GROUP MUTUAL EXCLUSION IN LINEAR TIME AND SPACE

Yuan He

Department of Computer Science

East Carolina University

July 2014

Director of Thesis: Dr. Krishnan Gopalakrishnan

The Group Mutual Exclusion (GME) problem, introduced by Joung, is a natural extension of the classical Mutual Exclusion problem. In the classical Mutual Exclusion problem, two or more processes are not simultaneously allowed to be in their CRITICAL SECTION, a piece of code where a common resource is accessed. In the GME problem, it is necessary to impose mutual exclusion on different groups of processes in accessing a resource, while allowing processes of the same group to share the resource. The Group Mutual Exclusion problem arises in several applications and is the focus of this thesis.

We present an algorithm for the GME problem that satisfies the properties of *Mutual Exclusion*, *Starvation Freedom*, *Bounded Exit*, *Concurrent Entry* and *First-Come-First-Served*. Our algorithm has $\Theta(N)$ shared space complexity and $O(N)$ RMR (Remote Memory Reference) complexity. Our algorithm is developed by generalizing the well-known Lamport's Bakery Algorithm for the classical mutual exclusion problem, while preserving its simplicity and elegance. Just like Lamport's Bakery Algorithm, our algorithm has the disadvantage that the token numbers can grow in an unbounded manner.

When all shared variables are required to be of bounded size, Hadzilacos presented an algorithm, whose shared space complexity is $\Theta(N^2)$ and whose RMR complexity is claimed to be $O(N)$. Hadzilacos posed as an open problem, the development of a linear time and space algorithm that uses only bounded shared variables and only simple read and write instructions. As a solution to the open problem, Jayanti et al. presented a space efficient adaptation of the above algorithm that uses only $\Theta(N)$ shared space and inherited the claim that the RMR complexity is $O(N)$. We show that both of these algorithms are of RMR complexity $\Omega(N^2)$ and thus demonstrate that both claims are erroneous. So, the open problem posed by Hadzilacos is still open.

GROUP MUTUAL EXCLUSION IN LINEAR TIME AND SPACE

A Dissertation

Presented to the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Yuan He

July 2014

GROUP MUTUAL EXCLUSION IN LINEAR TIME AND SPACE

By

Yuan He

APPROVED BY:

DIRECTOR OF THESIS:	______________________ Dr.Krishnan Gopalakrishnan
COMMITTEE MEMBER:	______________________ Dr. Karl Abrahamson
COMMITTEE MEMBER:	______________________ Dr. Junhua Ding
CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:	______________________ Dr. Karl Abrahamson
DEEN OF THE GRADUATE SCHOOL:	______________________ Dr. Paul J. Gemperline

Acknowledgments

I am indebted to many people who first helped me to get into East Carolina University, and then helped me to get out again. The first and the most important of all, the best part of my student life in ECU is to have Krishnan Gopalakrishnan as my advisor. He shows me how attractive the computer science research is, and how wonderful an academic life will be. It wouldn't be possible for me to finish this without his help.

I would like to express my sincerest appreciation to Junhua Ding, for receiving guidance and advices about research. I am also grateful to Karl Abrahamson, for the time and energy he spent on my dissertation and thesis defense. I would also like to express my thanks and appreciation to faculties in the Computer Science department, Qing Ding, Masao Kishore, Nasseh Tabrizi and Ronnie Smith. Many thanks also to my roommate Hui Guo, for helping me a lot in past two years.

Foremost, I thank my parents for their unwavering support and encouragement, both during graduate school and the years before.

Finally, I want to thank Lei Wang, for all the love and support, for her patience. I could not have finished this without you.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The Mutual Exclusion Problem

Mutual Exclusion problem is a classical problem in computer science introduced by Dijkstra in 1965 [7]. The problem arose in the context of contention for use of shared resources by concurrent processes. For example, several processes may want to access a printer or a disk driver. It would be a complete disaster if multiple processes use the printer or a disk simultaneously.

To formalize the problem, we assume all running processes repeatedly cycle through four sections of code viz. REMAINDER SECTION, ENTRY SECTION, CRITICAL SECTION (CS) and EXIT SECTION, in that order (Figure 1.1).

If a process does not request accessing the shard resource, it is in the REMAINDER SECTION. A process accesses the shared resources only when in the CRITICAL SECTION code. The ENTRY SECTION consists the code between the REMAINDERS SECTION execution and subsequent CRITICAL SECTION execution, and the EXIT SECTION generates the code between a CRITICAL SECTION execution and the subsequent REMAINDER SECTION execution.

Solving Mutual Exclusion problem consists of designing code for the ENTRY SECTION and the EXIT SECTION such that following properties are satisfied:

Mutual Exclusion (ME): No two processes can be in the CRITICAL SECTION at the same time.

Starvation Freedom (SF): If no process stays in the CRITICAL SECTION forever, then any process that enters the ENTRY SECTION eventually enters the CRITICAL SECTION.

Figure 1.1 Process Structure in Mutual Exclusion Problem

```
repeat
    REMAINDER SECTION
    ENTRY SECTION
    CRITICAL SECTION (CS)
    EXIT SECTION
forever
```

A mutual exclusion algorithm guarantees that processes requesting the CS eventually enter the CS. However, if multiple processes request to enter the CS at the same time, the order that processes enter the CS is undefined. Consider, for instance, a process i requests to enter the CS and stays in the ENTRY SECTION. After a long time, a process j requests and enters the CS while process i is still in the ENTRY SECTION. To ensure fairness in this case, a mutual exclusion algorithm is often required to satisfy the property *First-Come-First-Served (FCFS)*. The idea is to make processes enter the CS in the same order in which their requests are made. Lamport [16] originally formalized the FCFS property. We split the ENTRY SECTION into two parts: a *doorway* part and a following *waiting room* part. The doorway is a code part that the process can finish within a bounded number of its own steps. The waiting room contains the part that the process is busy-waiting on. Once a process goes through the waiting room, it enters the CS. Thus, a mutual exclusion algorithm is fair if it satisfies the following property:

First-Come-First-Served (FCFS): If process i finished the doorway before process j starts the doorway, process j does not enter the CRITICAL SECTION before process i.

1.2 The Group Mutual Exclusion Problem

Mutual exclusion guarantees competing processes access a shared resource exclusively. However, in applications such as Computer Supported Cooperative Work (CSCW), it is necessary to keep mutual exclusion while still having some degree of concurrency. An example named "Congenial Taking Philosophers" is introduced by Joung [12]. Consider a set of n philosophers who want to think alone and talk in forum. Suppose that there is only one meeting room for talk (CRITICAL SECTION), a philosopher can attend a forum only if the meeting room is empty or another philosopher interested in the same forum is already in the meeting room (concurrency). The difficulty of solving the problem is to ensure that philosophers who want to attend the same forum can enter the meeting room simultaneously, while philosophers requesting different forum do not stay in the meeting room at the same time.

Obviously, a mutual exclusion algorithm can not provide the concurrency while preserving the mutual exclusion property. In the mutual exclusion problem, no two philosophers are allowed in the meeting room at any point of time. This leads us to the following generalization of the mutual exclusion problem.

