

國立交通大學

資訊工程學系

博士論文

互斥問題在空間與遠端存取次數的最佳解

Tight Bounds on Space and Remote Reference Time Complexity of
Mutual Exclusion

研究生：陳勝雄

指導教授：黃廷祿 教授

中華民國九十七年二月

互斥問題在空間與遠端存取次數的最佳解
Tight Bounds on Space and Remote Reference Time Complexity of Mutual
Exclusion

研究生：陳勝雄

Student : Sheng-Hsiung Chen

指導教授：黃廷祿

Advisor : Ting-Lu Huang

國立交通大學
資訊工程學系
博士論文



A Dissertation

Submitted to Department of Computer Science

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

February 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年二月

互斥問題在空間與遠端存取次數的最佳解

學生：陳勝雄

指導教授：黃廷祿

國立交通大學 資訊工程學系 博士班

摘 要

互斥問題為非同步共享記憶體系統中的基本問題，用來管理系統中的資源。本論文針對此問題，分別就空間使用與遠端存取次數上提出最佳解。

針對像嵌入式即時系統這樣具有時間與資源限制的環境，互斥演算法應該符合公平性並且降低記憶體的使用。在文獻中，已有數個演算法僅用一個共享變數並且具有公平性。然而，這些演算法使用一些從未在任何系統中出現的假設性指令來設計。在不使用這樣的指令之下，我們首先提出兩個具公平性的演算法，並且僅需多用一個共享變數。所採用的指令為常見於一般系統的 *fetch&store* 與 *read/write*。第一個演算法符合 bounded bypass 條件。第二個則是改進第一個演算法，使其達到 FCFS 的公平性。改進公平性所需的代價為需更大的共享變數，在第一個演算法中共享變數大小為 $2\log_2(n+1)$ 位元，第二個演算法則需 $1 + 3\log_2(n+1)$ 位元，其中 n 代表所有 process 的個數。此外，我們進一步去證明在使用相同指令的條件下，至少需兩個共享變數才能達到 bounded bypass 的公平性。因此，就共享變數的使用個數上，所提出的演算法為最佳解。

而針對分散式共享記憶體系統，近期的研究主要為設計降低遠端存取次數的互斥演算法。頻繁地遠端存取會產生大量記憶體與處理器之間的流量，進而降低系統的效能。在此研究方向上，我們提出一個遠端存取次數的 lower bound。所假設的系統為採用通用 read-modify-write 指令的分散式共享記憶體系統。此通用 read-modify-write 指令為一般常見於系統一次存取一

個共享變數的不可分割指令之一般化模型，因此所提出的 lower bound 適用於所有採用這類指令的系統。再者，根據黃廷祿博士於 *ICDCS'99* 提出的演算法，此 lower bound 為最佳。

關鍵字：互斥問題、共享記憶體系統、嵌入式即時系統、公平性、空間複雜度、時間複雜度、最佳解。



Tight Bounds on Space and Remote Memory Reference Time Complexity of Mutual Exclusion

Student: Sheng-Hsiung Chen

Advisor: Dr. Ting-Lu Huang

Department of Computer Science
National Chiao Tung University

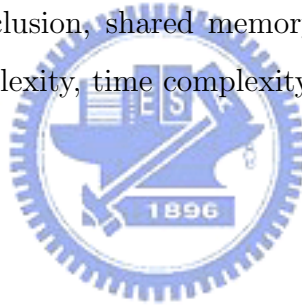
ABSTRACT

The mutual exclusion problem is fundamental to resource allocation in asynchronous shared memory systems. In this dissertation we present mutual exclusion algorithms with fairness and the minimum number of shared variables, and then show a tight bound on remote reference time complexity.

For shared memory systems under time and memory constraints such as embedded real-time systems, a mutual exclusion mechanism that is both fair and space-efficient can be highly valuable. Several algorithms that utilize only one shared variable and guarantee a certain level of fairness have been proposed. However, these use hypothetical read-modify-write primitives that have never been implemented in any system. We present two fair algorithms that do not use such primitives, but each algorithm has one additional shared variable. The proposed algorithms employ commonly available primitives, *fetch&store* and *read/write*, on two shared variables. The first algorithm satisfies the bounded bypass condition. The second is an improvement on the first that satisfies the FCFS condition, which is the most stringent fairness condition. The improvement is at the cost of increasing the size of a shared variable from $2 \log_2(n + 1)$ bits to $1 + 3 \log_2(n + 1)$ bits, where n is the number of processes. In addition, it is shown that achieving the bounded bypass condition using the same set of primitives requires two shared variables. Both of the algorithms are thus space-optimal in terms of the number of shared variables.

For distributed shared memory (DSM) systems, recent work on this problem has focused on the design of mutual exclusion algorithms that minimize the number of remote memory references, which generate processor-to-memory traffic and therefore may result in a bottleneck. We establish a lower bound of three on remote reference time complexity for mutual exclusion algorithms in a DSM model where processes communicate by means of a general read-modify-write primitive that accesses at most one shared variable in one instruction. Since the general read-modify-write primitive is a generalization of a variety of atomic primitives that have been implemented in multiprocessor systems, the lower bound holds for all mutual exclusion algorithms that use such primitives. Additionally, the lower bound is tight because it matches the upper bound of Huang's algorithm proposed in ICDCS'99.

Key words: Mutual exclusion, shared memory systems, embedded real-time systems, fairness, space complexity, time complexity, tight bounds



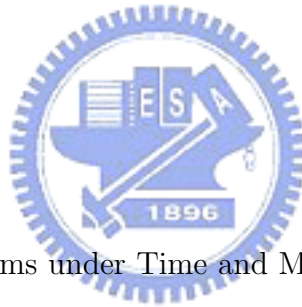
Acknowledgment

I would like to express my sincere thanks to my advisor, Prof. Ting-Lu Huang, for his supervision and perspicacious advice. Special thanks are due to my committee members: Prof. Shih-Kun Huang, Prof. Chung-Ta King, Prof. Ce-Kuen Shieh, Prof. Shi-Chun Tsai, Prof. Yih-Kuen Tsay and Dr. Da-Wei Wang for their valuable comments and encouragement.



Contents

Abstract in Chinese	i
Abstract in English	iii
Acknowledgment	v
Contents	vi
List of Figures	viii
1 Introduction	1
1.1 Algorithms for Systems under Time and Memory Constraints	3
1.2 Algorithms for Systems Whose Memory Has Locality	6
1.3 Contributions	11
1.4 Organization	11
2 System Models and Definitions	13
2.1 Shared Memory Model	13
2.2 Distributed Shared Memory Model	16
2.3 The Mutual Exclusion Problem	17
2.4 An Indistinguishability Relation	19
3 Related Algorithms	23
3.1 The MCS Lock	23
3.2 The CL Algorithm	26

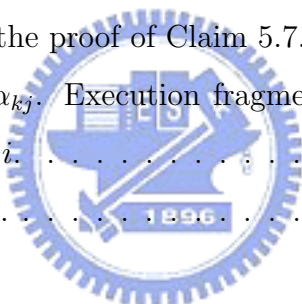


3.3	Huang’s Algorithm	30
3.3.1	The Algorithm	30
3.3.2	A Correctness Argument	35
4	Tight Bound on Space Complexity	40
4.1	The 2-bounded-bypass Algorithm	40
4.1.1	An Informal Description of the Algorithm	41
4.1.2	Proposed Algorithm	44
4.1.3	Proof Outlines	46
4.2	The FCFS Algorithm	49
4.2.1	An Informal Description of the Algorithm	50
4.2.2	Proposed Algorithm	54
4.2.3	An Informal Correctness Argument	55
4.3	An Impossibility Result	56
4.4	Summary	60
5	Tight Bound on RMR Time Complexity	62
5.1	The General RMW Primitive	62
5.2	An RMR Time Complexity Lower Bound	63
5.2.1	Basic Properties	64
5.2.2	Proof Outline	67
5.2.3	Detailed Proof	70
5.3	Summary	79
6	Conclusions and Future Work	80
6.1	Tight Bound on Space Complexity	80
6.2	Tight Bound on RMR Time Complexity	81
6.3	Future Work	81
	Bibliography	84

List of Figures

3.1	<i>fetch&store</i> , <i>compare&swap</i> and <i>swap&compare</i> primitives.	24
3.2	The MCS lock.	25
3.3	An execution of the MCS lock. An arrow from node p to node q indicates that process q has updated process p 's <i>Next</i> variable so that p is aware of the identity of its successor.	26
3.4	The CL algorithm.	27
3.5	An execution of the CL algorithm. A gray node indicates a process that has finished one life cycle. An upward arrow from a process points to the process's predecessor, and a downward arrow from a process, which must be a controller, points to the tail of a waiting list to which the process is responsible.	29
3.6	Huang's algorithm.	31
3.7	An execution of Huang's algorithm in Fig. 3.6. A gray node indicates a process that has finished one life cycle. An upward arrow from a process points to the process's predecessor, and a downward arrow from a process, which must be a controller, points to the tail of the waiting list to which the process is responsible. The label of a downward arrow from a process represents the permission word conveyed to the tail by the process.	32

4.1	An execution of the 2-bounded-bypass algorithm. A gray node indicates a process that has finished one life cycle. The symbol \triangle denotes an arbitrary value. An arrow from process a to b represents that a has the identity of b	43
4.2	The 2-bounded-bypass algorithm.	45
4.3	An execution of the FCFS algorithm. The notation is the same as that in Fig. 4.1.	51
4.4	The FCFS algorithm.	53
4.5	The execution for the proof of Theorem 4.7.	59
5.1	A goal execution extended from α_{ij} in which $time(i, \alpha_{ij}) \geq 2$	68
5.2	A goal execution extended from either α_{ij} or α_{ik} . We write e to denote the RMR step from i to j	70
5.3	Shared variables for the proof of Claim 5.7.1.	73
5.4	Executions α_{ij} and α_{kj} . Execution fragment α'' ends with the first RMR step from j to i	75
5.5	Executions in Case 1.	77



Chapter 1

Introduction

The mutual exclusion problem [18] is fundamental in multiprocessing systems for managing access to a single indivisible resource. In mutual exclusion, a process accesses the resource within a distinct part of code known as the *critical region*. A process executes trying and exit regions, respectively, before and after executing the critical region, to guarantee the following basic requirements.

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

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

The progress condition is necessary for the system to make any progress at all. However, an algorithm satisfying the condition does not guarantee that the critical region is granted fairly to different processes; for example, it allows one process to be repeatedly granted access to its critical region while other users trying to gain access are forever prevented from doing so. This situation is known as lockout, or starvation. Therefore, there are other fairness conditions of granting the critical region, several of which are enumerated in the following.

Lockout Freedom: A mutual exclusion algorithm is said to be lockout-free if no process can be kept waiting indefinitely either for the critical region or for the remainder region.

The next two conditions constrain the number of processes that may bypass a requesting process. To define such conditions, a definition is needed to specify when a process has made a request in its trying region. We adopt a widely-used definition that assumes the trying region is composed of a doorway and a waiting part. Only the entry to the waiting part of the trying region bounds the possible orders of entry to the critical region.

Bounded Bypass: A mutual exclusion algorithm is said to be bounded-bypass if it is b -bounded-bypass for some constant b . A mutual exclusion algorithm is defined to satisfy the b -bounded bypass property if no process that has finished its doorway can be bypassed more than b times by any other process when competing for a resource.

FCFS: The most stringent fairness condition is the first-come-first-served (FCFS) property that if a process i passes through its doorway before j performs a step in its doorway, then j can not enter its critical region before i does so. It is clear that a FCFS algorithm is also bounded-bypass.

Starting with an algorithm by Dijkstra [18], early work on this problem was focused on improving Dijkstra's algorithm by guaranteeing fairness conditions described above or by weakening the type of shared memory that is used [33, 17, 20, 34, 9, 38]. Due to the increasing interest on embedded real-time systems, we address that none of the previous algorithms is feasible for such systems, and proposes suitable algorithms in Chapter 4.

In contrast, recent work on the mutual exclusion problem has focused on the design of algorithms that reduce the number of remote memory references, which may produce a large amount of processor-to-memory traffic in shared memory systems. For this direction of research, we show a tight bound on the number of remote memory references in Chapter 5.

1.1 Algorithms for Systems under Time and Memory Constraints

Embedded real-time systems, *e.g.*, automotive control systems, mobile computing devices and home electronics, have received increasing interest in recent years. An algorithm for such systems should consider time and memory constraints. The time constraint imposes a deadline for each process in executing a particular job because the process often interacts with users or a dynamic environment. Additionally, embedded systems often have small memory (about 32–64 kBytes) since minimizing production costs, weight and power consumption are primary concerns in their designs [25, 42, 43]. As shown below, a mutual exclusion algorithm, in particular, should consider fairness and space efficiency.

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

A space-efficient mutual exclusion algorithm largely focuses on reducing the memory consumption. This requirement is crucial for systems under the memory constraint. In terms of the space complexity, most n -process mutual exclusion algorithms in previous literature use at least n shared variables, as shown in surveys by Anderson *et al.* [5] and Raynal [39]. For systems with limited memory, an algorithm using a constant number of shared variables would be more suitable.

For systems under time and memory constraints, we provide two fair and space-efficient mutual exclusion algorithms in Chapter 4. A 2-bounded-bypass algorithm with two shared variables is first presented to show the basic idea. A FCFS algorithm, which is based on the first algorithm, and uses the same number of shared variables, is then presented. The cost at improving the fairness from bounded bypass to FCFS is that the size of a shared variable is increased from $2 \log_2(n + 1)$ bits to $1 + 3 \log_2(n + 1)$ bits, where n denotes the number of all processes.

In terms of the fairness, both of the proposed algorithms satisfy bounded bypass, so that a process in either algorithm can roughly estimate the waiting time. (Note that a FCFS algorithm is also bounded-bypass.) For instance, in the 2-bounded-bypass algorithm, a process cannot be bypassed more than $2(n - 1)$ times by other processes after its requesting the critical region. By contrast, a process might be bypassed without limitation in an algorithm that does not satisfy bounded bypass, easily violating the deadline for executing a particular job.

In terms of the space complexity, only two shared variables are utilized in each of the algorithms. Moreover, no dynamic memory allocation is needed when executing the algorithm, so the system overhead is reduced. Since mutual exclusion is a basic synchronization mechanism frequently used in multiprocessing systems both in operating system kernel level and in users' application level [37], the system performance can be significantly improved.

In addition to atomic *read* and *write* primitives, both of the algorithms are implemented by *fetch&store*, which atomically writes a value into a shared variable and returns the old value of the same variable. Burns and Lynch [11] showed that n shared variables are necessary to solve the n -process mutual exclusion problem if only *read* and *write* are available. Fich *et al.* [21] recently extended the linear lower bound to systems that support *conditional* read-modify-write (RMW) primitives, such as *compare&swap*. A primitive is said to be RMW provided that it reads the value of a shared variable and changes the value of the shared variable in a single step. An RMW primitive is said to be conditional provided that it changes the value of a variable only if the variable has a particular value. Hence, some primitives other

than *read/write* and conditional RMW primitives are needed to decrease the space requirement. Primitive *fetch&store* is adopted to implement the algorithms since it is commonly supported in modern microprocessors such as a series of processors of Intel and AMD, Motorola 88000, and SPARC [40], and is also available in the ARM processor family [1]¹, which is arguably the most popular embedded architecture today. Thus, *fetch&store* improves the portability of the algorithm.

