14, 6, 7, 10, 8]. Our tight bound shows that it is impossible to obtain better algorithms than Huang's [8] in terms of minimizing the number.

References

[1] J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:1–20, 1997.

[2] J. H. Anderson and J.-H. Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[3] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.

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

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

[6] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 148–156, December 1993.

[7] S. S. Fu and N.-F. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(6):628–639, June 1997.

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

[9] T.-L. Huang and J.-H. Lin. An assertional proof of a lock synchronization algorithm using fetch_and_store atomic instructions. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*, pages 759–768. IEEE, 1994.

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

[11] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):673–685, July 2001.

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

[13] P. Magnusson, A. Landin, and E. Hagersten. Oueue locks on cache coherebt multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE, April 1994.

[14] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[15] J.-H Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.