Group Mutual Exclusion (GME) problem, first introduced and defined by Joung [12] in 1998 , is a natural generalization of the classical mutual exclusion problem. As before, we can think of the processes repeatedly cycling through four sections of code viz., REMAINDER SECTION, ENTRY SECTION, CRITICAL SECTION (CS) and EXIT SECTION, in that order. In GME, when a process leaves the REMAINDER SECTION, it requests a “session”, and wants to enter into the CS. Unlike the mutual exclusion problem, multiple processes with the same session are allowed to be in the CS at the same time. An execution of the last three sections is called an *invocation*. To request a “session”, a process picks a session number (usually it is done by simply writing an integer to a private variable) when it leaves the REMAINDER SECTION, and this session number can be different in each invocation. A process is said to be an *active* process, if it is in one of its invocation. We will say that processes are in *conflict* if their session numbers are different.

A solution to the GME problem consists of developing code for the ENTRY SECTION and the EXIT SECTION so that the following four properties are satisfied:

Mutual Exclusion (ME): No two conflicting processes can be in the CRITICAL SECTION at the same time.

Starvation Freedom (SF): If no process stays in the CRITICAL SECTION forever, then any process that enters the ENTRY SECTION eventually enters the CRITICAL SECTION.

Concurrent Entering (CE): In the absence of conflicting processes, a process in the ENTRY SECTION should be guaranteed to enter the CRITICAL SECTION within a bounded number of its own steps.

Bounded Exit (BE): After entering the EXIT SECTION, a process is guaranteed to leave it within a bounded number of its own steps.

The crucial difference between GME problem and the classical mutual exclusion problem is the concurrent entering property. The property was stated informally by Joung [12] and then was later formalized by Hadzilacos [9]. The intent of the concurrent entering property is to ensure concurrency: if multiple active processes request the same session and there is no conflicting process, these processes should be allowed to enter the CS without unnecessary synchronizations among themselves. If we do not care to ensure the concurrent entering property for GME problem, then almost any algorithm for classical mutual exclusion problem solves the GME. On the other hand, an algorithm of GME also solves the classical mutual exclusion problem, by simply assigning each process an unique session.

Several algorithms for the GME problem [13, 19, 1] satisfy a weaker version of this concurrency property, called *Concurrent Occupancy* property. Keane and Moir [13] gave a precise definition of the concurrent occupancy property.

Concurrent Occupancy: In the absence of conflicting processes, a process that requests a session will eventually enter the CRITICAL SECTION (even if other processes do not leave the CS).

The concurrent occupancy property wrongly formalizes Joung's intention that was stated informally in his paper. The original idea of the concurrency in the group mutual exclusion problem is that in the absence of conflicting processes, processes requesting the same session can not only concurrently stay in the CS but also enter the CS simultaneously without unnecessary synchronizations. Consider a general solution of GME which has the properties of ME, SF, BE and Concurrent Occupancy. Suppose process i and process j are requesting the CS with the same session. If process i is synchronizing with process j in the ENTRY SECTION and we temporarily stop process j, then the concurrent occupancy does not guarantee that process i will enter the CS within a bounded number of its own steps. Accordingly, process i might be delayed to enter the

CS for arbitrarily long even though there is no conflict. On the other hand, a GME solution with the Concurrent Entry property will guarantee process i enters the CS within a bounded number of its own steps. It is obvious to see that the concurrent entering property implies the concurrent occupancy property.

1.3 Fair Properties of Group Mutual Exclusion

A GME algorithm has the concurrency to let similar processes access the resources simultaneously while preserving the safety to prevent conflicting accesses. However, like the classical mutual exclusion problem, no stipulation is made about the order in which processes can enter the CS. To ensure fairness, it is often required to satisfy the *First-Come-First-Served* (FCFS) property in addition to the above mentioned four basic properties. Informally, FCFS property demands that conflicting processes are allowed into the CS in the same order in which they made requests. To formalize the idea, we split the ENTRY SECTION into two parts viz., *Doorway* section and *Waiting room* section, like we did in the classical mutual exclusion problem. The doorway section is a wait-free section of code, i.e., a section of code that can be completed by a process in a bounded number of its own steps. The waiting room section is where the actual synchronization with other processes occurs and may entail indefinite waiting. Processes enter the CS after finishing the waiting room section.

The concept of doorway helps us to define the following two relations between processes:

Doorway-Precede: If process i finished executing the doorway section before process j begins to execute the doorway section, then we say process i doorway-precedes process j.

Doorway-Concurrent: If neither process i doorway-precedes process j nor process j doorway-precedes process i, then we say process i is doorway-concurrent with process j

Now, we are ready to state the FCFS property formally:

First-Come-First-Served (FCFS): If process i doorway-precedes process j, and they request different sessions, then process j does not enter the CRITICAL SECTION before process i.

The FCFS property in GME ensures fairness if conflicting processes request to enter the CRITICAL SECTION. However, it fails to ensure fairness among processes with the same session. For example, Suppose process i and process j request the CS with the same session S. Process i doorways-proceeds process j. Suppose another process k requesting a conflicting session S' is doorway-concurrent with both process i and process j. It is possible that process j enters the CS first because of the concurrent entry property. However, process i can be blocked by the conflicting process k in the ENTRY SECTION. Thus, although process i finished the doorway before process j starts the doorway, the FCFS property does not guarantee the order in which these three processes enter the CS. Therefore, Jayanti et al. [11] formalized another property to ensure fairness among processes with the same sessions.

First-In-First-Enable (FIFE): Suppose a process i doorway-precedes a process j, and both process i and process j request the same session. If process j enters the CRITICAL SECTION before process i, then process i should enter the CRITICAL SECTION within a bounded number of its own steps.

The concurrent entry property captures the Joung's idea that, if there is no conflict, a process should enter the CS without waiting. A natural extension to strengthen the property is as follows: if a process finished the doorway before any conflicting process, that process should enter the CS without waiting. This property also ensures fairness in the sense that conflicting processes arriving later can not prevent a process that has already finished the doorway from entering the CS. Jayanti et al. [11] gave such a stronger version of the concurrent entry property.

Strong Concurrent Entering: If a process i requests a session S and process i doorway-precedes every conflicting process, then process i enters the CRITICAL SECTION within a bounded number of its own steps.

In conclusion, an algorithm that solves the group mutual exclusion problem must satisfy four basic properties of ME, CE, BE and SF. In addition to these four basic properties, it should be desirable if the algorithm can satisfy the fairness properties of GME: FCFS, FIFE and strong concurrent entering.

1.4 Modeling and Measures of Complexity

In distributed computing, there are two major types of models for mutual exclusion problems: message-passing model and shared-memory model. In this thesis, we discuss the problem and solutions in shared-memory model.

We consider a system consisting of N processes, named $1, 2, \ldots N$ and a set of shared variables. Each process also has its own private variables, which are not accessible to other processes. A process can communicate with other processes only by writing into and reading from the shared variables. An execution is modeled as a sequence of process steps. In each step, a process performs some local computation or writes into or reads from a shared variable. The processes take steps asynchronously. Specifically, this means that an unbounded numbers of steps of other processes could be performed in between two consecutive steps of a process. Also, we assume all processes are *live*; this means that if a process has not terminated it will eventually execute its next step. For simplicity of our discussion of GME algorithms, we sometimes temporarily suspend a process; this means some steps by other processes are taken during the "freezing". The process continues to execute the next step after we resume the process from freezing.