Several algorithms that use only a single shared variable and guarantee a certain level of fairness have been presented. For instance, Fischer *et al.* [23] devised a FCFS algorithm, and Burns *et al.* [10] devised a bounded-bypass algorithm and a lockout-free algorithm. Unfortunately, all of these algorithms used hypothetical RMW primitives that have never yet been implemented in any system. In contrast, none of the algorithms we propose use a hypothetical RMW operation, and each of them requires only one more shared variable than these algorithms.

The proposed algorithms are inspired by the circular list-based mutual exclusion algorithm presented by Fu and Tzeng [24, 30]. (Fu and Tzeng's algorithm is referred to as the CL algorithm throughout the rest of the dissertation.) The proposed algorithms, like the CL algorithm, organize waiting processes into lists, but pass the permission within and among lists very differently. The CL algorithm may block a process in the exit region. However, the proposed algorithms eliminate this drawback. Whereas Fu and Tzeng reduced the number of remote memory references, our algorithms target the space complexity and guarantee a certain level of fairness.

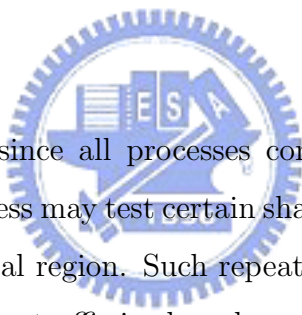
Furthermore, we prove that two shared variables are necessary to solve the mutual exclusion problem with b -bounded bypass for any constant b using only *fetch&store* and *read/write*. This impossibility result is proven by showing a more general result, that two object instances are required to implement a bounded-bypass mutual exclusion algorithm when using only historyless objects, regardless of the size of the objects. The definition of a historyless object is given by Fich *et al.* [22] and is restated in Section 4.3. According to the definition, shared variables associated with

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

fetch&store and *read/write* belong to the class of historyless objects, so the more general result implies the proposed algorithms are space-optimal. Informally, an object is historyless if applying a sequence of operations yields the same value in the object as applying just the last nontrivial operation in the sequence. A nontrivial operation is one that writes a value to the object.

The lower bound proof technique is related to an elegant method introduced by Burns and Lynch in proving the lower bound of n on the number of *read/write* objects required to solve the n -process mutual exclusion problem [11]. Their method, called *covering argument*, aims at *read/write* objects, and is generalized herein to historyless objects.

1.2 Algorithms for Systems Whose Memory Has Locality



In shared memory systems, since all processes communicate through the shared memory, each competing process may test certain shared variable(s) repeatedly while it is waiting to enter its critical region. Such repeated testing may produce a large amount of processor-to-memory traffic in shared memory systems, heavily degrading the system performance. This problem can be avoided in two architectural paradigms of shared memory systems: distributed shared memory (DSM) systems, in which each process has a local portion of shared memory, and cache coherent (CC) systems, in which each process has a local cache [37]. In DSM systems, a memory reference to a shared variable will not cause interconnect traffic if the variable is stored in the local portion of shared memory. In CC systems, whether a memory reference causes interconnect traffic depends on the caching protocol. Generally speaking, the first reference (be it read, write, or both) to a shared variable will cause interconnect traffic and establish a cached copy. Subsequent references, however, will not cause traffic until the cached copy of the shared variable is updated or invalidated. In general, a memory reference is regarded as *local* if it does not cause

any interconnect traffic; otherwise, it is *remote*.

Much work on the mutual exclusion problem has focused on the design of *local-spin* algorithms, which reduce the number of remote memory reference (RMR) steps by busy waiting only on locally-accessible shared variables. A number of performance studies [6, 8, 26, 31, 37, 41] have shown that synchronization algorithms minimizing the number of RMR steps have the best performance.

To evaluate mutual exclusion algorithms, the conventional time complexity, which counts all steps for one process in the worst case, might be inappropriate. This is because in any algorithm in which a process enters a busy-waiting loop when its critical region is unavailable, the worst case number of steps taken by one waiting process is unbounded. In other words, the conventional time complexity yields no useful information concerning the performance of such algorithms. Since the number of RMR steps significantly reflects the performance of an algorithm, Anderson and Yang [7] were the first to propose the number of RMR steps as a time complexity metric. To be more specific, the RMR time complexity of a mutual exclusion algorithm is the worst case number of RMR steps taken by any single process to enter and exit its critical region once. One may consider the amortized number of RMR steps instead of the worst case number as the RMR time complexity of an algorithm. But, as Anderson and Yang did, we adopt the worst case number rather than the amortized one because of the following reasons.

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

Throughout the rest of this dissertation, the RMR time complexity means the worst case RMR time complexity.

Known constant RMR time algorithms. In the literature, with some read-modify-write primitives in addition to atomic *read* and *write*, many mutual exclu-

sion algorithms of constant RMR time complexity are proposed:

- Anderson [8] proposed a constant RMR time algorithm for CC systems using *fetch&increment* and *fetch&add*.
- Graunke and Thakkar [26] proposed a constant RMR time algorithm for CC systems using *fetch&store*.
- Mellor-Crummey and Scott [37] first proposed an algorithm (referred to as the MCS lock in literature) for both CC and DSM systems using *fetch&store* and *compare&swap*.
- Craig [14], Magnusson *et al.* [36], and Huang and Lin [29] independently proposed the same constant time algorithm with *fetch&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 [4] presented a genetic constant RMR time algorithm for both CC and DSM systems using *fetch& ϕ* .

For more details of these algorithms, see the recent survey paper [5] of Anderson *et al.*

Because of these constant RMR time algorithms, the asymptotic tight bound on RMR time complexity is $\Theta(1)$. From a theoretical point of view, constant time is the best an algorithm can achieve in the RMR time complexity. Nevertheless, some researchers such as Fu and Tzeng [24, 30] continue to strive for minimizing the number of RMR steps. We consider it worthwhile to reduce the number as much as possible. In practice, remote memory references are orders of magnitude slower than references to the local memory. And mutual exclusion is a basic synchronization mechanism frequently used in multiprocessing systems both at the operating system kernel level and the users' application level [37]. Consequently, minimizing the number of RMR steps yields considerable performance improvement.

Our result for this direction of research is a tight bound on the number of RMR steps needed to solve the mutual exclusion problem in DSM systems. We prove

three is a lower bound on RMR time complexity. The lower bound is tight because it matches the upper bound of the algorithm proposed by Huang in ICDCS'99 [28]. (The algorithm is referred to as Huang's algorithm throughout the rest of the dissertation.) To prove the correctness of Huang's algorithm, we sketch a proof in Section 3.3.2.

Huang's algorithm is related to the MCS lock [37] and the CL algorithm by Fu and Tzeng [24, 30]. Fu and Tzeng tried to improve the MCS lock, whose RMR time complexity is four, and obtained a better algorithm in terms of the amortized RMR time complexity. But, in the CL algorithm, some process in its exit region (*i.e.*, the code fragment after executing its critical region) may take an unbounded number of RMR steps for the purpose of scheduling other competing processes. Thus, the worst case number of RMR steps taken by some process is unbounded, *i.e.*, the RMR time complexity is unbounded. Huang follows the line of their algorithm but eliminate the above drawback.

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

Known Lower Bounds on RMR time complexity. Several related lower bounds have been proved in the literature. All of these bounds are asymptotic. Anderson and Yang [7] first initiated a series of studies of lower bounds on RMR time complexity. They established a trade-off between the amount of contention, which was defined by Dwork *et al.* [19], and the RMR time complexity. The amount of contention of an algorithm is the maximum number of processes that are enabled to access the same shared variable simultaneously. Since our aim is minimizing the number of RMR steps, we focus on the RMR time complexity when contention may equal the number of all processes. Applying their result to the model with the general RMW primitive, we have that $\Omega(\log_c n)$ RMR steps are required in both DSM and CC systems, where c is the amount of contention and n is the number of processes. Thus, the lower bound on RMR time complexity is $\Omega(1)$, a trivial bound, when contention is n . Then, Cypher [15] showed a lower bound of $\Omega(\log \log n / \log \log \log n)$ on RMR time complexity in DSM and CC systems with only atomic *read* and *write* primitives. This result implies that there is no constant time mutual exclusion algorithm if only *read* and *write* are available. He went on to show that the lower bound holds even if conditional RMW primitives are available in addition to *read* and *write*. In a later work, Anderson and Kim [2] improved Cypher's lower bound to $\Omega(\log n / \log \log n)$. Cypher's lower bound and the improved bound by Anderson and Kim hold for *read*, *write* and conditional RMW primitives, whereas ours holds for all commonly-available primitives that access at most one shared variable in an instruction.

In addition, Kim and Anderson [32] provided an RMR time complexity lower bound for *adaptive* mutual exclusion algorithms in which the RMR time complexity is a function of the number of contending processes. They showed that for any k , there exists some n such that, for any n -process mutual exclusion algorithm based on *read*, *write* or conditional RMW primitives, there exists an execution involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ RMR steps to enter and exit its critical region. The result applies to both DSM and CC systems. In another paper [3], Anderson and Kim showed that for any n -process mutual exclusion al-

gorithm based on *non-atomic read* and *write*, there exists an execution involving only one process in which that process performs $\Omega(\log n / \log \log n)$ RMR steps in DSM systems to enter its critical region. Moreover, these RMR steps must access $\Omega(\sqrt{\log n / \log \log n})$ distinct remote shared variables, which implies that the process performs $\Omega(\sqrt{\log n / \log \log n})$ RMR steps in CC systems to enter its critical region.

Unlike the researchers who provided related lower bounds on the RMR time complexity, we establish a lower bound only for DSM systems; the lower bound proof herein is not applicable to CC systems. Future work is needed to establish the exact lower bound in CC systems.

1.3 Contributions

In summary, we first provide two fair and space-efficient algorithms for shared memory systems without resorting to any hypothetical primitive, and also show that the proposed algorithms are space-optimal in terms of the number of shared variables, making them highly valuable for systems under time and memory constraints.

We then improve the tight bound of mutual exclusion algorithms on RMR time complexity from $\Theta(1)$ to three in DSM systems. From the complexity-theoretic point of view, it may not be so surprising. But, this result is of importance for algorithm designers. Focus of mutual exclusion algorithms for shared memory systems for the last 15 years has been on minimizing the number of remote memory references [14, 24, 28, 30, 37]. The tight bound shows that it is impossible to obtain any better algorithm than Huang's algorithm in terms of minimizing the number.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 provides the system models and definitions of the mutual exclusion problem. Chapter 3 reviews the MCS lock, the CL algorithm and Huang's algorithm, which inspire our algorithms. Chapter 4 presents the space-optimal mutual exclusion algorithms for systems under

time and memory constraints. Chapter 5 presents the tight bound on the RMR time complexity in DSM systems. Conclusions and future directions for this research are finally drawn in Chapter 6.



Chapter 2

System Models and Definitions

The purpose of this chapter is to introduce formal models that are adopted. We first describe the shared memory model and then extend it to the distributed shared memory model. The only difference between these two models is that the shared memory of the latter has locality. The shared memory model is adopted in Chap. 4, where a tight bound on the number of shared variables is provided; while, the distributed shared memory model is utilized in Chap. 5 to present a tight bound on the number of remote memory references.

Besides, the mutual exclusion problem is formally defined. And, an indistinguishability relation is defined in order to prove the impossibility results in this dissertation.

2.1 Shared Memory Model

The model of an asynchronous shared memory system is based on the model described by Lynch in [35].

An algorithm in a 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.

Each shared variable $v \in \mathcal{V}$ has an associated set of values, among which some are designated as the initial values, I_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;
- $\Pi_i : \{(v, f)_i \mid v \in \mathcal{V} \text{ and } f \text{ is a function mapping from the value set of } v \text{ to the same set}\}$. Informally, Π_i specifies the steps that i may execute. Each step $(v, f)_i$ is a read-modify-write operation that atomically reads the current value of v , say old , and writes back $f(old)$ to the same variable v . That is, step $(v, f)_i$ means that process i accesses v by executing $\text{RMW}(v, f)$.

The system is asynchronous. That is, process steps do not necessarily take place in lock-step synchrony; rather, they may happen in an arbitrary order.

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

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

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

1. Suppose $(t, (v, f)_i, t')$ is an arbitrary transition in δ , with the same step of i . If $s(i) = t(i)$ and $s(v) = t(v)$, then $s'(i) = t'(i)$.

Informally, the present state of i and the present value of v uniquely determine the state of i after i takes step $(v, f)_i$.

2. $s'(v) = f(s(v))$.

The new value of v is determined by the function f and the current value of v .

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

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 any system state t at which $t(i) = s(i)$ holds, there exists a system state t' such that $(t, (v, f)_i, t') \in \delta$.

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

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

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

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

- $|\alpha|$: the length of α . (if α is a finite fragment)
- $\alpha|i$: the subsequence of α consisting of all steps of process i in α .
- $Pro(\alpha)$: the set of processes that take at least one step in α .
- $Var(\alpha)$: the set of shared variables accessed by any step in α .

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

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

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

2.2 Distributed Shared Memory Model

The distributed shared memory (DSM) model is the same as the shared memory model proposed in the previous section, except that in the DSM model, each process has a segment of shared memory that is local to it. We adopt the definition of a remote memory reference step proposed by Anderson and Yang [7], and thus use the number of remote memory reference steps as the RMR time complexity metric.

In the DSM model, \mathcal{V} is partitioned into disjoint nonempty subsets \mathcal{V}_i for each $i \in \mathcal{P}$. In other words, each variable belongs to a segment of shared memory that is local to a single process. This captures the essence of distributed shared memory systems. \mathcal{V}_i denotes the set of all shared variables located at process i . To a process i , a shared variable v is *remote* if $v \notin \mathcal{V}_i$; otherwise, it is *local*.

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

2.3 The Mutual Exclusion Problem

The shared memory model and the distributed shared memory model have been described so far. A formal definition of the mutual exclusion problem, which is similar the one proposed by Burns *et al.* in [10], is given below for both models.

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

For each process i , Σ_i is partitioned into nonempty disjoint subsets R_i , T_i , C_i and E_i . We say that a process i is in its remainder (R) region, trying (T) region, critical (C) region or exit (E) region at system state s if $s(i)$ belongs to R_i , T_i , C_i or E_i , respectively. A system state is said to be *idle* if all processes are in R . Each initial system state is assumed to be idle. In addition, we assume that the transition relation δ for a mutual exclusion algorithm satisfies the following *well-formedness* conditions.