We only allow simple read and write operations on shared variables. Although we assume that these read and write operations are atomic, we do not assume that processes have access to more powerful synchronization primitives such as *Fetch-and-Add*, *Compare-and-Swap* etc. However, one of our algorithms in chapter 3 uses *Fetch-and-Add*, and hence, we will discuss that instruction further there.

Two types of shared variables are commonly used in distributed algorithms: *bounded registers* and *unbounded registers*. Bounded shared register means there is a bound on the maximum value that can be stored in the shared variable. Unbounded register can store an arbitrarily large value in them.

There are two general types of shared-memory models in the literature, viz., *Distributed Shared Memory (DSM)* model and *Cache-Coherent (CC)* model. The difference of these two models depends on the physical location of the shared variable and how processes access the shared variables.

In the DSM model, each processor has its own memory module and each shared variable is assigned and belongs to a particular processor. When a process is referencing a shared variable, if the shared variable is allocated to itself, the access is a *local memory reference*; if the shared variable is allocated to another processor, the access is a *remote memory reference*. For example, if there are two processes i and j, and two shared variable v_i and v_j. We assume v_i is stored in the memory module of process i and v_j is stored in the memory module of process j. If process i is referencing v_i or process j is referencing v_j, the access is a local memory reference. On the other hand, if process i is accessing v_j or process j is accessing v_i, the operation is a remote memory reference.

In the CC model, all shared variables are located in a global memory module that is not allocated to any particular processor. Each processor has its private cache that is associated with the global memory module. On such a machine, a shared variable becomes locally accessible by migrating it to a local cache. In the cache, the copy remains there until it is invalidated as a result of the shared variable being modified by some other process i.e., some other processes overwriting the shared variable in the global memory module (even if the newly written value is the same as the previous value). Migrating a shared variable to a processor's local cache is considered as a remote memory reference. Also, when a process writes a shared variable, the process writes the variable to the global memory module, which also involves a remote memory reference. In general, a process must make a remote memory reference whenever it writes a shared variable, tries to read a shared variable for the first time and tries to read an invalid shared variable. For example, if there are two processes i and j, and a shared variable v in a system. If process i or process j reads v for the first time, they need to copy the value of v from the global memory module into its local cache, which is a remote memory reference. If process i writes a value into v, it also makes a remote memory reference. When process j subsequently reads v after process i's write operation on v, even if v is already in process j's local cache, it is invalidated by the hardware protocol and thus results in a remote memory reference for process j to copy the new value of v to its local cache.

The idea of the remote memory reference is formally defined as follows:

Remote Memory Reference (RMR): A Remote Memory Reference (RMR) by process i is an attempt to access a memory location that is not physically located in process i's local memory or its local cache.

In this thesis, the term space complexity means shared space complexity in this thesis. Shared space complexity counts the total amount of shared space a solution entails. We do not count the private variables when measuring space complexity.

We use the term time complexity to denote Remote Memory Reference (RMR) complexity. Remote memory references are the most time consuming operations because they involve interconnect traversal and hence we use RMR complexity as a measure of the performance of the algorithm.

RMR Complexity: The RMR complexity of an algorithm that solves the GME problem is defined to be the maximum number of remote memory references that a process uses when it executes the ENTRY SECTION and EXIT SECTION.

It is important to note that most mutual exclusion algorithms use busy-wait loops. The idea is that a process *spins* on a variable until another process modifies that variable to let the waiting process make progress. In the CC model, the spinning on a shared variable is counted as only one remote memory reference if the value is not changed. In the DSM model, if the spin variable is not located in local memory of the process, the busy-wait will involve an unbounded number of remote memory references. Hence, it is desirable to make processes spin on local variable only.

Local-Spin: A spin that involves only registers those are physically in the local memory.

An algorithm in which all spins are local is called a *Local-Spin* algorithm.

Chapter 2

Literature Survey

We conduct a literature survey in this chapter. We first introduce Joung's algorithm [12], which is the first algorithm that solves the group mutual exclusion problem. Then, in section 2.1, we present an algorithm discovered by Hadzilacos [9], which is the first group mutual exclusion algorithm that satisfies the FCFS property. Next, in section 2.2, we present a modification of the above algorithm that has reduced shared space complexity of $\Theta(N)$ by Jayanti et al.[11]. At last, we present an algorithm by Takamura and Igarashi [19]. They developed the algorithm by generalizing Lamport's Bakery Algorithm. Their algorithm does not have the concurrent entry property and the starvation freedom property. Hence, their algorithm is not a valid GME algorithm.

Keane and Moir [13], Alagarsamy and Vidyasankar [15] also presented several GME algorithms, however, their algorithms do not satisfy the concurrent entry property. We will not discuss these algorithms in this thesis since they do not correctly solve the problem.

2.1 Joung's Algorithm

Joung [12] formulated the notion of the group mutual exclusion and presented the first algorithm to solve it in 1998. The idea of Joung's algorithm comes from Knuth's 2-process mutual exclusion algorithm [14]. It uses a shared variable *Turn* to resolve the competition between two conflicting processes. Joung's algorithm has $\Theta(N)$ shared variable complexity and unbounded RMR complexity in both DSM and CC model. Another drawback of the algorithm is that a bound on the number of sessions should be known in advance.

Three shared variables are used in Joung's algorithm depicted in Figure 2.1. The first one is *Turn*, an integer variable with a value in {1,2,..,m}, where m is total number of sessions. *Turn* indicates the session that is currently enabled. Processes requesting the enabled session have higher priority to enter the CS than other processes. The second one is *Flag*, an array of size N, where N is the total number of processes. *Flag*$[i]$ represents process i's requesting status in the system and it has two components. The first component of *Flag*$[i]$ is a single variable with a value in {*PASSIVE*, *REQUEST*, *IN_CS*}. A value of "PASSIVE" means process i is in the REMAINDER SECTION. A value of "REQUEST" indicates process i is in the ENTRY SECTION and requesting a session. A value of "IN_CS" represents process i has a temporary permission to enter the CS or process i is currently in the CS. A process that has a temporary permission will fail to enter the CS if there could be a conflict. The second component of *Flag*$[i]$ is a single variable of an integer. It represents process i's current session. If it is 0, then process i is not requesting any session. The third shared variable is *Successor*, an integer array of size N. Process i is said to be "captured" if it finds *Successor*$[i]$ is equal to its session number. A captured process can directly enter the CS. The private variable *mysession* saves the session number of the process.

It is stored in the process' local memory and can only be accessed by it.

Figure 2.1 Header for Joung's Algorithm in Figure 2.2

```
shared variables:
    Turn: integer from $\{1..m\}$, initialized arbitrarily
    Flag: array$[1..N]$ of $\{(\text{PASSIVE}, 0), (\text{REQUEST}, integer), (\text{IN\_CS}, integer)\}$,
        initially all $(\text{PASSIVE}, 0)$
    Successor: array$[1..N]$ of integer, initially all $0$

private variables:
    mysession: integer, initially $0$
```

Figure 2.2 Joung's Algorithm; Process $i \in \{1..N\}$

```
repeat
    REMAINDER SECTION
    $Flag[i] := (\text{REQUEST}, mysession)$
    $Successor[i] := 0$
    repeat
        $Flag[i] := (\text{REQUEST}, mysession)$
        await $(Successor[i] = mysession) \vee (\text{NEXT\_SESSION}(Turn) = mysession)$
        $Flag[i] := (\text{IN\_CS}, mysession)$
    until $((Successor[i] = mysession) \vee ((\text{NONE\_IN\_CS}(mysession))$
          $\overrightarrow{\wedge}(\text{NO\_SUCCESSOR}(mysession))$
          $\overrightarrow{\wedge}((Turn = mysession) \vee (\text{ALL\_PASSIVE}(Turn)))))$
    if $Successor[i] \neq mysession$ then
        $Turn := \text{NEXT\_SESSION}(mysession + 1)$
        for $j := 1$ to $N$ do
            if $j \neq i$ then
                if $Flag[j] \in \{(\text{REQUEST}, mysession), (\text{IN\_CS}, mysession)\}$ then
                    $Successor[j] := mysession$
                end if
            end if
        end for
    end if
    CRITICAL SECTION
    $Flag[i] := (\text{PASSIVE}, 0)$
forever
```

Figure 2.3 Function NEXT_SESSION(*session*) for Joung's algorithm in Figure 2.2

```
$k :=$ session;   /* session is a input of a session number */
next $:= k + m$;   /* m is the total number of sessions */
for $j := 1$ to $N$ do
    $(\_, x) :=$ Flag[$j$]
    if $x \neq 0$ then
        if $x \leq k$ then
            $x = x + m$
        end if
        if $x \leq$ next then
            next $:= x$
        end if
        $next := next$ mod $m$
    end if
end for
return $next$
```

Figure 2.4 Method NONE_IN_CS(*mysession*) for Joung's Algorithm in Figure 2.2

```
for $j := 1$ to $N$ do
    if $j \neq i$ then
        (state, $x$) := Flag[$j$]
        if ((state = IN_CS) $\wedge$ ($x \neq$ mysession)) then
            return false
        end if
    end if
end for
return true;
```

Figure 2.5 Method NO_SUCCESSOR(*mysession*) for Joung's Algorithm in Figure 2.2

```
for $j := 1$ to $N$ do
    if $j \neq i$ then
        $(\_, x) :=$ Flag[$j$]
        if ((Successor[$j$] $:= x$) $\wedge$ $x \neq$ mysession) then
            return false
        end if
    end if
end for
return true;
```

Figure 2.6 Method ALL_PASSIVE(*Turn*) for Joung's algorithm in Figure 2.2

```
for $j := 1$ to $N$ do
    if $j \neq i$ then
        $(\_, x) :=$ Flag[$j$]
        if $x =$ Turn then
            return false
        end if
    end if
end for
return true;
```

Four functions are used in this algorithm:

1. NEXT_SESSION(*session*) (Figure 2.3):

   The function accepts an input of a session number *mysession* and outputs the first requested session number in the sequence $\textit{session}, \textit{session}+1, ..., \textit{session}+m-1$, where each element in the sequence is reduced by the total number of sessions m. For example, if $h = ((\textit{session}+m-1) \bmod m)$ and $h < \textit{session}$, then the actual sequence is: $\textit{session}, \textit{session}+1, .., m-1, 0, 1, .., h-1, h$. The function will access the *Flag* variable of every process to record their sessions and requesting status. Then it returns the first session in the sequence that some process is requesting.

2. ALL_PASSIVE(*mysession*) (Figure 2.6)

   The input is a session number. The output is a Boolean value. The function checks whether there exists a process that is requesting the same session as the input session. The function accesses the *Flag* variable of every process and records the sessions. If some process is requesting the same session as the input, it returns false. If no process requests the same session, it returns true.

3. NONE_IN_CS(*mysession*) (Figure 2.4)

When a process i executes the function with the input *mysession*, it will check whether there exists a conflicting process in the CS or with a temporary permission in the ENTRY SECTION, ready to enter the CS. At first, the function accesses the *Flag* variable of every process. If the *Flag* variable of a process contains a "IN_CS" state and a different session than that of process i, the function returns false. If no such process exists, it returns true.

4. NO_SUCCESSOR(*mysession*) (Figure 2.5)

   The function checks if there exists a conflicting process that is captured. At first, it reads the *Flag* variable of all processes and records the sessions being requested. If some process j is requesting a conflicting session and *Successor*$[j]$ has the same conflicting session number, then the function returns false. Else, it returns true.

The algorithm is presented in Figure 2.2. The doorway consists of lines 3-4. The waiting room starts from line 5 to line 19. After a process i leaves the REMAINDER SECTION, it first updates its requesting status (line 3), setting *Flag*$[i]$ to <REQUEST, *mysession* >. It indicates process i is in the ENTRY SECTION and requesting a session *mysession*. Then, process i initializes *Successor*$[i]$ to empty (line 4), to prepare to be captured by a "captain process". A process p is a "captain process" if *Successor*$[p]$ is not equal to its session (line 10). A "captain process" p will capture processes with the same session and update their *Successor* (line 12-18) before process p enters the CS. Processes that are captured will enter the CS within a bounded number of their own steps.

After going through the doorway, process i executes a loop to check whether it is safe to enter the CS, corresponding lines 5-9. First, process i firstly resets *Flag*$[i]$ (line 6) as it did in line 1, the reason for which will be explained later. Next, process i waits until one of the following two conditions is satisfied (line 7). The first condition is (*Successor*$[i]$ $=$ *mysession*), which means process i is captured by a "captain process". The second one is NEXT_SESSION(*Turn*) $=$ *mysession*, which indicates that process i's session is the "closest" requested session to the current enabled session among all requested sessions. If either one of the condition is satisfied, process i sets *Flag*$[i]$ to be $(\text{IN\_CS}, \textit{mysession})$ (line 8), to indicate that process i has a temporary permission to enter the CS. With a temporary permission, process i further checks whether there will be a conflict if it enters the CS (line 9). Process i will pass the find check if either one of the following conditions is satisfied:

1. Process i is captured by a "captain process" (NEXT_SESSION(*Turn*) $=$ *mysession*).
2. The following three sub conditions are all satisfied:

    i. No conflicting process has a "IN_CS" state. That is no conflicting process is in the CS or has a temporary permission (NONE_IN_CS(*mysession*) = True).

    ii. No conflicting process is captured by a "captain process" (NO_SUCESSOR(*mysession*) = True).

    iii. The session of process i is the currently enabled session or no process is requesting the current enabled session, indicated by *Turn* ((*Turn* $=$ *mysession*) $\vee$ (ALL_PASSIVE(*Turn*))).

If the first condition is met, process i enters the CS without any wait. If the second condition is satisfied, no conflicting process will enter or stay in the CS. So the request of process i can be fulfilled. Process i exits the loop (lines 5-9) if either condition is satisfied. If process i finds none of the above two conditions is satisfied, it gives up its temporary permission and executes the loop again starting from resetting *Flag*$[i]$ to (REQUEST, *mysession*) (line 6). The reset is essential because after process i fails to make progress with a temporary permission (*Flag*$[i]$ = (IN_CS, *mysession*)), it needs to check again if it still has the priority to enter (line 7). Hence, process i needs to reset *Flag*$[i]$ to prevent stopping other processes which may enter the CS.