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

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

For all steps, we assume that a step enabled in R or C never accesses a shared variable that may be accessed by a step enabled in T or E . Thus, a step taken in R or C will not affect the processes in T and E .

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

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

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

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

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

An algorithm satisfying the condition does not guarantee that the critical region is granted fairly to each individual process. To avoid entering a situation in which some process is denied indefinitely access to the critical region, it is often desirable to have some level of fairness other than the progress condition.

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

To define the fairness properties below, which guarantee a bound on the number of bypasses, we assume that the trying region of each process consists of two parts: a *doorway* followed by a *waiting* part. The doorway part is *wait-free*: its execution requires only a bounded number of steps. The following properties prevent any

process that has finished its doorway from being bypassed arbitrary times by any other process.

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

b -bounded bypass: Once a process i has passed through its doorway, no process can enter its C more than b times before i does so.

A mutual exclusion algorithm is said to be **first-come-first-served** (FCFS) if process i completes its doorway before j performs a step in its doorway, then j can not enter C before i does so. It is intuitively clear that a FCFS algorithm is also an algorithm satisfying the bounded bypass condition.

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

Then, a local-spin mutual exclusion algorithm can be formally defined as follows. This definition has been used implicitly or explicitly in related work about local-spin algorithms [5].

Definition 2.1 *A mutual exclusion algorithm is local-spin if its RMR time complexity is bounded, that is, a constant c exists such that its RMR time complexity is less than or equal to c .*

2.4 An Indistinguishability Relation

Variants of the notion of indistinguishability are frequently used to prove impossibility results in distributed systems [35]. Here, we first define an equivalence relation among system states, and then propose several ways to manipulate execution fragments.

Definition 2.2 Let P be a subset of \mathcal{P} and V a subset of \mathcal{V} . System states s and t are said to be indistinguishable to P with respect to V , denoted by $s \stackrel{P}{\sim}_V t$, if

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

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

Our definition is a generalization of the indistinguishability relation defined by Lynch [35]: when $V = \mathcal{V}$, the two indistinguishability relations become equal. The generalized version of indistinguishability makes it easier to define a weaker condition imposed on two system states such that an execution fragment executable from one system state is also executable from the other. Intuitively, it is enough to consider the set of all shared variables accessed in the execution fragment rather than the whole set \mathcal{V} . Furthermore, for a shared memory model whose memory has locality, this definition is useful in characterizing properties related to local shared memory, as we will see in Lemma 2.2 below and Lemma 5.2 in Section 5.2.1.

Now, we present two lemmas about ways to manipulate execution fragments based on the indistinguishability relation defined above. The first holds for both of the proposed models; in contrast, the latter holds only for the DSM model. These lemmas can be easily proved by the localized update and localized enabling assumptions.

Suppose that execution fragment α is executable from system state s . Let $P = \text{Pro}(\alpha)$ and $V = \text{Var}(\alpha)$. If $s \stackrel{P}{\sim}_V t$, Lemma 2.1 says that α is also executable from system state t . This is because each process and each shared variable involved in α have the same state and the same value, respectively, at s and t . By the localized update and localized enabling assumptions, an induction on each prefix of α can show that α is also executable from system state t . If, in addition, α is finite, the resulting system states $\alpha(s)$ and $\alpha(t)$ will be also indistinguishable to P with respect to V , i.e., $\alpha(s) \stackrel{P}{\sim}_V \alpha(t)$.

Lemma 2.1 *Let s and t be system states. Suppose that α is an execution fragment executable from s . Let $P = \text{Pro}(\alpha)$ and $V = \text{Var}(\alpha)$. If $s \stackrel{P}{\sim}_V t$, then α is also executable from t . If, in addition, α is finite, then $\alpha(s) \stackrel{P}{\sim}_V \alpha(t)$.*

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

The above lemma can be applied on both of the models, whereas the next lemma, Lemma 2.2, is only for the DSM model. Lemma 2.2 is for system states s and t that are indistinguishable to a process i consulting only shared variables in \mathcal{V}_i . Informally, if an execution fragment α executable from system state s contains neither RMR steps from i nor RMR steps to i , then no communication between i and any other process can occur in α . Lemma 2.2 says that $\alpha|i$ is also executable from all system states t at which $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$ holds. If, in addition, α is finite, then the resulting system states $\alpha(s)$ and $(\alpha|i)(t)$ will be also indistinguishable to process i with respect to \mathcal{V}_i .

Lemma 2.2 *Let s and t be system states and i a process. Suppose α is an execution fragment that is executable from s and contains neither RMR steps from i nor RMR steps to i . If $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$, then $\alpha|i$ is also executable from t . If, in addition, α is finite, then $\alpha(s) \stackrel{i}{\sim}_{\mathcal{V}_i} (\alpha|i)(t)$.*

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

Suppose that $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$. We show that $\alpha|i$ is also executable from t . Since $\alpha|i$ is an i -execution fragment and i does not access any remote shared variable in $\alpha|i$ (i.e.,

$Var(\alpha|i) \subseteq \mathcal{V}_i$, $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$ implies $s \stackrel{P}{\sim}_V t$ where $P = Pro(\alpha|i) = \{i\}$ and $V = Var(\alpha|i)$. Hence, by Lemma 2.1, $\alpha|i$ is also executable from t and if, in addition, α is finite, $(\alpha|i)(s) \stackrel{i}{\sim}_{\mathcal{V}_i} (\alpha|i)(t)$.

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

□

When α ending with an RMR step from i satisfies the assumptions on α in Lemma 2.2 except the last step, the following corollary says that $\alpha|i$ is also executable from t . Let α' be the prefix of α , just excluding the last step of α . By Lemma 2.2, $\alpha'|i$ is also executable from t and the states of i at $\alpha'(s)$ and $(\alpha'|i)(t)$ are the same. Thus, the RMR step from i at the end of α is also enabled at $(\alpha'|i)(t)$. Namely, the execution fragment $\alpha|i$ ($\alpha|i = \alpha'|i \circ$ the RMR step from i) is also executable from t . However, since the last step from i is an RMR step, the state of i at $\alpha(s)$ might be different from that at $(\alpha|i)(t)$.

Corollary 2.3 *Let s and t be system states and i a process. Suppose α is a finite execution fragment that is executable from s , ends with an RMR step from i , and contains neither RMR steps from i nor RMR steps to i except the last step. If $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$, then $\alpha|i$ is also executable from t .*

Chapter 3

Related Algorithms

Before presenting our results, three algorithms that aim at reducing the number of RMR steps are reviewed. They demonstrate how to order requests using RMW primitives. These algorithms also inspire the proposed algorithms in Chapter 4. The first is the MCS lock, which is proposed by Mellor-Crummey and Scott [37]; the second is the CL algorithm, which is proposed by Fu and Tzeng [24]; the last is Huang’s algorithm, which is proposed by Huang [28]. Due to Huang’s algorithm, the lower bound on RMR time complexity in Chapter 5 is tight. Notably, the original version of the CL algorithm suffers a deadlock error in the trying region, and the version herein is the one corrected by Huang and Shann [30].

Both of the MCS lock and Huang’s algorithm employ *fetch&store* and *compare&swap* to order requests to the critical region in a list-based way; while the CL algorithm employs *fetch&store* and *swap&compare* to do so in a circular-list-based way. The primitive *swap&compare* is a hypothetical RMW primitive defined by Fu and Tzeng. Definitions of these RMW primitives are given in Fig. 3.1.

3.1 The MCS Lock

As shown in Fig. 3.2, the MCS lock uses a *fetch&store* on a lock to chain competing processes as a list. Each process in the doorway, which is composed of line T1 in Fig. 3.2, executes *fetch&store* on the shared variable L (*i.e.*, the lock), announcing

```

fetch&store (shared variable v, value new)
  previous := v
  v := new
  return previous

compare&swap (shared variable v, value old, value new)
  previous := v
  if previous = old then
    v := new
  fi
  return previous

swap&compare (shared variable v, private variable old, value new)
  previous := v
  v := old
  old := previous
  if v = old then
    v := new
  fi

```

Figure 3.1: *fetch&store*, *compare&swap* and *swap&compare* primitives.

its identity and obtaining the identity of its predecessor if there is one. It then enters the waiting part of its trying region, which is composed of lines T2–T4. If the returned value is *nil*, *i.e.*, the requesting process is the head of the list, then it immediately enters its critical region. Otherwise, if it has a predecessor, it first writes a value to its predecessor’s *Next* variable, notifying its predecessor to refer back to its identity (T3). It then starts to spin on a locally-accessible shared variable until it is awakened (T4).

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

Shared variables:

$L \in \{\text{nil}, 0, 1, \dots, n-1\}$, initially nil $\triangleright L$ can be located at any process

for every $i \in \{0, \dots, n-1\}$:

$Spin(i) \in \{\text{true}, \text{false}\}$, initially true

$Next(i) \in \{\perp, 0, \dots, n-1\}$, initially \perp

$\triangleright Spin(i)$ and $Next(i)$ are located at process i

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

Private variables of i :

$pred, suc \in \{\text{nil}, 0, 1, \dots, n-1\}$, initially arbitrary

while true do

R: *Remainder region*

T1: $pred := \text{fetch\&store}(L, i);$

T2: **if** $pred \neq \text{nil}$ **then**

T3: $Next(pred) := i$

T4: **await** $\neg Spin(i)$; **fi**

\triangleright locally spin until $Spin(i) = \text{false}$

C: *Critical region*

E1: **if** $Next(i) = \perp$ **then**

E2: **if** $\text{compare\&swap}(L, i, \text{nil}) \neq i$ **then**

E3: **await** $Next(i) \neq \perp$;

\triangleright locally spin until $Next(i)$ is updated

E4: $suc := Next(i);$

E5: $Spin(suc) := \text{false};$ **fi**

\triangleright wake up its successor

E6: **else**

E7: $suc := Next(i);$

E8: $Spin(suc) := \text{false};$

\triangleright wake up its successor

E9: **fi**

E10: $Spin(i) := \text{true};$

\triangleright set $Spin(i)$ to true

E11: $Next(i) := \perp;$

\triangleright set $Next(i)$ to \perp

od

Figure 3.2: The MCS lock.

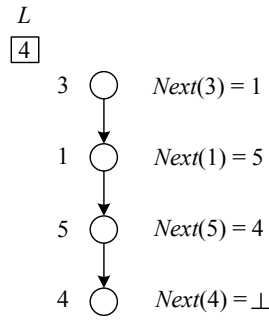


Figure 3.3: An execution of the MCS lock. An arrow from node p to node q indicates that process q has updated process p 's $Next$ variable so that p is aware of the identity of its successor.

returned value of *compare&swap* is i , *i.e.*, i has no successor, then *compare&swap* has modified L 's value to nil , setting the system state to the starting state.

Figure 3.3 illustrates a simple execution of the MCS lock. Process 3 first executes *fetch&store* in T1 and gets nil from L , so it enters C immediately. While process 3 is in C , processes 1, 5 and 4 execute T1 in turn. Each of processes 1, 5 and 4 updates its predecessor's $Next$ variable and then starts to wait. The permission is conveyed from 3 to 1, then from 1 to 5, and then from 5 to 4. After process 4 leaves C , if there is no other request, process 4 modifies L 's value to nil ; otherwise, it passes the permission to its successor.

The MCS lock satisfies mutual exclusion, progress and the FCFS condition. Inspecting the algorithm, the worst case number of RMR steps taken by any single process in T and E is four (Steps T1, T3, E2 and E5).

3.2 The CL Algorithm

Fu and Tzeng tried to improve the MCS lock and proposed the CL algorithm, which is better in terms of the amortized RMR time complexity. But, the FCFS condition is not satisfied. Furthermore, although the CL algorithm is bounded-bypass in the trying region, some process may be blocked in the exit region. Figure 3.4 is the CL

algorithm. Explanation of the algorithm follows.

Shared variables:

$L \in \{\text{nil}, 0, 1, \dots, n-1\}$, initially nil ▷ L can be located at any process
 for every $i \in \{0, \dots, n-1\}$:
 $\text{Spin}(i) \in \{\text{true}, \text{false}\}$, initially true ▷ $\text{Spin}(i)$ is located at process i

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

Private variables of i :

$\text{pred} \in \{\text{nil}, 0, 1, \dots, n-1\}$, initially arbitrary

```

while true do
R:   Remainder region
T1:   $\text{pred} := \text{fetch\&store}(L, i);$ 
T2:  if  $\text{pred} \neq \text{nil}$  then
T3:    await  $\neg \text{Spin}(i);$  fi ▷ locally spin until  $\text{Spin}(i) = \text{false}$ 
C:   Critical region
E1:  if  $\text{pred} = i$  then ▷ as a controller
E2:    while true do
E3:       $\text{pred} := \text{nil};$ 
E4:       $\text{swap\&compare}(L, \text{pred}, \text{nil});$ 
E5:      if  $\text{pred} = i$  then
E6:        break; ▷ leave the inner while loop
E7:      else
E8:         $\text{Spin}(i) := \text{true};$ 
E9:         $\text{Spin}(\text{pred}) := \text{false};$  ▷ wake up the tail of the waiting list
E10:       await  $\neg \text{Spin}(i);$  ▷ locally spin until  $\text{Spin}(i) = \text{false}$ 
E11:      fi
E12:    od
E13:  else
E14:     $\text{Spin}(\text{pred}) := \text{false};$  ▷ wake up its predecessor
E15:  fi
E16:   $\text{Spin}(i) := \text{true};$  ▷ set  $\text{Spin}(i)$  to  $\text{true}$ 
od

```

Figure 3.4: The CL algorithm.

As in the MCS lock, each process in its doorway, which is composed of line T1 in Fig. 3.4, executes *fetch&store* on the shared variable L to make public its identity and obtain the identity of its predecessor if there is one. The process then enters its waiting part, which is composed of lines T2 and T3, and starts to check whether it is the first process that references L (T2), either since system start-up or since the last step that the value nil was written back to L . If so, the process enters C ; otherwise it starts to spin on its spin variable (T3). Unlike the MCS lock,

the CL algorithm eliminates the remote memory reference that notifies a requesting process's predecessor to refer back to the process's identity (*i.e.*, step T3 in the MCS lock). As a result, the MCS lock orders processes in a list according to when they make requests, whereas the CL algorithm orders processes in the opposite order. For instances, suppose that processes 3, 1, 5 and 4 make requests in turn. In the MCS lock, they are linked into a list as shown in Fig. 3.3; while, in the CL algorithm, they are linked in the opposite order as shown in Fig. 3.5(a).

When a process i leaves C , if it does not get nil from L in T (*i.e.*, $pred \neq nil$), i just passes the permission to its predecessor (E14), and then enters R after setting its spin variable to $true$ (E16). Otherwise, if $pred = nil$, it is selected as a *controller* and has additional responsibility for servicing other requesting processes. It executes steps E2–E12 to take care of the followings. Two possibilities exist. If L is still equal to i , no other processes are interested in entering C . Process i writes nil to L when performing step E4 and moves to R . Otherwise, if L has some other process's identity, there is a list of waiting processes. Process i stores the value of L , which is the identity of the tail of the current waiting list, to $pred$ (as a result of E4) and passes the permission to the tail (E9). The permission will be conveyed along the list from the tail to the head. While the permission is being transmitted, process i , the head of the list, is blocked at E10.