Once process i exits the loop (lines 5-9), it examines whether it is a "captain process" (line 10). If *Successor*$[i] \neq$ *mysession*, then process i is a "captain process", and it will update the *Turn* to the next session to be enabled (line 11). The next enabled session is set to NEXT_SESSION(*mysession* $+ 1$). It returns the first requested session among the sequence: *mysession* $+ 1$, *mysession* $+ 2, ...,$ *mysession* $+ m$, where all elements are reduced by mod m. Next, process i captures other processes by updating their *Successor* (lines 12-18). It will catch every process j that requests the same session (line 14) and set their *Successor*$[j]$ to *mysession* (line 15). After that, process i enters the CS. When it leaves the CS, it sets *Flag*$[i]$ to (PASSIVE, 0), which indicates process i is in the REMAINDER SECTION (Line 21).

The algorithm has the properties of ME, CE, BE and SF. The mechanism of getting a temporary permission and then checking for a conflict repeatedly guarantees the mutual exclusion property. The concurrent entry property is ensured as the value of the *Turn* variable at any time is the enabled session. To see this, observe that in the absence of conflicting processes, process i entering the ENTRY SECTION will find its session is the next enabled session because that is the only session requested among all processes. Therefore, when process i is executing line 9 the condition is immediately satisfied and so, it will enter the CS within a bounded number of its own steps. The bounded exit property is obvious since the EXIT SECTION is made up of only one line without any wait. The starvation freedom property is ensured as *Turn* is updated by executing the function NEXT_SESSION(*mysession* $+ 1$). Thus, every requested session will be enabled eventually. A full proof of the correctness is given by Joung [12].

Note that the order of evaluating the conditions NONE_IN_CS(*mysession*), NO_SUCCESSOR(*mysession*) and ($Turn = mysession$) $\vee$ (ALL_PASSIVE(*Turn*)) in line 9 is crucial to guarantee the mutual exclusion property (the notation $c_1 \overrightarrow{\wedge} c_2$ denotes the conjunction of c_1 and c_2 where c_1 is evaluated before c_2). To see this, for example, assume we evaluate NO_SUCCESSOR(*mysession*) before NONE_IN_CS(*mysession*) at line 9. The total number of sessions is 5 and they are s_1, s_2,..., s_5. Suppose processes i and j request session s_2 and process k requests session s_3. The initial value of *Turn* is 1, which means session s_1 is currently enabled. Consider the following scenario:

1. Process k enters the ENTRY SECTION first, it sees that no process is requesting, and so the NEXT_SESSION(*Turn*) returns 3. Hence, process k exits the busy-wait loop in line 7.
2. Processes i and j enter the ENTRY SECTION. Since they request s_2, which has the smaller session identifier than s_3, NEXT_SESSION(*Turn*) returns 2 at line 7. So, both processes i and j also exit the wait loop in line 7.
3. Process j sets its state to (IN_CS, s_2) and evaluates the condition at line 9. At this time, since no other process has the state of IN_CS and no process is captured by a "captain process", the conditions of NO_SUCCESSOR(*mysession*) and NONE_IN_CS(*mysession*) is true. Also, the ALL_PASSIVE(*Turn*) is true at this time. Thus, process j finished line 9 and jumps out of the repeat loop (lines 5-9).

4. Process k changes its *Flag* to (IN_CS, s_3) and starts to evaluate the conditions in line 9. Because of the assumption, NO_SUCCESSOR(*mysession*) is evaluated first. Since no processes is captured by a "captain process", process k finds out the condition is true. Next, it begins to evaluate NONE_IN_CS(*mysession*) and starts from process i. Since process k finds that the *Flag*$[i]$ is (REQUEST,s_2), process k passes process i and continue to check process j's *Flag*.

5. Process j sets the *Turn* to be the next requesting session s_3. Before process k inspects process j's *Flag*, process j finds that process i is also requesting session s_2. Therefore, process j captures process i and sets *Successor*$[i]$ to s_2. After that, process j finishes the CS, enters the EXIT SECTION and resets its *Flag* to (PASSIVE, 0).

6. Process k now sees process j's *Flag* and finds out process j has a "PASSIVE" state. Thus, the NONE_IN_CS(*mysession*) is true. Moreover, since *Turn* is s_3 at this time, the conditions at line 9 is all satisfied, and then process k enters the CS.

7. Process i at line 9 learns that it is captured by a "captain process" and so it enters the CS directly. Therefore, process i and process k are in the CS with different sessions, which is a violation of mutual exclusion property.

Note that if NONE_IN_CS(*mysession*) is evaluated first, when process k checks NO_SUCCESSOR(*mysession*), process j already captured process i. So, process k will find out process i is captured and so, it will wait for process i to finish the CS.

It is easy to see that the algorithm uses $\Theta(N)$ bounded shared variables. However, it has unbounded RMR complexity in the CC model. Here we give an example to show that. Assume only two processes i and j are requesting the

CS. process i requests session s_1 and process j requests the different sessions s_2. Process i enters the ENTRY SECTION first. Since it is the only active process, process i finishes the repeat loop (lines 5-9). Then process j enters the ENTRY SECTION. Before process i enters the CS, it updates *Turn* to 2 because it finds a conflicting session s_2 is requested by process j (line 11). Next, when process j executes line 7, it gets a temporary permission because NEXT_SESSION(*Turn*) returns s_2. If we temporarily suspend process i, process j will not enter the CS because the condition of NONE_IN_CS(*mysession*) is false (process i is in the CS). Hence, process j gives up its temporary permission and restarts the repeat loop from line 5. As long as process i stays in the CS, process j will execute the loop an indefinite number of times. Since process j needs $O(N)$ RMR at line 7 and line 9 in the CC model and so, the total number of remote memory reference that process j made in the ENTRY SECTION is unbounded. Therefore, it is trivial to see the RMR complexity in the DSM model is also unbounded.

2.2 Hadzilacos's Algorithm

Hadzilacos's algorithm, shown in Figure 2.8, is the first group mutual exclusion algorithm that satisfies the FCFS property. Another advantage of it, when compared to Joung's algorithm, is that it does not require knowing a bound on the number of sessions in advance. The algorithm can be thought of as a modular composition of two independent algorithms, one, the "FCFS algorithm" provides the FCFS property (but does not necessarily grantee mutual exclusion), and the other, the "ME algorithm" provides the mutual exclusion property (but not necessarily FCFS). The idea of the "FCFS algorithm" is somewhat based on the FCFS mutual exclusion algorithm by Lycklama and Hadzilacos [17]. The "ME algorithm" is an extension of the One-bit mutual exclusion algorithm developed independently by Burns[5] and Lamport[16].

Four shared variable are used in the algorithm (Figure 2.7). The first one, *Session* is an integer array of size N, where N is the total number of processes. *Session*$[i]$ represents the session that process i is currently requesting. Process i is in the REMAINDER SECTION if *Session*$[i]$ is 0. *Turn* is an integer array of size N. It has a value from the set {0,1,2,3}. *Turn* is used somewhat like a counter. Process i increments *Turn*$[i]$ mod 4 each time once it goes through the doorway. Other processes wait on *Turn*$[i]$ to maintain the FCFS property. The third shared variable *Bypass* is a two-dimensional Boolean array of size N^2. It is used for ensuring the FCFS property and the starvation freedom property. The complete details of using *Turn* and *Bypass* variables will be explained later. *Competing* is a Boolean array of size N. If *Competing*$[i]$ is true, it means process i is competing with other processes to enter the CS. If *Competing*$[i]$ is false, it indicates process i is temporarily not competing to enter the CS.

Figure 2.8 presents the algorithm. The doorway of the ENTRY SECTION consists of lines 3-8. The "FCFS algorithm" consists of lines 3-15. The "ME algorithm" consists of lines 16-26.

Figure 2.7 Header for Hadzilacos's Algorithm in Figure 2.8

```
shared variables:
    Session: array[1..N] of integer, initially all 0
    Turn: array[1..N] of 0..3, initially all 0
    Bypass: array[1..N, 1..N] of Boolean, initially all false
    Competing: array[1..N] of Boolean, initially all false
private variables:
    mysession: integer, initially 0
    turn-snap: array[1..N] of 0..3, initially all 0
    bypassers: set of 1..N, initially $\emptyset$
```

Figure 2.8 Hadzilacos's Algorithm; Process $i \in \{1..N\}$

```
 1: repeat
 2:    REMAINDER SECTION
 3:    $Session[i] := mysession$
 4:    for $j := 1$ to $N$ do
 5:       $Bypass[i, j] :=$ false
 6:       $turn\text{-}snap[j] := Turn[j]$
 7:    end for
 8:    $Turn[i] := (Turn[i] + 1)$ mod $4$
 9:    $bypassers := \emptyset$
10:    for $j := 1$ to $N$ do
11:       await$((Session[j] \in \{0, mysession\}) \vee (Turn[j] \neq turn\text{-}snap[j]) \vee (Bypass[i, j]))$
12:       if $Session[j] = mysession$ then
13:          $bypassers := bypassers \cup \{j\}$
14:       end if
15:    end for
16:    $Competing[i] :=$ true
17:    for $j := 1$ to $i - 1$ do
18:       if $((Competing[j] =$ true$) \wedge (Session[j] \neq mysession))$ then
19:          $Competing[i] :=$ false
20:          await $((\neg Competing[j]) \vee (Session[j] = mysession))$
21:          goto line 16
22:       end if
23:    end for
24:    for $j := i + 1$ to $N$ do
25:       await $((\neg Competing[j]) \vee (Session[j] = mysession))$
26:    end for
27:    CRITICAL SECTION
28:    for all $j \in bypassers$ do
29:       $Bypass[j, i] :=$ true
30:    end for
31:    $Competing[i] :=$ false
32:    $Session[i] := 0$
33: forever
```

At first, process i updates its session *Session*$[i]$ to be *mysession* (line 2). Then process i initializes *Bypass*$[i, j]$ and *turn-snap*. It sets *Bypass*$[i, j]$ to false for every j (line 5) and records *Turn*$[j]$ into its local variable *Turn-snap*$[j]$ for every j (line 6). Next, process i increments *Turn*$[i]$ by one using arithmetic modulo 4 each time it passes through the doorway (line 8). After passing the doorway, i sets the private variable *bypasser* to be empty (whose purpose will be explained later). Next, to ensure the FCFS property, process i waits for every other process j until one of following conditions is satisfied:

i. Process j is in the REMAINDER SECTION or requesting the same session as process i ($\mathit{Session}[j] \in \{0, \mathit{mysession}\}$).

ii. Process i is doorway-concurrent with process j. Process j increments $\mathit{Turn}[j]$ after process i recorded $\mathit{Turn}[j]$ ($\mathit{Turn}[j] \neq \mathit{turn\text{-}snap}[j]$).

iii. Process i is allowed to bypass process j ($\mathit{Bypass}[i, j] = true$).

If process i finds $\mathit{Turn}[j] \neq \mathit{turn\text{-}snap}[j]$ at line 11, then process j must have updated the $\mathit{Turn}[j]$ (line 8) after process i copied $\mathit{Turn}[j]$ (line 6). If process i finds $\mathit{Turn}[j] = \mathit{turn\text{-}snap}[j]$ and process j is requesting a different session, then $\mathit{Turn}[j] = \mathit{turn\text{-}snap}[j]$ remains true until process j increments $\mathit{Turn}[j]$ when it enters the doorway next time. Consequently, process i waits for process j until it finished its current invocation.

The system can lead to a deadlock if we drop the shared array *Bypass*. Suppose a process p is in the ENTRY SECTION and we stop process p temporarily at line 9. Process q repeatedly requests the same session as process p for three times. In each request, process q increments its $\mathit{Turn}[q]$ and enters the CS by the concurrent entry property. In the fourth invocation, process q requests a conflicting session and increments $\mathit{Turn}[q]$ for the fourth time by modulo 4. If we resume process p at this time, it will find $\mathit{Turn}[q] = \mathit{turn\text{-}snap}[q]$ at line 11. Thus, process p will continue waiting for process q and process q will also wait on process p because of the FCFS property, which leads to a deadlock. The algorithm uses *Bypass* to solve this problem. When process q requests the same session as process p, it adds process p into the set *bypasser* (line 12-13). After process q finished the CS, it sets *Bypass*$[k, q]$ to true for every k in the set (line 29). Therefore, the previous deadlock can not occur as process p will find *Bypass*$[p, q]$ to be true in line 11.

The “FCFS algorithm” does not guarantee the mutual exclusion. It is possible that while process i is in the middle of its doorway, process j enters its doorway. Then, we say that processes i and j are doorway-concurrent. If two processes are doorway-concurrent, then the FCFS property does not dictate as to who should get into the CS first. Consider two conflicting processes i and j, both of which update their *turn-snap* before either one increments the *Turn*. Then, neither process i nor process j will wait for each other in line 10 because *Turn*$[i] \neq$ *turn-snap*$[i]$ and *Turn*$[j] \neq$ *turn-snap*$[j]$. Therefore, both processes i and j will get out of the “FCFS algorithm” and enter the CS simultaneously. In order to prevent this mutual exclusion violation, we need the “ME algorithm”.

The “ME algorithm” (line 16) is based on the elegant one-bit mutual exclusion algorithm independently discovered by Burns [5] and Lamport [16]. In “ME algorithm”, every process has a one bit shared variable *Competing*. After process i finished the “FCFS algorithm”, it sets *Competing*$[i]$ to true (line 16). Then process i checks all processes with smaller process identifier (lines 17-23). If such a process j has its bit *Competing*$[j]$ set to true, then process i resets *Competing*$[i]$ to false to allow process j to make progress (line 19) and waits for *Competing*$[j]$ to become false (line 20). Once *Competing*$[j]$ becomes false, process i sets *Competing*$[i]$ to be true and rechecks again all processes with smaller process identifiers (line 21). If process i finds that no process with lower identifier is competing, it then checks the higher-numbered processes (lines 24-26). If any of them is competing, process i waits on them. However, at this time process

i does not reset its *Competing*$[i]$ to false before waiting. Once it finds that no higher-numbered process is competing, process i enters CS. When it exits the CS, process i sets *Competing*$[i]$ to false and *Session* to 0, and enters the REMAINDER SECTION.

Hadzilacos's algorithm satisfies the properties of ME, CE, BE, SF and the FCFS. The algorithm uses $\Theta(N^2)$ shared variables. Unlike Joung's algorithm, a bound on the total number of sessions is not required to be known in advance. So, the set of sessions can be arbitrarily large.