After i passes E10, all processes in the waiting list have finished C but more processes may have arrived and have been kept waiting. Process i should go back to E2 to prepare for the next run of playing controller. It will be kept in this potentially unbounded number of runs of playing controller as long as there are processes interested in entering C .

Figure 3.5 depicts an example execution. Process 3 first takes step T1, gets nil from L and thus enters C immediately. At about the same time, processes 1, 5 and 4 execute T1 in turn. A waiting list, called list 1, is formed as shown in Fig. 3.3(a). In Fig. 3.5(b), process 3 leaves C , sets L to its identity and obtains the identity of the tail. It then passes the permission to the tail of the list (*i.e.*, process 4). The permission will be conveyed from 4 to 5, then from 5 to 1, and then

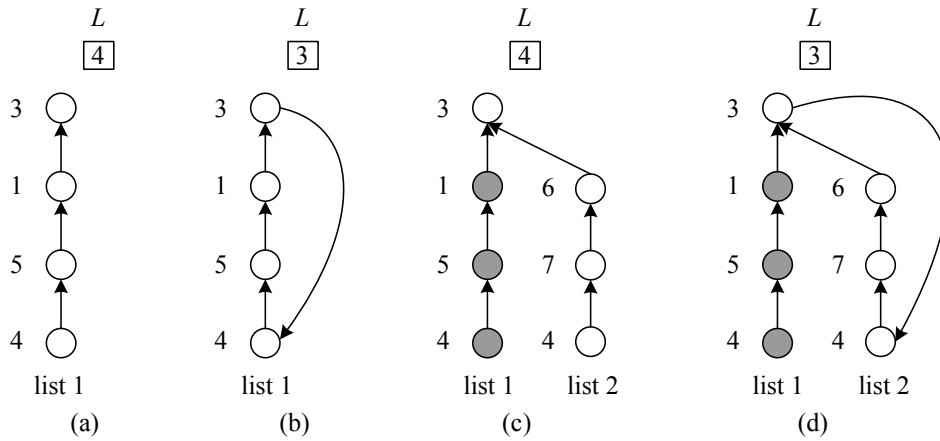


Figure 3.5: An execution of the CL algorithm. A gray node indicates a process that has finished one life cycle. An upward arrow from a process points to the process's predecessor, and a downward arrow from a process, which must be a controller, points to the tail of a waiting list to which the process is responsible.

from 1 to 3. Process 3 will be blocked until the permission is passed back to itself. As Fig. 3.5(c) shows, while the permission is transmitted along list 1, subsequent requesting processes form another waiting list, called list 2. In Fig. 3.5(d), the permission is conveyed back to process 3, the process takes the role of the controller again and redirects the permission to the process 4, which is the tail of list 2.

The concept of using a controller to convey the permission to the tail of a waiting list also appears in our algorithms in Chap. 4 and Chap. 5. The differences are which process is selected as a controller and how to pass the responsibility of controller to the next one.

The CL algorithm satisfies mutual exclusion, progress and bounded bypass in the trying region. But, since a process may be kept an unbounded number of times at the while loop in E , the RMR time complexity of the CL algorithm is unbounded.

3.3 Huang’s Algorithm

This section presents Huang’s algorithm, whose RMR time complexity is three. The key to minimizing the number of RMR steps is encoding different messages into an RMR step. Based on the algorithm, the lower bound result on RMR time complexity in Chapter 5 is tight.

The algorithm also satisfies bounded bypass and lockout-freedom besides the basic requirements. To argue the correctness, we sketch a proof in the end of the section.

3.3.1 The Algorithm

The algorithm is shown in Fig. 4.2. Figure 4.1 illustrates an example to help explain the working of the algorithm.

As in the MCS lock, Huang’s algorithm uses a *fetch&store* on a lock to link competing processes, but, as in the CL algorithm, it eliminates the remote memory references needed in the MCS lock to notify its predecessor to re-direct the link for each process in a list. With this modification, the CL algorithm proposed a way to pass the lock among processes. However, this way suffers from blocking in the exit region. To eliminate this drawback, the algorithm provides a new way to convey the lock.

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

We now explain the key idea of the algorithm. Each requesting process executes *fetch&store* on the shared variable L (*i.e.*, the lock) to announce its identity and obtain its predecessor’s identity if there is one. If the returned value is *nil*, the critical region is available and the requesting process enters the critical region immediately; otherwise, it waits by repeatedly testing its local spin variable. Since each process

Shared variables:

$L \in \{\text{nil}, 0, 1, \dots, 2n - 1\}$, initially nil $\triangleright L$ can be located at any process
 for every $i \in \{0, \dots, n - 1\}$:

$\text{Spin}(i) \in \{(\text{Head}, \text{Tail}) \mid \text{Head}, \text{Tail} \in \{\text{nil}, 0, 1, \dots, 2n - 1\}\}$, initially (nil, nil) $\triangleright \text{Spin}(i)$ is located at process i

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

Private variables of i :

$\text{id} \in \{i, n + i\}$, initially i
 $\text{pred} \in \{\text{nil}, 0, 1, \dots, 2n - 1\}$, initially arbitrary
 $\text{head}, \text{tail} \in \{\text{nil}, 0, 1, \dots, 2n - 1\}$, initially arbitrary

```

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

```

Figure 3.6: Huang's algorithm.

makes a request by executing *fetch&store* on the same variable L , a waiting list will be formed if some process has been in C . For instance, in Fig. 3.7(a), as process 3 is in C , all competing processes (1, 5, and 4) form a waiting list.

When a process leaves C , it takes an RMR step to write a value, called the permission word, to the spin variable of some waiting process. Since the waiting process is testing its spin variable repeatedly, the permission word in effect serves as a wake-up signal. In order to minimize the number of remote memory references, the permission word not only serves as permission to enter C , but also carries enough

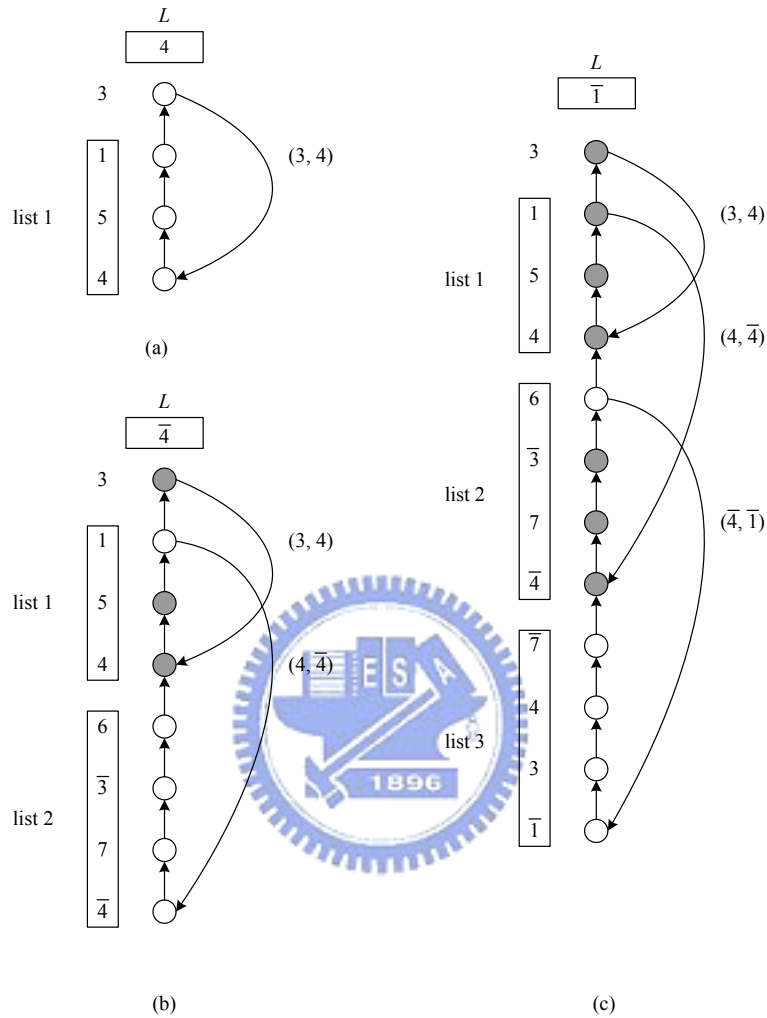


Figure 3.7: An execution of Huang's algorithm in Fig. 3.6. A gray node indicates a process that has finished one life cycle. An upward arrow from a process points to the process's predecessor, and a downward arrow from a process, which must be a controller, points to the tail of the waiting list to which the process is responsible. The label of a downward arrow from a process represents the permission word conveyed to the tail by the process.

information for processes to arrange among themselves the order to enter C , without using any other control word.

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

For example, in Fig. 3.7(a), when process 3 (the controller at the time) leaves C , it conveys the permission to process 4, the tail of the current waiting list, called list 1. Pair (3,4) serves as the permission, where 3 is used for each process receiving the permission to check whether it is the head of list 1, and 4 indicates the tail of the list and will be used to encode the next permission. The permission will be transmitted along list 1. In Fig. 3.7(b), when process 1 in list 1 leaves C , *i.e.*, all processes in the list have finished their critical regions, process 1 knows that it is the head of list 1 by checking whether its predecessor is 3. Process 1 encodes new information into the permission and redirects it to the tail of the current waiting list, called list 2.

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

In the trying region, a process in its doorway, which is composed of line T1 in Fig. 3.6, executes *fetch&store* on L . It then enters the waiting part, which is composed of lines T2 and T3. If the returned value of the primitive is nil , the requesting process enters its critical region immediately; otherwise, it waits by repeatedly testing its spin variable until the value is not equal to (nil, nil) (T3).

In the exit region, each process reads its spin variable and stores the permission word into its private variables *head* and *tail* (E1). A process will identify itself as a controller if the result of checking E2 is “yes”—that is, *pred* is equal to nil or *head*. If the process is not a controller, it just transmits the permission to its predecessor by executing E12. Otherwise, it first encodes new control information into a new permission word by executing steps E3–E8. Steps E3–E7 set the new value of *head*: if the controller gets nil from L , then *head* is set to its current identity; otherwise, *head* is set to the value of *tail* in the old permission word. This is because the value of *head* will be used by processes in the new waiting list to check whether it is the head of the list. Step E8 sets *tail* to the returned value of *compare&swap* on L , which is the identity of the tail of the new waiting list if there is one. If there is no new waiting list, E8 atomically modifies L 's value to nil . Otherwise, the controller redirects the modified permission word to the tail of the new list by executing E10.

The algorithm has been presented. It remains to explain the reason why each process uses different identities in any two consecutive life cycles. Each process alternately uses one of its identities to avoid a subtle situation. Although a process cannot appear more than once in a waiting list, it may appear in two neighboring lists. A process's identity in one life cycle is different from that in the next cycle since

a process always changes its current identity in E15. Therefore, no two identities of the same process in any two consecutive lists are the same. This is important for a process to determine whether it should act as the controller for the next waiting list. For example, in Fig. 3.7(c), process 3 in list 3 would not be able to tell the difference between 4 in list 3 and $\bar{4}$ in list 2 if process 4 uses the same identity. With the different identities, process 3 should pass the permission to process 4, rather than taking up the role of a controller. The situation occurs whenever a process at the tail of a waiting list, after having been given permission to enter C , quickly makes a new request in the next waiting list. Fortunately, the subtlety needs to be resolved only between two neighboring waiting lists, thus two identities for each process suffice.

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



3.3.2 A Correctness Argument

Mutual Exclusion

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

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

- Step E8 ($tail := compare\&swap(L, head, nil)$) assigns the current value of L

to *tail*, and modifies L 's value to *nil* only if $L = \text{head}$. If the step indeed modifies L 's value, it is regarded as *successful*. A successful E8 allows at most one process to gain permission.

- Each of E10 and E12 modifies some process's spin variable. Since the spin variables of any two processes are distinct, each of the two steps allows at most one process to gain permission.

According to the algorithm, a process that executes a successful E8 bypasses E10 since $\text{tail} = \text{head}$. Hence, a process in E executes exactly one of the following steps: successful E8, E10, or E12. That is, a process in E passes its permission to at most one process.

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

Lemma 3.1 *Huang's algorithm guarantees mutual exclusion.*



Lockout-freedom

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

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

Next, we try to divide all requests in a busy period into *lists*. A *list* in a busy period is a sequence of processes that execute T1 between the first T1, which obtains *nil* from L , and the following unsuccessful E8, or between an unsuccessful E8 and

the next unsuccessful one. Starting from the last process in a list, we can trace the whole list from the tail to the head through the value of $pred$ of each process in the list. A process that executes E8 is called a *controller*. If a controller executes an unsuccessful E8, then it defines a new list and is also called the controller of the new list. Otherwise, if a controller executes a successful E8, then it ends the busy period.

Lemma 3.2 *Huang's algorithm guarantees lockout-freedom.*

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

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

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

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

Basis: $i = 1$. List 1 contains all processes that make requests between the first T1 in the busy period and the unsuccessful E8 executed by the controller of list 1. Through the returned value of *compare&swap* in E8, the controller has the identity of the tail of the list. Lemma 3.1 implies that at most one process has permission at any system state. Thus, before the controller passes the permission to the tail, all requesting processes will be blocked at T3. The controller then executes E10 to

pass the permission to the tail by writing pair $(head, tail)$ to the tail's *Spin* variable, where *head* is the controller's identity and *tail* is the tail's identity. In list 1, each process except the head will not be selected as a new controller since $pred \neq nil$ and $pred \neq head$, and will pass the permission to its predecessor by executing E12. Thus, the permission will be conveyed along the whole list from the tail to the head so that each process in list 1 eventually enters *C*. When the head of list 1 leaves *C*, it identifies itself as a new controller since its *pred* is equal to the previous controller's identity (*i.e.*, $pred = head$).

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

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

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

Bounded Bypass

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

Lemma 3.3 *Huang's algorithm guarantees bounded bypass.*



Chapter 4

Tight Bound on Space Complexity

In this chapter, two algorithms are proposed for systems under time and memory constraints. Each of the algorithms utilizes constant two shared variables with *fetch&store* as well as *read/write*. The first algorithm satisfies the bounded bypass condition; the second is an improvement on the first that satisfies the FCFS condition. To improve the fairness, the FCFS algorithm increases the number of values taken on by a shared variable from $(n + 1)^2$ to $2(n + 1)^3$, where n is the number of all processes.

Additionally, a lower bound result shows that any bounded-bypass algorithm using the same set of primitives must utilize at least two shared variables. The proposed algorithms are therefore space-optimal. In other words, a tight bound of two on the number of shared variables is obtained.