In the DSM model, process i executing the algorithm owns *Competing*$[i]$, *Session*$[i]$, *Turn*$[i]$, and *Bypass*$[i, j]$ for all j in its local memory. Since process i does not busy-wait on its local variable (lines 11, 20, 25), the algorithm is of unbounded RMR complexity in the DSM model. Hadzilacos claims that the algorithm has $O(N)$ RMR complexity under the CC model. However, the "ME algorithm" they used is actually of $\Omega(N^2)$ RMR complexity in the CC model. Consequently, Hadzilacos's algorithm is actually of $\Omega(N^2)$ RMR complexity in the CC model. A detailed RMR complexity analysis of the "ME algorithm" is presented in Chapter 4.

2.3 JPT Algorithm

Hadzilacos mentioned [9] that there exists a simpler algorithm that solves the group mutual exclusion problem using only $\Theta(N)$ shared variables. However, in the algorithm, the shared variables *Turn*$[i]$ are unbounded integers (instead of integers in the range $0..3$), and process i in its doorway increments *Turn*$[i]$ as a regular integer (not modulo 4). Hence, the disadvantage of the algorithm is that it uses unbounded shared variables. Hadzilacos left it as an open problem to determine whether it is possible to devise a FCFS group mutual exclusion algorithm that runs in linear time and space using only *bounded* shared variables. Couple of years later, Jayanti et al. [11] presented an algorithm as a solution to the open problem. They came up with a clever modification to the Hadzilacos's algorithm to reduce the space complexity. The algorithm retained the idea of modular composition and also the "ME algorithm" from Burns [5] and Lamport [16]. They dropped the shared variable *Bypass* while retaining the mechanism to ensure the FCFS property.

As we mentioned before, variable *Bypass* and *bypasser* are used for solving the deadlock issue: process j passes process i enough number of times that process i can not figure out whether *Turn*$[j]$ is changed too many times or it is not changed at all. In this algorithm, Jayanti et al. resolve the problem by letting a process increments its *Turn* variable only if there is a conflict. Hence, if process i finds that no process requests a different session than that of process i, it leaves the *Turn*$[i]$ unchanged in the doorway. Notice that the value of *Turn* is bounded by doing modulo 11 arithmetic in this algorithm. To see that increasing the bound guarantees the deadlock freedom property, we show the deadlock will not happen as before. If a process j passes process i many times with the same session as that of process i, and also increments *Turn*$[j]$, then each time process j passes process i, there exists another process k that requests a different session than process i and process j. Since process j and process k request different sessions, they will synchronize when they are executing the "FCFS algorithm". Meanwhile, process i and process j will also synchronize in the same manner. After process j increments *Turn*$[j]$ at most 11 times, process j has to wait for process k to enter the CS and process k has to wait for process i to enter the CS. Thus processes i and j will not wait for each other even if process j passes process i and increases the *Turn*$[j]$ enough times. Hence, process i will enter the

CS eventually. A detailed proof of the algorithm is available in [11].

Figure 2.9 JPT Algorithm; Process $i \in \{1..N\}$

```
shared variables:
    Session: array[1..N] of integer, initially all 0
    Turn: array[1..N] of 0, 1, 2..11, initially all 0
    Competing: array[1..N] of Boolean, initially all false
private variables:
    mysession: integer, initially all 0
    turn-snap: array[1..N] of 0, 1, 2..11, initially all 0
1: repeat
2:    REMAINDER SECTION
3:    $Session[i] := mysession$
4:    for $j := 1$ to $N$ do
5:       $turn\text{-}snap[j] := Turn[j]$
6:    end for
7:    if CONFLICT(mysession) then
8:       $Turn[i] := (Turn[i] + 1)$ mod 12
9:    end if
10:   for $j := 1$ to $N$ do
11:      await $((Session[j] \in \{0, mysession\}) \vee (Turn[j] \neq turn\text{-}snap[j]))$
12:   end for
13:   $Competing[i] :=$ true
14:   for $j := 1$ to $i - 1$ do
15:      if $((Competing[j] = \textbf{true}) \wedge (Session[j] \neq mysession))$ then
16:         $Competing[i] :=$ false
17:         await $((\neg Competing[j]) \vee (Session[j] \in \{0, mysession\}))$
18:         goto line 13
19:      end if
20:   end for
21:   for $j := i + 1$ to $N$ do
22:      await $((\neg Competing[j]) \vee (Session[j] \in \{0, mysession\}))$
23:   end for
24:   CRTICAL SECTION
25:   $Competing[i] :=$ false
26:   $Session[i] := 0$
27: forever
```

Figure 2.10 Method CONFLICT($mysession$) for JPT Algorithm in Figure 2.9

```
1: for $j := 1$ to $N$ do
2:     if $Session[j] \notin \{0, mysession\}$ then
3:         return true;
4:     end if
5: end for
6: return false;
```

Figure 2.9 describes the algorithm. It uses only three shared variables: *Session*, *Turn* and *Competing*. These shared variables originate from the algorithm of Hadzilacos (Figure 2.7). The only crucial difference is that *Turn* has an integer value from {0, 1, 2, .., 11}. Two private variables "*mysession*" and "*turn-snap*" are used. The ENTRY SECTION consists of the "FCFS algorithm" (lines 3-12) and the "ME algorithm"(lines 13-23). The doorway is made up of line 3-9.

At first, process i writes its session into *Session*$[i]$ and records the *Turn* of every other process (lines 3-6). Then process i updates *Turn*$[i]$ in case there exists a conflicting process in the system. The condition is checked by executing the function CONFLICT(*mysession*) (line 7). It accesses the session of every other process and checks whether there is a process requesting a different session. If no process requests a conflicting session, the function returns false, else it returns true. So, process i increments *Turn*$[i]$ if the function returns true. Else, process i keeps *Turn*$[i]$ unchanged. After updating *Turn*$[i]$, process i waits on other processes to ensure the FCFS property (line 10-12). At line 11, process i waits for process j if process j requests a conflicting session ($Session[j] \notin \{0, mysession\}$) and ($Turn[j] = turn\text{-}snap[j]$) until one of these two conditions is no longer true. Thus the FCFS property is guaranteed. Notice that suppose a process i completed the doorway before process j begins its doorway. Then process j reads *Turn*$[i]$ in line 5 after process i completed line 8. If process i and j request different sessions, then process j has to wait on process i at line 11 to change *Turn*$[i]$ or reset the *Session*$[i]$ to 0 or the process j's session, neither of which can happen before process i leaves the CS. After process i finished the "FCFS algorithm", it enters the "ME algorithm" (line 13-23). Jayanti et al. uses the same "ME algorithm" as Hadzilacos [9] to guarantee the mutual exclusion property.

The algorithm satisfies the properties of ME, CE, BE, SF and the FCFS. It only uses only $\Theta(N)$ shared variables as compared to $\Theta(N^2)$ shared variables in Hadzilcos's algorithm. A bound on number of sessions is also not required in advance. Like Hadzilcos's algorithm [9], this algorithm is of unbounded RMR complexity in DSM model because process i does not on wait on its local shared variable at lines 11, 17 and 22. In the CC model, although Jayanti et al. did not explicitly claim so, their algorithm is considered to be linear time and space. For example, the recent paper by Bhatt and Huang [4] explicitly stats that the RMR complexity of the algorithm by Jayanti et al. is $O(N)$. Thus the open problem raised by Hadzilacos appears to be solved. However, since they use the same "ME algorithm" by Burns [5] and Lamport [16], this algorithm is actually of $\Omega(N^2)$ RMR complexity in the CC model. We will demonstrate this in detail the details in Chapter 4. Thus, the open problem prosed by Hadzilacos [9] is still open.