The computational model adopted in this chapter is the shared memory model. The formal description of the model appears in Chapter 2.

4.1 The 2-bounded-bypass Algorithm

This section presents a bounded-bypass mutual exclusion algorithm using two shared variables, as shown in Fig. 4.2. Figure 4.1 illustrates an example to help explain the working of the algorithm.

In summary, the algorithm links competing processes as circular lists by *fetch&store*

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

According to the construction of a list, a competing process has the identity of its predecessor rather than its successor. Consequently, the permission is conveyed from the tail of a list to the head, resulting in the failure to meet the FCFS condition. The next section describes a modified algorithm that eliminates this drawback and achieves the FCFS condition by initiating an additional phase for every list to redirect the links in the list.

4.1.1 An Informal Description of the Algorithm

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

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

After announcing its request by executing *fetch&store* on L , each process enters

the waiting part of its trying region and starts to test P repeatedly until it has the permission to enter C . A controller repeatedly tests P until $P = nil$, which is the permission for a controller. A non-controller repeatedly tests P until P equals its identity, which indicates that it gains the permission. Since $P = nil$ initially, the first controller always gains the permission.

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

After a list is closed, the permission is transmitted along the list from the tail to the head, allowing each process in the list to enter C in order. If the controller is not the only process in the list, then it passes the permission to the tail of the list by setting P to the identity of the tail. Each non-controller in the list then hands the permission to its predecessor, by setting P to the identity of its predecessor, when it leaves C . However, if the predecessor is the head of the list, then the process passes the permission to the next waiting list rather than to the predecessor because the predecessor has left C . Some information is needed to check this situation, and is encoded into P . Let P hold a pair $(Receiver, Head)$, each being the identity of a process or nil .¹ The *Receiver* component serves the original purpose of P , indicating which process can enter C , and the *Head* component stores the identity of the head of the list so that each process can determine whether its predecessor is the head. If the predecessor's identity of a process is equal to *Head*, the process modifies *Receiver*'s value to nil instead of the predecessor's identity to convey the permission to the head of the next waiting list.

Figure 4.1 illustrates an execution of the algorithm. Variables L and P are respectively set to nil and (nil, Δ) , as shown in Fig. 4.1(a). The symbol Δ in the

¹Thus, P consists $(n + 1)^2$ distinct values, where n is the number of processes; while L consists $n + 1$ values.

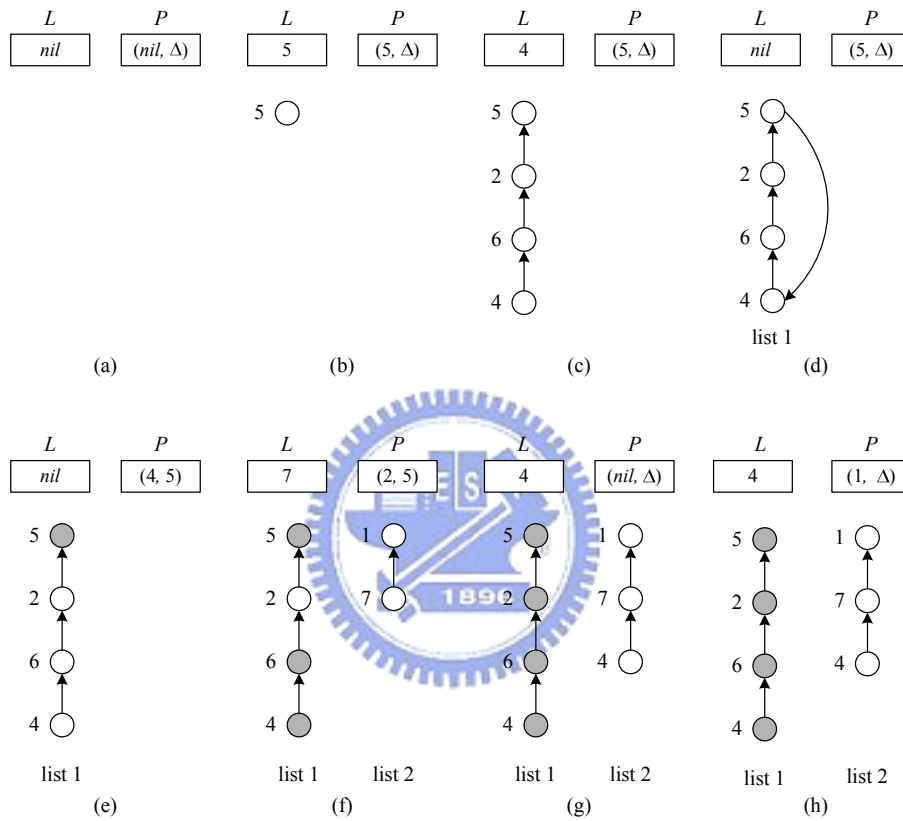


Figure 4.1: An execution of the 2-bounded-bypass algorithm. A gray node indicates a process that has finished one life cycle. The symbol Δ denotes an arbitrary value. An arrow from process a to b represents that a has the identity of b .

Head component represents that the component is not used at this time and can be an arbitrary value.

In Fig. 4.1(b), process 5 first makes a request by executing $fetch\&store(L, 5)$ and obtains nil from L . Since $Receiver = nil$, process 5 enters C after setting $Receiver$ to 5. While process 5 is in C , processes 2, 6 and 4 make requests in turn. Processes 5, 2, 6 and 4 form a list as shown in Fig. 4.1(c). Because none of processes 2, 6 and 4 succeeds in acquiring nil from L , they repeatedly test P until the $Receiver$ component of P equals their respective identities.

When leaving C , process 5 closes the list, called list 1, by executing $fetch\&store(L, nil)$. This operation obtains the identity of the tail and modifies L 's value to nil . The edge from 5 to 4 in Fig. 4.1(d) indicates that the returned value is 4, *i.e.*, the tail of the list is process 4. Process 5 then passes the permission to the tail and sets $Head$ to 5 by writing (4,5) to P , as shown in Fig. 4.1(e). In Fig. 4.1(f), process 4 gains the permission, enters C and then passes the permission to process 6. Similarly, process 6 gains the permission from process 4, enters C and then hands the permission to process 2 by writing (2, 5) into P . In Fig. 4.1(g), since the predecessor of process 2 is the head of the list, process 2 modifies $Receiver$'s value to nil to transmit the permission to the next waiting list, called list 2. Finally, in Fig. 4.1(h), because $Receiver$'s value has been changed to nil , process 1, which is the head of list 2, enters C after setting $Receiver$ to 1.

4.1.2 Proposed Algorithm

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

In the trying region, the doorway is composed of line T1 in Fig. 4.2, and the waiting part is composed of the rest (T2–T8). A process i in the doorway executes $fetch\&store(L, i)$ (T1) to announce its request, and then enters the waiting part. If $pred = nil$, *i.e.*, the returned value of T1 is nil , then the process identifies itself as

Shared variables:

$L \in \{nil, 1, \dots, n\}$, initially nil

$P \in \{(Receiver, Head) \mid Receiver, Head \in \{nil, 1, \dots, n\}\}$, initially (nil, Δ)

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

Private variables of i :

$pred, tail, receiver, head \in \{nil, 1, \dots, n\}$, initially arbitrary

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

Figure 4.2: The 2-bounded-bypass algorithm.

a controller and begins repeatedly checking P until the *Receiver* component of P equals nil (T4–T5). When $Receiver = nil$, process i sets $Receiver$ to i (T6) and then enters the critical region. In contrast, if $pred \neq nil$, then process i repeatedly tests P until $Receiver = i$ holds (T7–T8).

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

4.1.3 Proof Outlines

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

Mutual Exclusion

In the algorithm, whether a process in T can enter C depends on the value of $Receiver$. If $Receiver = nil$, then a controller waiting for nil in T is permitted to enter C , while if $Receiver = i$, $1 \leq i \leq n$, then only process i is permitted to do so. Inspection of the algorithm clearly indicates that only the process in E can modify $Receiver$'s value to nil or the identity of some other process using one of steps E4, E5, E7 or E8. (Although a controller in T modifies $Receiver$'s value by executing T6, it sets $Receiver$ to its identity, allowing no other process to enter C .) We show

that a *nil* can be taken as the permission for at most one process. Hence, a process in E allows at most one process to enter C . Additionally, since *Receiver* is set to *nil* initially, and a *nil* permits at most one process to enter C , at most one process can enter C from the starting state. Thus, the mutual exclusion condition is ensured.

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

Lemma 4.1 *At any reachable system state,*

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

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

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

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

Theorem 4.2 *The algorithm guarantees mutual exclusion.*

Progress

We argue that the algorithm satisfies the lockout-freedom condition, that if no process stays in C indefinitely, any process in T eventually enters C ; and any

process in E eventually enters R . A lockout-free algorithm is intuitively also an algorithm satisfying the progress condition.

Theorem 4.3 *The algorithm guarantees lockout-freedom.*

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

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

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

To prove that each requesting process eventually enters C , it remains to be shown that each controller receives the permission. Since *Receiver* is initially set to nil , the first controller always gains the permission. The controller closes the list, and conveys the permission to the tail of the list, when it leaves C . Since a closed list contains a finite number of processes, if no process stays in C indefinitely, then each process in the list eventually gains the permission to enter C . When the process next

to the controller receives the permission, since its $pred$ equals $Head$, it redirects the permission to the next controller by setting $Receiver$ to nil after passing through C . (From Lemma 4.1, if one controller is waiting for nil , exactly one such controller exists.) Thus, each controller eventually receives the permission. \square

Bounded Bypass

A process i is said to be in the doorway if $pc_i = T1$, and it is said to be in the waiting part if $pc_i \in \{T2, \dots, T8\}$. As shown in the proof of Theorem 4.3, a process is recorded in a waiting list after passing through its doorway (*i.e.*, after executing T1). Since a list does not receive the permission until each process in the previous list has left C , a process in a list may be bypassed by those processes in the same list and in the previous list. In addition, because a process can occur in a list at most once, a waiting process may be bypassed by any individual process at most twice. In other words, the algorithm satisfies 2-bounded bypass. The worst case, in which a process that has finished its doorway is bypassed twice by another process, may occur when a non-controller in a list quickly makes a request appending to the new list after receiving the permission. For example, in Fig. 4.1(f–h), process 7 is bypassed twice by process 4, which makes a request after receiving the permission. Consequently, the following theorem holds.

Theorem 4.4 *The algorithm guarantees 2-bounded bypass.*

4.2 The FCFS Algorithm

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

The FCFS algorithm follows the same concept of the 2-bounded-bypass algorithm, except that it initiates an additional phase to redirect the links in a list to meet the FCFS condition. Owing to the implementation of the communication

phase, the number of values taken on by the shared variable P increases from $(n+1)^2$ to $2(n+1)^3$.

4.2.1 An Informal Description of the Algorithm

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

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

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

Implementing this phase requires some communication mechanism. In the algorithm, the shared variable P is used for two purposes: to indicate which process is permitted to enter C , and to inform processes of their respective successors. The use of a shared variable for these two purposes is inspired by the algorithms [10] proposed by Burns *et al.* To serve both purposes, the variable holds a 4-tuple

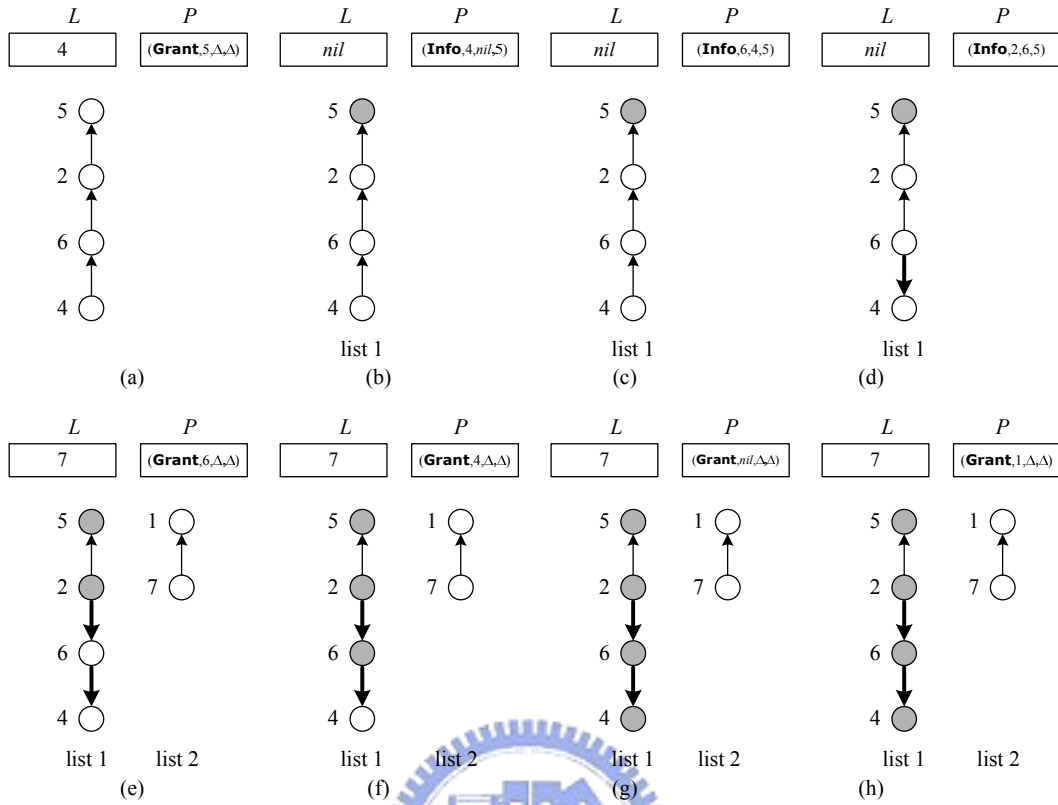


Figure 4.3: An execution of the FCFS algorithm. The notation is the same as that in Fig. 4.1.

$(Type, Receiver, Successor, Head)$, where $Type$ is a value in $\{\mathbf{Info}, \mathbf{Grant}\}$, and the other parts take on values from $\{nil, 1, \dots, n\}$. The number of values taken on by P in this algorithm is $2(n + 1)^3$, compared with $(n + 1)^2$ in the 2-bounded-bypass algorithm.

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

Figure 4.3 illustrates an execution of the algorithm. The sequence of requests in list 1 is the same as that given to the 2-bounded-bypass algorithm in Fig. 4.1, but the order in which the permission is conveyed among non-controllers is opposite.

Variables L and P are initially set to nil and $(\mathbf{Grant}, nil, \Delta, \Delta)$, respectively. In Fig. 4.3(a), processes 5, 2, 6, 4 make requests in turn and constitute a list. Because process 5 obtains nil from L , and receives a message that $Type = \mathbf{Grant}$ and $Receiver = nil$, it has the permission for a controller. It enters C after setting P to $(\mathbf{Grant}, 5, \Delta, \Delta)$. In contrast, processes 2, 6, and 4 repeatedly test P until they receive their respective messages. Process 5 performs $fetch\&store(L, nil)$ to obtain the identity of the tail, 4 in this case, and modifies L 's value to nil , when it leaves C . It then starts a communication phase by writing $(\mathbf{Info}, 4, nil, 5)$, as shown in Fig. 4.3 (b). This message notifies process 4 that it is the tail of the list (because $Successor = nil$), and that the head is process 5. Process 4 then receives the message and writes a new message $(\mathbf{Info}, 6, 4, 5)$ to process 6, as shown in Fig. 4.3(c). The new message informs process 6 that its successor is process 4, and that the head of the list is process 5. Similarly, process 6 receives the message from process 4, and writes a new message $(\mathbf{Info}, 2, 6, 5)$ to inform process 2, as shown in Fig. 4.3(d).

Process 2 receives the message written by process 6, and becomes aware that it is the immediate successor of the head, because its predecessor is the head of the list. This means that the communication phase is completed. Process 2 enters C , and conveys the permission to its successor, process 6, by writing $(\mathbf{Grant}, 6, \Delta, \Delta)$ into P , as shown in Fig. 4.3(e). Process 6 then gains the permission, and conveys it to process 4, as shown in Fig. 4.3(f). Since process 4 is the tail of the list, process 4 hands the permission to the next waiting list, by setting P to $(\mathbf{Grant}, nil, \Delta, \Delta)$, when it leaves C , as shown in Fig. 4.3(g). Process 1, which is the head of the next list, then receives the permission to enter C , as shown in Fig. 4.3(h), because $Type = \mathbf{Grant}$ and $Successor = nil$.

Shared variables:

$L \in \{\text{nil}, 1, \dots, n\}$, initially nil

$P \in \{(Type, Receiver, Successor, Head) \mid Type \in \{\mathbf{Info}, \mathbf{Grant}\},$

$Receiver, Successor, Head \in \{\text{nil}, 1, \dots, n\}\}$, initially $(\mathbf{Grant}, \text{nil}, \Delta, \Delta)$

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

Private variables of i :

$type \in \{\mathbf{Info}, \mathbf{Grant}\}$

$pred, tail, receiver, successor, head \in \{\text{nil}, 1, \dots, n\}$, initially arbitrary

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

Figure 4.4: The FCFS algorithm.

4.2.2 Proposed Algorithm

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

In the trying region, the doorway is composed of line T1 in Fig. 4.4, and the waiting part is composed of the rest (T2–T12). If the returned value of T1 is nil (*i.e.*, $pred = nil$), then process i identifies itself as a controller, and repeatedly checks P until it gains the permission for a controller (*i.e.*, $type = \mathbf{Grant}$ and $receiver = nil$) (T4–T5). When $type = \mathbf{Grant}$ and $receiver = nil$, process i enters C after setting $Receiver$ to i (T6). In contrast, if $pred \neq nil$, then process i repeatedly tests P until a message belonging to it is received (T8–T9). If the received message has value \mathbf{Grant} in the $Type$ component, then i enters C immediately; otherwise, if the message has value \mathbf{Info} in $Type$, then two cases may occur.

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

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

4.2.3 An Informal Correctness Argument

Since the correctness argument of the 2-bounded-bypass algorithm has shown the basic idea about arranging waiting processes into lists, this subsection simply provides a proof sketch. The notation and the definition of a controller are the same as those in Section 4.1.3.

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

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

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

Condition 2: $type = \mathbf{Grant}$ and $receiver = i$ hold. Informally, process i , which is a non-controller, obtains the permission to enter C .

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

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

Step E4 modifies P 's value to $(\mathbf{Info}, tail, nil, i)$. A value in P is said to be a *communication word* if $Type = \mathbf{Info}$. According to the algorithm (T10–T12), a communication word may enable the receiver of the word to satisfy Condition 3. If not, the receiver writes a new communication word to its predecessor and backs to step T7. Thus, step E4 enables at most one process to satisfy Condition 3.

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

Steps E5 and E7 set P to the initial value, (**Grant**, nil , Δ , Δ). Using the argument in Lemma 4.1, we have the counterpart of Lemma 4.1 below.

Lemma 4.5 *At any reachable system state,*

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

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

We now argue that the proposed algorithm satisfies the lockout-freedom condition. The argument is similar to that of Theorem 4.3. A process i is said to be in the doorway if $pc_i = T1$, and it is said to be in the waiting part if $pc_i \in \{T2, \dots, T12\}$. Due to the *fetch&store* primitive, each requesting process is properly arranged in a list after finishing its doorway. Moreover, each process has the identity of its predecessor, by which the head of a list initiates a communication phase to reverse the order in a list. Consequently, the permission can be conveyed according to the sequence of the requests; the algorithm thus satisfies not only the lockout-freedom condition, but also the FCFS condition.

4.3 An Impossibility Result

This section shows that the bounded-bypass mutual exclusion problem cannot be solved at all with fewer than two shared variables if only *fetch&store* and *read/write* are used. This result implies that both of the algorithms in this chapter are space-optimal. In the proof of this impossibility result, a shared variable associated with *fetch&store* and *read/write* is modelled as a type of historyless objects for two reasons. First, this approach simplifies the presentation of the proof. Second, a more general result is thus provided: using only historyless objects, two objects are

required to solve the bounded-bypass mutual exclusion problem. Historyless objects, as proposed by Fich *et al.* [22], are defined below, and then the proof is presented.

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

The proof can now be presented by following the proving strategy proposed by Burns and Lynch [11]. Two more definitions are needed. First, because a shared variable is modelled as an object, Definition 2.2 for the case in which $V = \mathcal{V}$ is rewritten as follows.

Definition 4.1 *System states s and t are indistinguishable to process i with respect to all objects, written as $s \stackrel{i}{\sim} t$, if the state of process i and the values of all the objects in the system are the same at s and t .*

The second definition generalizes that of Burns and Lynch [11], which states that a process *covers* a shared variable x provided that a *write* operation of the process is enabled to write to x . An enabled *write* operation can overwrite the variable involved. Similarly, an enabled nontrivial operation of a historyless object can also overwrite the object. Thus, the concept of “covering” is generalized to historyless objects.

Definition 4.2 *Process i covers a historyless object x at system state s provided that a nontrivial operation of i is enabled to manipulate x .*

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

A basic lemma showing that any process that reaches R from C on its own must take a nontrivial operation to some object is presented before proving the lower bound. The proof of this lemma is similar to the result provided by Lynch in [35, pp. 301–302] which shows that a process reaching C from R on its own must write something into shared memory before doing so.

Lemma 4.6 *Suppose that A is a mutual exclusion algorithm, shared by $n \geq 2$ processes, using only historyless objects. Let s be a reachable system state of A at which process i is in C . If process i reaches R in a finite execution fragment starting from s that involves steps of i only, then it must take a nontrivial operation to some object along the way.*

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

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

According to the progress condition, a finite execution fragment α_2 , starting from s_1 and not including any step of process i , exists such that some process reaches C . Because $s \stackrel{j}{\sim} s_1$ for every $j \neq i$, α_2 is also executable from s according to Lemma 2.1.

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

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

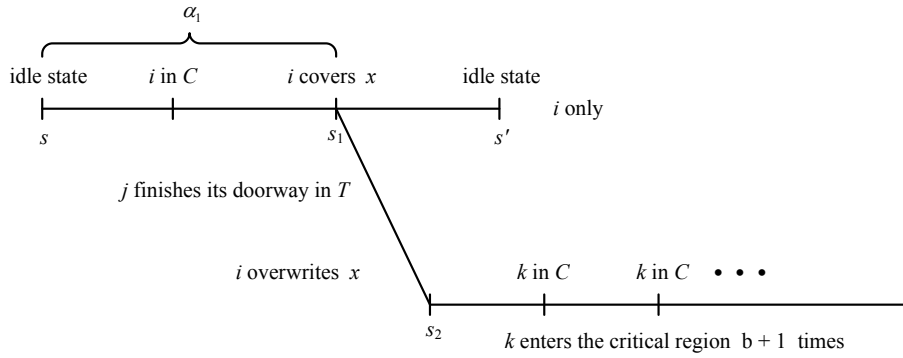


Figure 4.5: The execution for the proof of Theorem 4.7.

Theorem 4.7 *If algorithm A solves the bounded-bypass mutual exclusion problem for n processes where $n > 2$, using only historyless objects, then A must use at least two objects.*

Proof. Suppose for the sake of contradiction that there is such an algorithm A using only one historyless object, say x , and guaranteeing b -bounded bypass for some constant b . Let s be an initial system state. An execution of A that violates the bounded bypass condition is constructed below and is depicted in Fig. 4.5.

The progress condition implies that there is an execution involving process i only, starting from s , that causes process i to enter C once and back to an idle system state s' . Lemma 4.6 implies that process i must take a nontrivial operation to some object in E in this solo execution. Since only one object is used, process i must take a nontrivial operation to the historyless object x in E . Thus, i must cover x at some point in E .

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

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

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

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

4.4 Summary

Two fair and space-efficient algorithms are proposed for systems under time and memory constraints. The first algorithm is 2-bounded-bypass; the second is a FCFS algorithm based on the first algorithm. Each algorithm adopts the commonly available primitives *fetch&store* and *read/write*.

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

Furthermore, it is shown that any bounded-bypass algorithm using the same set of primitives must utilize at least two shared variables, regardless of the size of the

variables. This lower bound is proven by showing a more general result that two objects are necessary to solve the bounded-bypass mutual exclusion problem when using only historyless objects. Since shared variables associated with *fetch&store* and *read/write* belong to the class of historyless objects, the more general result applies to our model, implying that both of the algorithms in this chapter are optimal with respect to the number of shared variables. The proof technique is derived from that of Burns and Lynch [11].



Chapter 5

Tight Bound on RMR Time Complexity

This chapter establishes a tight bound of three on the RMR time complexity in DSM systems. We show that three is a lower bound on the RMR time complexity. The lower bound matches the upper bound of Huang’s algorithm in Section 3.3, it is therefore tight.

The computational model adopted is the DSM model with the general RMW primitive. To prove the lower bound, a definition on an indistinguishability relation is needed. Formal descriptions of the model and the indistinguishability relation appear in Chapter 2.

5.1 The General RMW Primitive

The general RMW primitive atomically accesses one shared variable, reading the value of the variable and writing back a new value according to the submitted function. Let V be the set of all possible values for the variable. The submitted function can be any function $f : V \rightarrow V$. Formally, the general RMW primitive is defined below, where v is the shared variable and f is the submitted function.

RMW (shared variable v , function f)

```

previous := v
v := f(v)
return previous

```

It is not hard to show that the RMW primitives used in Huang's algorithm are special cases of the general RMW primitive.

- Primitive *fetch&store*(*v*, *new*): It atomically writes value *new* to shared variable *v* and returns the old value, and is equivalent to $\text{RMW}(v, f)$ where *f* is a constant function that always maps to value *new*.
- Primitive *compare&swap*(*v*, *old*, *new*): It atomically writes value *new* to shared variable *v* exactly if its old value equals *old*, and returns the old value regardless of what happens in the comparison. The primitive is equivalent to $\text{RMW}(v, f)$ where *f* is a function defined as follows. Let *x* be any value in the value set of *v*.

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

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

5.2 An RMR Time Complexity Lower Bound

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

Theorem 5.1 *Let A be a mutual exclusion algorithm for $n > 3$ processes. Then the RMR time complexity of A must be three or more.*

This section is organized as follows. We first make a simplifying restriction on the mutual exclusion algorithms. Next, we present several properties of a process

that is busy waiting only at certain local shared variable(s) in T , *i.e.*, a process that is locally spinning in T . These properties will be used in our lower bound proof. Finally, we present the outline of the lower bound proof, and then show the detailed proof.

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

5.2.1 Basic Properties

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

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

Definition 5.1 *Let s be a system state of a mutual exclusion algorithm. We say that process i is locally spinning in T at s if*

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

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

Lemma 5.2 *Let s and t be system states of a mutual exclusion algorithm such that $s \stackrel{i}{\sim}_{V_i} t$ for process i . Then i is locally spinning in T at s if and only if i is locally spinning in T at t .*

Proof.

1. (\rightarrow) Suppose i is locally spinning in T at s . Since i is in T at s and $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$, i is in T at t . It remains to show that for any finite i -execution fragment α executable from t , α contains no RMR step and i remains in T from t to $\alpha(t)$. By way of contradiction, suppose that α is a finite i -execution fragment executable from t such that α contains an RMR step or i changes regions in α . Let α' be the prefix of α including and ending with the first step that is either an RMR step or an operation that makes i change regions. Since $s \stackrel{i}{\sim}_{\mathcal{V}_i} t$, α' is also executable from s . (If α' ends with an RMR step, this follows from Corollary 2.3; otherwise, this follows from Lemma 2.2.) This contradicts the assumption that i is locally spinning in T at s .
2. (\leftarrow) The other direction follows from symmetry.

□

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

Lemma 5.3 *Let s be a system state of a mutual exclusion algorithm at which process i is locally spinning in T . In any execution fragment executable from s , no RMR step from i exists before the first RMR step to i occurs.*

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

Let α' be the prefix of α including and ending with the first RMR step from i . Note that the assumption on α implies that α' contains no RMR step to i . We show that $\alpha'|i$ is also executable from s . This is the needed contradiction because $\alpha'|i$ ends with an RMR step from i . By the definition of α' , it is executable from s , ends

with an RMR step from i , and contains neither RMR steps from i nor RMR steps to i except the last step. Also, it is trivial that $s \stackrel{i}{\underset{\nu_i}{\sim}} s$. By Corollary 2.3, $\alpha|i$ is also executable from s . \square

Intuitively, if a process i is locally spinning in T at some point and i enters C at a later point, then some other process must have taken at least one RMR step to wake up i . The next lemma, also called the inherent cost lemma, formalizes this intuition. A similar observation in a message-passing model can be found in Chandy and Misra’s work about “knowledge” among processes [12].

Lemma 5.4 (inherent cost) *Let s be a system state of a mutual exclusion algorithm at which process i is locally spinning in T . Suppose that process i reaches C in a finite execution fragment α executable from s . Then, α must contain at least one RMR step to i .*

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

By Lemma 5.3, α contains no RMR step from i and therefore it contains neither RMR steps from i nor RMR steps to i . In addition, it is clear that $s \stackrel{i}{\underset{\nu_i}{\sim}} s$. Thus, by Lemma 2.2, $\alpha|i$ is also executable from s and process i has the same state at $\alpha(s)$ and $(\alpha|i)(s)$. Since i is in C at $\alpha(s)$, i is also in C at $(\alpha|i)(s)$. Thus, $\alpha|i$ is the needed execution fragment because i changes regions in $\alpha|i$. \square

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

Lemma 5.5 *Let A be a local-spin mutual exclusion algorithm for $n > 1$ processes. Let s be a reachable system state of A at which process i is in R and some other process is in C . Then there exists a finite i -execution fragment α executable from s such that i is locally spinning in T at system state $\alpha(s)$.*

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

5.2.2 Proof Outline

Throughout the rest of this paper, we let $A = (\mathcal{P}, \mathcal{V}, \delta)$ be an arbitrary local-spin algorithm for $n > 3$ processes and let s_{init} be an initial system state of A . To prove the lower bound of three on the RMR time complexity, the objective is to construct an execution of A from s_{init} in which some process takes at least three RMR steps to enter and exit C once. We call such an execution a *goal execution*.

A goal execution will be constructed in the following way. We start by constructing n solo executions, one per process, each starting from s_{init} and involving its steps only until it has just entered C . (The progress condition implies that this is possible.) For each $i \in \mathcal{P}$, let α_i denote the solo execution of i . Next, for each α_i and each process $j \neq i$, we extend α_i to what is denoted by α_{ij} by running j alone until j has just entered a system state at which j is locally spinning in T . Execution α_{ij} exists according to Lemma 5.5. The lower bound proof focuses on the set of all α_{ij} 's, called set \mathcal{E} . More precisely, we define

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

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

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

Case 1. $\exists \alpha_{ij} \in \mathcal{E} : time(i, \alpha_{ij}) \geq 2$ or $time(j, \alpha_{ij}) \geq 2$.

Let $\alpha_{ij} \in \mathcal{E}$ be such an execution. A goal execution can be extended from α_{ij} by applying the inherent cost lemma.

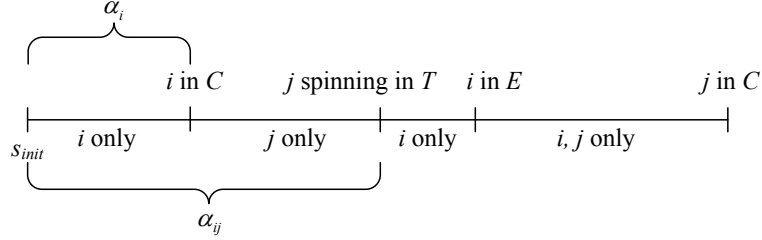


Figure 5.1: A goal execution extended from α_{ij} in which $time(i, \alpha_{ij}) \geq 2$.

If $time(i, \alpha_{ij}) \geq 2$, we extend α_{ij} to an execution in which i takes at least one RMR step in its corresponding exit region so that i takes at least three RMR steps in total. The way to extend α_{ij} is described as follows and is illustrated in Fig. 5.1. From the end of α_{ij} , we let i leave C first and then let i and j take enabled steps alternately until j enters C . Since i and j take enabled steps alternately, this execution is admissible. Thus, the progress condition implies that j eventually enters C . By the inherent cost lemma, there exists at least one RMR step to j , which must be taken by i because only processes i and j are involved, in the portion of the resulting execution after α_{ij} .

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

Case 2. $\forall \alpha_{ij} \in \mathcal{E} : time(i, \alpha_{ij}) < 2$ and $time(j, \alpha_{ij}) < 2$.

This case is the core of the lower bound proof. We construct a goal execution in which some process takes one RMR step in T and takes at least two RMR steps in its corresponding exit region.

We first use the following property, called the **rendezvous property**, which says that in most executions in \mathcal{E} , processes communicate through the same

remote shared variable: (The property will be proved in the next subsection.)

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

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

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

Take any three distinct processes i, j and k that are different from m . Processes i, j and k exist since $n > 3$. Consider α_{ij} and α_{ik} . By the conclusion above, i and j each take exactly one RMR step in α_{ij} , and they access the shared variable v , which is located at process m . Likewise, i and k each take exactly one RMR step in α_{ik} , and they access v . A goal execution can be constructed by extending either α_{ij} or α_{ik} , in which i takes at least two more RMR steps in addition to the one it has taken in α_{ij} or α_{ik} . The construction is illustrated in Fig. 5.2.

First, we extend α_{ij} to α'_{ij} in the same way as that shown in Fig. 5.1, *i.e.*, by letting i leave C and then running i and j until j reaches C . In the suffix of α'_{ij} after α_{ij} , if i takes at least two RMR steps, α'_{ij} is already a goal execution. Otherwise, the inherent cost lemma implies that i takes exactly one RMR step, which is from i to j . Based on this implication, a goal execution α'_{ik} is constructed below.

Execution α'_{ik} begins with α_{ik} , in which i has taken one RMR step. It then continues by letting process i run alone until it takes the RMR step to j

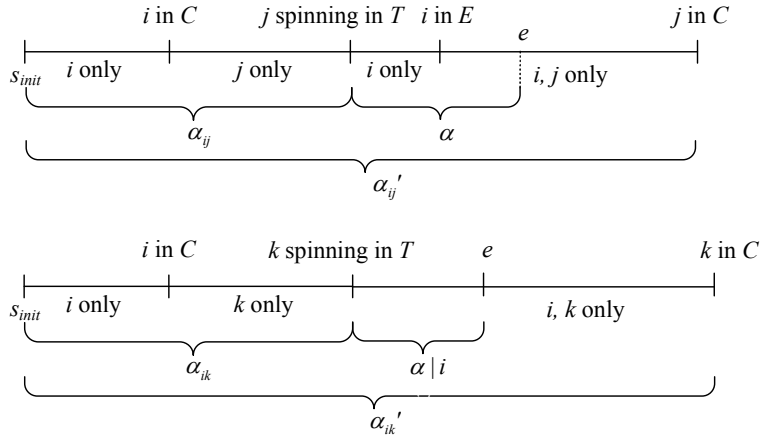


Figure 5.2: A goal execution extended from either α_{ij} or α_{ik} . We write e to denote the RMR step from i to j .

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

5.2.3 Detailed Proof

We begin by proving the rendezvous property (Lemma 5.8) and then provide the detailed lower bound proof.

In order to prove the rendezvous property, we first present two lemmas for all α_i of A , $i \in \mathcal{P}$. The first, Lemma 5.6, says that for any two distinct solo executions α_i and α_j , there exists at least one shared variable that is accessed in both α_i and α_j . That is, processes i and j must access at least one common shared variable in their respective solo executions. The other, Lemma 5.7, says that if every $i \in \mathcal{P}$ accesses at most one remote shared variable in its α_i , then there is exactly one shared variable, say v , that is accessed in all α_i , $i \in \mathcal{P}$. That is, every process i must access v in its α_i . Note that unlike Lemma 5.7, Lemma 5.6 holds without any assumption

on the number of remote shared variables accessed in each α_i .

For presenting Lemma 5.6 and Lemma 5.7, we need a definition: for every shared variable v , let \mathcal{P}_v denote the set of all processes that access v in their respective solo executions. That is, for every v in \mathcal{V} , define

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

First, we prove Lemma 5.6, also called the pairwise common lemma. Informally, although α_i and α_j are two independent executions, processes i and j should access at least one common shared variable for synchronization purposes. For otherwise, it is easy to yield an execution in which both i and j are in their critical regions simultaneously by concatenating α_i and α_j . This violates the mutual exclusion condition.

Lemma 5.6 (pairwise common) *For any two solo executions α_i and α_j , $i \neq j$, there exists at least one shared variable accessed in both α_i and α_j . More precisely, $\forall i, j \in \mathcal{P}, i \neq j, \exists v \in \mathcal{V} : \{i, j\} \subseteq \mathcal{P}_v$.*

Proof. By way of contradiction, suppose that there exists no shared variable accessed in both α_i and α_j . Thus, each shared variable accessed in α_j has the same value at system states s_{init} and $\alpha_i(s_{init})$. In addition, process j has the same state at s_{init} and $\alpha_i(s_{init})$, and therefore we have $s_{init} \stackrel{P}{\sim}_V \alpha_i(s_{init})$, where $P = Pro(\alpha_j) = \{j\}$ and $V = Var(\alpha_j)$. Hence, according to Lemma 2.1, α_j is also executable from $\alpha_i(s_{init})$. This violates the mutual exclusion condition because both i and j are in C at $(\alpha_i \circ \alpha_j)(s_{init})$. \square

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

Next, we prove Lemma 5.7. Suppose that every process $i \in \mathcal{P}$ accesses at most one remote shared variable in α_i . (Note that i may access many local shared variables in α_i .) Based on the pairwise common lemma, we show that all processes must access one common shared variable, say v , in their respective solo executions.

This implies that for all $i \in \mathcal{P}$, except the process to which v is local, i accesses exactly one remote shared variable in α_i and this shared variable is v .

Lemma 5.7 *Suppose that for all $\alpha_i, i \in \mathcal{P}$, i accesses at most one remote shared variable in α_i . Then there exists exactly one shared variable v such that for all $\alpha_i, i \in \mathcal{P}$, i accesses v in α_i . More precisely, there exists exactly one shared variable v such that $|\mathcal{P}_v| = n$.*

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

Claim 5.7.1 *Suppose that for all $\alpha_i, i \in \mathcal{P}$, i accesses at most one remote shared variable in α_i . Then, $\exists v \in \mathcal{V} : |\mathcal{P}_v| > 2$.*

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

Consider four distinct processes i, j, k and l . (Processes i, j, k and l exist because $n > 3$.) By the pairwise common lemma, for α_i and α_j , there exists one shared variable accessed in both α_i and α_j . Let variable w be such a variable, *i.e.*, $\{i, j\} \subseteq \mathcal{P}_w$. Since $|\mathcal{P}_v| \leq 2$ for all $v \in \mathcal{V}$, $\mathcal{P}_w = \{i, j\}$. Likewise, we conclude that $\mathcal{P}_x = \{j, k\}$ for some variable x , $\mathcal{P}_y = \{k, l\}$ for some variable y and $\mathcal{P}_z = \{j, l\}$ for some variable z . Since $\mathcal{P}_w = \{i, j\} \neq \mathcal{P}_x = \{j, k\}$, w and x must be two different shared variables. Similarly, we conclude that w, x, y and z are four different shared variables. (See Fig. 5.3.)

Since a shared variable is local to only one process, variable w is remote to at least one of processes i and j . Assume, without loss of generality, w is

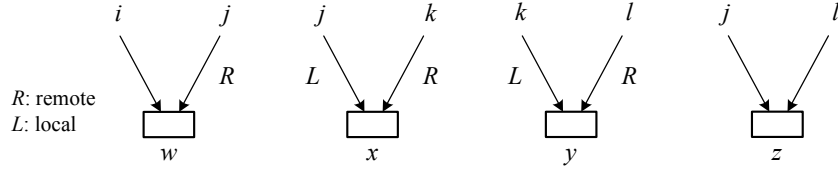


Figure 5.3: Shared variables for the proof of Claim 5.7.1.

remote to j . Since j accesses at most one remote shared variable in α_j and it has accessed w , variable x must be local to j and therefore x is remote to process k . Similarly, y is remote to process l . Hence, we know that j accesses remote shared variable w in α_j and l accesses remote shared variable y in α_l . However, variable z , which is accessed in both α_j and α_l , is remote to at least one of j and l . Thus, at least one of j and l accesses more than one remote shared variable, which is the needed contradiction. \square

Next, we prove that $\exists v \in \mathcal{V} : |\mathcal{P}_v| = n$. Again, by way of contradiction, suppose that $|\mathcal{P}_v| < n$ for all $v \in \mathcal{V}$. We will show that some process accesses more than one remote shared variable, which contradicts the assumption that each process accesses at most one. By Claim 5.7.1, we conclude that there exists one shared variable v such that $n > |\mathcal{P}_v| > 2$. Let variable w be such a shared variable. Since $n > |\mathcal{P}_w| > 2$, assume $\{i, j, k\} \subseteq \mathcal{P}_w$ and $\{l\} \not\subseteq \mathcal{P}_w$. For processes i, j and k , variable w is remote to at least two of them. Without loss of generality, assume w is remote to j and k . Namely, j and k each access remote shared variable w in α_j and α_k .

We now show that some process accesses more than one remote shared variable. Consider α_j and α_l . By the pairwise common lemma, there exists a variable accessed in both α_j and α_l . Let variable x be such a variable, *i.e.*, $\{j, l\} \subseteq \mathcal{P}_x$. Similarly, for α_k and α_l , let variable y be a variable that $\{k, l\} \subseteq \mathcal{P}_y$. Clearly, both x and y are variables different from variable w since $\{l\} \subseteq \mathcal{P}_x$, $\{l\} \subseteq \mathcal{P}_y$, but $\{l\} \not\subseteq \mathcal{P}_w$.

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

Otherwise, if x and y are two different shared variables, since j has accessed

remote shared variable w , variable x is local to j and therefore x is remote to l . Thus, for processes k and l , we know that k accesses remote shared variable w in α_k , and l accesses remote shared variable x in α_l . However, because y , which is accessed in both α_k and α_l , is remote to at least one of k and l , at least one of k and l accesses more than one remote shared variable. \square

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

Lemma 5.8 (rendezvous property) *Suppose that for all $\alpha_{ij} \in \mathcal{E}$, processes i and j each access at most one remote shared variable in α_{ij} . Then there exists a shared variable v such that for all $\alpha_{ij} \in \mathcal{E}$ that v is remote to both i and j , both i and j must access v in α_{ij} . More precisely,*

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

Proof. Since for all $\alpha_{ij} \in \mathcal{E}$, processes i and j each access at most one remote shared variable in α_{ij} , it is true, *a fortiori*, that for all $\alpha_i, i \in \mathcal{P}$, process i accesses at most one remote shared variable in α_i . From Lemma 5.7, there exists exactly one common shared variable, say v , that is accessed in all $\alpha_i, i \in \mathcal{P}$. Let m be the process to which v is local. We prove that j also accesses v in every $\alpha_{ij} \in \mathcal{E}$ with $i \neq m$ and $j \neq m$, which completes the proof because v is the needed shared variable.

We first show the following claim.

Claim 5.8.1 *In every $\alpha_{ij} \in \mathcal{E}$ with $i \neq m$ and $j \neq m$, process j must access some shared variable that has been accessed by i .*

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

Let α be the subsequence of α_{ij} containing all steps executed by j , that is, the suffix of α_{ij} after α_i ($\alpha_{ij} = \alpha_i \circ \alpha$). We will show that α is executable from s_{init} and $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha(s_{init})$. By Lemma 5.2, and because $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha(s_{init})$

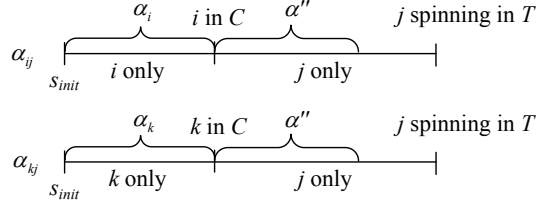


Figure 5.4: Executions α_{ij} and α_{kj} . Execution fragment α'' ends with the first RMR step from j to i .

and j is locally spinning in T at $\alpha_{ij}(s_{init})$, this implies that j is also locally spinning in T at $\alpha(s_{init})$. But this easily yields a j -execution α' violating the progress condition. Starting from s_{init} , execution α' begins with α . It then continues by running j alone. Since j is locally spinning in T at $\alpha(s_{init})$, no finite j -execution fragment executable from $\alpha(s_{init})$ will lead j to C . This violates the progress condition.

It remains only to show that α is executable from s_{init} and $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha(s_{init})$. Since j does not access any shared variable that has been accessed by i , we have $\alpha_i(s_{init}) \stackrel{j}{\sim}_V s_{init}$ where $V = Var(\alpha)$. By the definition of α , $Pro(\alpha) = \{j\}$ and α is executable from $\alpha_i(s_{init})$. Thus, by Lemma 2.1, α is also executable from s_{init} and $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_V \alpha(s_{init})$. In addition, since v is the only remote shared variable accessed by i in α_{ij} , i does not access any shared variable located at j . We now show that $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha(s_{init})$ holds. Let w be any variable in \mathcal{V}_j . If w is in V , it has the same value at $\alpha_{ij}(s_{init})$ and $\alpha(s_{init})$ because we have proved that $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_V \alpha(s_{init})$. Otherwise, if w is not accessed by j in α , because i does not access any shared variable located at j , the value of w is never changed in α_{ij} and α . Hence, $\alpha_{ij}(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha(s_{init})$. \square

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

In α_{ij} , the possible shared variables accessed by i are v and the shared variables located at i . By Claim 5.8.1, since j does not access v , j must access some shared

variable located at i . Since j accesses at most one remote shared variable in α_{ij} , j must access exactly one remote shared variable and this shared variable is located at i .

Consider another α_k , $k \neq m, i, j$. We show that in α_{kj} , j does not access any shared variable that has been accessed by k in α_{kj} , contradicting Claim 5.8.1. As shown in Fig. 5.4, let α'' be the subsequence of α_{ij} starting from the end of α_i (not including the end of α_i) until j has just finished its first RMR step, which is from j to i . In α_i and α_k , since i and k do not access any shared variable located at process j (variable v , which is located at m , is the only remote shared variable accessed by i and k), we have $\alpha_i(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha_k(s_{init})$. Since $\alpha_i(s_{init}) \stackrel{j}{\sim}_{\mathcal{V}_j} \alpha_k(s_{init})$, j enables the same step at $\alpha_i(s_{init})$ and $\alpha_k(s_{init})$ by the determinism and the localized enabling assumptions of the model. Furthermore, if the step is not remote, the resulting system states are also indistinguishable to j with respect to \mathcal{V}_j by the localized update assumption. Using such an argument repeatedly, it is easy to see that j also performs α'' in α_{kj} after α_k as it does in α_{ij} . Thus, j also accesses a shared variable located at i in α_{kj} .

Since process j accesses at most one remote shared variable in α_{kj} by the assumption on every execution in \mathcal{E} , j accesses exactly one remote shared variable and this shared variable is located at i . Therefore, j does not access any shared variable that has been accessed by k in α_{kj} . (Note that the possible shared variables accessed by k in α_{kj} are v and the shared variables located at k .) This contradicts Claim 5.8.1. \square

The main lemma, rendezvous property, has been proven. To finish the lower bound proof, it remains to provide the details that are skipped in the proof outline in Section 5.2.2.

Proof (of Theorem 5.1). We show that there exists an execution of A in which some process performs at least three RMR steps to enter and exit C once. We complete the proof with a case analysis on \mathcal{E} , getting a goal execution for each possibility.

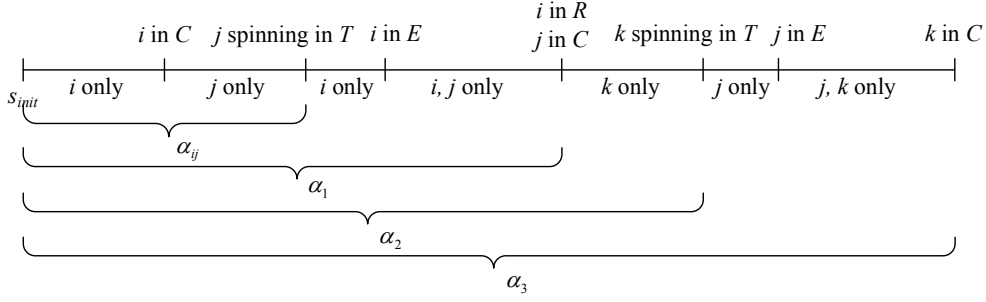


Figure 5.5: Executions in Case 1.

Case 1. $\exists \alpha_{ij} \in \mathcal{E} : \text{time}(i, \alpha_{ij}) \geq 2$ or $\text{time}(j, \alpha_{ij}) \geq 2$.

Let $\alpha_{ij} \in \mathcal{E}$ be such an execution.

If $\text{time}(i, \alpha_{ij}) \geq 2$, we have presented the construction of a goal execution in Section 5.2.2. If $\text{time}(i, \alpha_{ij}) \geq 2$ does not hold, it must be the case that $\text{time}(j, \alpha_{ij}) \geq 2$ holds. It remains to construct a goal execution in this case. We extend α_{ij} to an execution in which j must take at least one RMR step in E .

As shown in Fig. 5.5, we extend α_{ij} , in which j has taken at least two RMR steps, to α_1 by letting i leave C first and then alternately executing enabled steps of i and j until i enters R and j enters C . This follows from the progress condition. Then we extend α_1 to α_2 by running a new competing process k alone until k is locally spinning in T . This follows from Lemma 5.5. Finally, we extend α_2 to α_3 by letting j leave C first and then alternately executing enabled steps of j and k until k enters C ; along the way, by the inherent cost lemma, process j must take at least one RMR step to k . In total, j takes at least three RMR steps to enter and exit C once in α_3 .

Case 2. $\forall \alpha_{ij} \in \mathcal{E} : \text{time}(i, \alpha_{ij}) < 2$ and $\text{time}(j, \alpha_{ij}) < 2$.

By the rendezvous property, there exists a shared variable v such that for any distinct processes i and j to which v is remote, both i and j must access v in α_{ij} . Let m be the process to which v is local. Take any three distinct processes

i , j and k that are different from m . As shown in Section 5.2.2, we first extend α_{ij} to α'_{ij} by letting i leave C and then running i and j until j reaches C . If process i takes at least two RMR steps in the portion of α'_{ij} after α_{ij} , execution α'_{ij} is already a goal execution. Otherwise, the inherent cost lemma implies that i takes exactly one RMR step, which is from i to j . We now construct a goal execution α'_{ik} . Let α be the subsequence of α'_{ij} starting from the end of α_{ij} (not including the end of α_{ij}) until i has just finished its RMR step, say step e . The precise construction of α'_{ik} is given below.

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

It remains to show that it is legitimate in our construction to concatenate α_{ik} by $\alpha|i$. This follows from Corollary 2.3. To apply the corollary, we need to show the following properties: $\alpha_{ij}(s_{init}) \stackrel{i}{\sim}_{\mathcal{V}_i} \alpha_{ik}(s_{init})$; α is executable from $\alpha_{ij}(s_{init})$ and it ends with an RMR step from i ; and α contains neither RMR steps from i nor RMR steps to i except the last step. In α_{ij} and α_{ik} , since i performs the same sequence of steps (*i.e.*, α_i), and j and k do not access any shared variable located at i (v , which is located at m , is the only remote shared variable accessed by j and k), we have $\alpha_{ij}(s_{init}) \stackrel{i}{\sim}_{\mathcal{V}_i} \alpha_{ik}(s_{init})$. By the definition of α , it is executable from $\alpha_{ij}(s_{init})$ and it ends with an RMR step from i . Since e is the only RMR step from i in the portion of α'_{ij} after α_{ij} , α contains no RMR step from i except the last one. In addition, by Lemma 5.3, α contains no RMR step from j and, *a fortiori*, α contains no RMR step from j to i . Thus, α , which is an $\{i, j\}$ -execution fragment, contains neither RMR steps from i nor RMR steps to i except the last step. Thus, by Corollary 2.3, $\alpha|i$ is executable from $\alpha_{ik}(s_{init})$.

□

5.3 Summary

We have proved that the remote reference time complexity of any mutual exclusion algorithm with at least four processes is at least three in DSM systems. Due to Huang’s algorithm, the bound is tight.

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



Chapter 6

Conclusions and Future Work

In this chapter, we draw conclusions and discuss directions for future research.

6.1 Tight Bound on Space Complexity

In Chapter 4, we propose two fair and space-efficient algorithms for systems under time and memory constraints. The first algorithm is 2-bounded-bypass; the second is a FCFS algorithm based on the first algorithm. Each algorithm adopts the commonly available primitives *fetch&store* and *read/write*, and employs only constant two shared variables. Furthermore, we show that with the same set of primitives, two shared variables are necessary to solve the bounded-bypass mutual exclusion problem. Both of the algorithms are therefore optimal in terms of the number of required shared variables.

One disadvantage of the algorithms is that the hot spot contention [19] can be up to n . The hot spot contention is the maximum number of pending operations for any individual shared variable in any execution, and this number is one of the principal determiners of the system performance. Because each algorithm utilizes only a constant number of shared variables to meet the memory constrain, $\Omega(n)$ hot spot contention is inevitable.

Additionally, each competing process in the algorithms repeatedly tests a shared variable while it is waiting to enter its critical region. As described in Section 1.2,

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

6.2 Tight Bound on RMR Time Complexity

In Chapter 5, we establish the tight bound of three on the RMR time complexity and show that it remains unchanged when we consider lockout-freedom and bounded bypass. The lower bound is proved by constructing an execution in which some process takes at least three remote memory references to enter and exit its critical region once. In the course of proving the lower bound, we need to formalize the notion of a process “entering a local-spin loop.” Danek and Hadzilacos [16] and we [13] independently proposed a similar formal definition at about the same time. Based on the definition, we also present several properties of local-spin mutual exclusion algorithms.

6.3 Future Work

In this section, we discuss some of the remaining open problems and directions for further research on the mutual exclusion problem.

Only *fetch&store* is investigated in Chapter 4. Herlihy [27] has provided a wait-

free hierarchy that classifies synchronization primitives according to their power to solve consensus. A future direction is to provide a separation among all multi-processor synchronization primitives based on the space complexity of solutions to the mutual exclusion problem. Interestingly, *compare&swap*, which is considered to be powerful according to Herlihy's wait-free hierarchy, is not a good choice to decrease the space requirement. A recent paper of Fich et al. [21] shows that at least n shared variables are required to solve the n -process mutual exclusion problem if only conditional primitives, such as *compare&swap*, are available. Their result indicates an inherent difference between the two problems.

The algorithms proposed in Chapter 4 are optimal with respect to the number of shared variables. Future work is needed to establish the tight bound on the size of shared variables. Although each of the algorithms utilizes two shared variables, the total values taken on by the shared variables in the FCFS algorithm is larger than that in the 2-bounded-bypass algorithm. An interesting question is how large the two shared variables must be in order to guarantee bounded bypass or FCFS.

One disadvantage of Huang's algorithm is that it uses two primitives, *compare&swap* and *fetch&store*, besides *read/write*. Since Cypher [15] showed that there is no constant time algorithm using conditional RMW primitives and *read/write*, a non-conditional RMW primitive is needed to implement an algorithm whose upper bound matches the lower bound in Chapter 5. An open question is whether such an algorithm is obtainable using only one non-conditional RMW primitive such as *fetch&store* in addition to *read/write*.

In addition, although Huang's algorithm satisfies lockout-freedom and bounded bypass, it does not satisfy the FCFS property. Hence, the tight bound on the RMR time complexity for the FCFS mutual exclusion problem remains to be solved. The tight bound must be either three or four, because the MCS lock [37] satisfies the FCFS property and its RMR time complexity is four, and our lower bound of three also holds for the problem.

We focus only on DSM systems when studying on the RMR time complexity of the mutual exclusion problem. The lower bound proof herein is not applicable

to CC systems. Future work is needed to establish the exact lower bound in CC systems.



Bibliography

- [1] *ARM1136JF-S and ARM1136J-S Technical Reference Manual*. ARM Inc., 2005. Available at http://www.arm.com/documentation/ARMProcessor_Cores/.
- [2] J. H. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2002.
- [3] J. H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12, July 2002.
- [4] J. H. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and- ϕ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 538–547, May 2003.
- [5] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, September 2003.
- [6] J. H. Anderson and M. Moir. Using local-spin k -exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.
- [7] J. H. Anderson and J.-H. Yang. Time/contention trade-offs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

- [8] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
- [9] J. E. Burns. Mutual exclusion with linear waiting using binary shared variables. *ACM SIGACT News*, 10(2):42–47, 1978.
- [10] 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.
- [11] J. E. Burns and N. A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.
- [12] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.
- [13] S.-H. Chen and T.-L. Huang. A tight bound on time complexity of mutual exclusion. In *Proceedings of the International Computer Symposium*, pages 1352–1357, Taipei, Taiwan, December 2004.
- [14] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 148–157, December 1993.
- [15] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.
- [16] R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *Proceedings of the 18th International Symposium on Distributed Computing*, October 2004.
- [17] N. G. deBruijn. Additional comments on a problem in concurrent programming control. *Communication of the ACM*, 10(3):137–138, 1967.

- [18] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
- [19] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.
- [20] M. A. Eisenberg and M. R. McGuire. Further comments on dijkstra’s concurrent programming control problem. *Communication of the ACM*, 15(11):999, 1972.
- [21] F. Fich, D. Hendler, and N. Shavit. On the inherent weakness of conditional primitives. *Distributed Computing*, 18(4):267–277, March 2006.
- [22] F. Fich, M. Herlihy, and N. Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.
- [23] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, January 1989.
- [24] S. S. Fu and N.-F. Tzeng. A circular list-based mutual exclusion scheme for large shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):628–639, June 1997.
- [25] J. G. Ganssle. *The Art of Programming Embedded Systems*. Academic Press, 1991.
- [26] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.
- [27] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [28] 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.

- [29] 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, December 1994.
- [30] 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.
- [31] 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.
- [32] Y.-J. Kim and J. H. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15, October 2001.
- [33] D. E. Knuth. Additional comments on a problem in concurrent programming control. *Communication of the ACM*, 9(5):321–322, 1966.
- [34] L. Lamport. A new solution of Dijkstra’s concurrent programming problem. *Communication of the ACM*, 17(8):453–455, August 1974.
- [35] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
- [36] P. Magnusson, A. Landin, and E. Hagersten. Oueue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE, April 1994.
- [37] 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.
- [38] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

- [39] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.
- [40] I. Rhee. Optimizing a FIFO, scalable spin lock using consistent memory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 106–114, December 1996.
- [41] J.-H Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.
- [42] K. M. Zuberi and K. G. Shin. An efficient semaphore implementation scheme for small-memory embedded systems. In *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*, pages 25–34, June 1997.
- [43] K. M. Zuberi and K. G. Shin. EMERALDS: a small-memory real-time micro-kernel. *IEEE Transactions on Software Engineering*, 27(10):909–927, October 2001.