2.4 Takamura and Igarashi's Algorithms

Takamura and Igarashi [19] developed an algorithm for the group mutual exclusion problem by generalizing the Lamport's Bakery Algorithm. The idea is based on the method used in bakery stores. In the store, only one customer can be served at any point of time. A customer gets a unique token number upon entering the bakery store. The customer that holds the lowest token number is the next one served.

The algorithm is presented in Figure 2.11. It uses three shared variables. The first one is *Session*, an array of size N, where N is the total number of processes. *Session*$[i]$ indicates the session number that process i requests in the current invocation. The second shared variable is *Token*, an integer array of size N and *Token*$[i]$ represents the token number selected by process i. If *Token*$[i]$ is 0, that indicates process i is in the REMAINDER SECTION or it is in the process of selecting a token. The third one is *Choosing*, a Boolean array of size N and *Choosing*$[i]$ is true would indicate that process i is currently attempting to determine its token number in the ENTRY SECTION. The *Session* array and the *Token* array are initialized to zero and the *Choosing* array is initialized to false. It is easy to see that the algorithm uses $\Theta(N)$ shared variable. However, the token numbers used in this algorithm can grow unbounded.

When a process i leaves the REMAINDER SECTION, it first sets *Choosing*$[i]$ to true to signal other processes that it is currently attempting to get a token number (line 3). Then it selects its token number to be one more than the maximum of the token numbers of all other processes and places it in *Token*$[i]$ (line 4). Next, process i places its session *mysession* in *Session*$[i]$ (line 5) and sets *Choosing*$[j]$ to false to signal other processes that process i already got its token number (line 6).

For each other process j, process i waits until process j has selected its token number (line 8). Then process i waits for process j until one of the following condition is satisfied (line 9): (i) process j is in the REMAINDER SECTION (*Token*$[j] = 0$), (ii) process i has a lower token number than process j $((\mathit{Token}[i], \mathit{mysession}) \leq (\mathit{Token}[j], \mathit{Session}[j]))$ and (iii) process j requests the same session with process i ($\mathit{Session}[j] = \mathit{mysession}$). It is possible that two different processes read the same set of token numbers and pick the same token number. In that case, we use the session number to resolve the ties. The relation "less than" on ordered pairs of integers is defined by $(a, b) < (c, d)$ if $(a < c)$ or if $(a = c)$ and $(b < d)$. After finishing the loop (lines 7-10), process i enters the CS. When it exits the CS, process i sets *Token*$[i]$ and *Session*$[i]$ to 0, which indicates it is in the REMAINDER SECTION.

Figure 2.11 Takamura and Igarashi's Bakery Algorithm

```
shared variables:
    Session: array[1..N] of integer, initially all 0
    Token: array[1..N] of integer, initially all 0
    Choosing: array[1..N] of Boolean, initially all false
private variables:
    mysession: integerb
 1: repeat
 2:   REMAINDER SECTION
 3:   Choosing[i] := true
 4:   Token[i] := 1+ max of other token numbers
 5:   Session[i] := mysession
 6:   Choosing[i] := false
 7:   for j := 1 to N do
 8:     await Choosing[j] = false
 9:     await ((Token[j] = 0) ∨ ((Token[i], mysession) ≤ (Token[j], Session[j]))
              ∨(Session[j] = mysession))
10:   end for
11:   CRITICAL SECTION
12:   Token[i] := 0
13:   Session[i] := 0
14: forever
```

The algorithm satisfies the mutual exclusion property and bounded exit property, however, it does not have the concurrent entry property and the starvation freedom property. To see that, assume two processes i and j request the same session. Suppose process i got a token number and finished line 6 while process j is still selecting its token number at line 4. When process i checks

Choosing$[j]$ at line 8, it finds it to be false and waits for process j to finish selecting its token. If we temporarily suspend process j at this time, process i will wait for process j indefinitely. This is a concurrent entry violation as process i can not enter the CS within a bounded number of its own steps even though there is no conflicting process. On the other hand, if we temporarily stop process i, process j will pass process i and enter the CS because it finds (*Choosing*$[i] =$ false) and (*Session*$[j] =$ *Session*$[i]$) at line 8 and line 9. After leaving the CS and subsequently the EXIT SECTION, suppose process j requests again the same session as process i and selects a token number at line 4. If we resume process i at this time, it will find *Choosing*$[j]$ is false and keep waiting for process j. We can repeat this scenario as many times as we choose and this leads to a situation where process i never enters the CS while process j repeatedly enters the CS. Hence, a starvation occurs in the system.

Takamura and Igarashi [19] also presented two more algorithms based on generalizing Lamport's Bakery Algorithm. Their second and third algorithms have the starvation freedom property, however, both the algorithms do not have the concurrent entry property because of the same reason mentioned above. Moreover, they contain busy-wait loops in the EXIT SECTION. So, they do not have the bounded exit property as well. Hence, none of three algorithms presented by Takamura and Igarashi [19] can be considered a correct group mutual exclusion algorithm. In the next chapter, we show a correct generalization of Lamport's Bakery Algorithm that solves the group mutual exclusion problem.

Chapter 3

Generalizing the Bakery Algorithm

Takamura and Igarashi [19] tried to generalize Lamport's Bakery Algorithm to solve group mutual exclusion problem. However, their algorithms are not proper generalizations as they do not satisfy the concurrent entry property.

In this chapter, we first introduce Lamport's Bakery Algorithm for the classical mutual exclusion problem. Then, we present two GME algorithms based on Lamport's Bakery Algorithm. Our first algorithm uses the synchronization primitive Fetch-and-Add instruction. Our second algorithm uses only simple read and write instructions.

3.1 Lamport's Bakery Algorithm

Lamport's Bakery Algorithm is one of the best-known algorithms for classical mutual exclusion problem. The algorithm is based on a simple and elegant idea, which is commonly used in bakery stores. When entering the bakery store, every customer gets a token number that is larger than the token numbers of other customers that are waiting. The store can only serve one customer at any point of time. The customer with the smallest token number is the next one to be served.

Figure 3.1 Lamport's Bakery Algorithm

```
shared variables:
     Token: array[1..N] of integer, initially all 0
     Choosing: array[1..N] of Boolean, initially all false
 1: repeat
 2:     REMAINDER SECTION
 3:     Choosing[i] := true
 4:     Token[i] := 1+ max of other token numbers
 5:     Choosing[i] := false
 6:     for j := 1 to N do
 7:         if $j \neq i$ then
 8:             await Choosing[j] = false
 9:             await $((Token[j] = 0) \vee ((Token[i], i) < (Token[j], j)))$
10:         end if
11:     end for
12:     CRITICAL SECTION
13:     Token[i] := 0
14: forever
```

Shared variable *Token* is an integer array of size N, where N is the total number of processes. *Token*$[i]$ stores the token number of process i. If process i is in the REMAINDER SECTION, *Token*$[i]$ will be set to 0. The second shared variable is *Choosing*, a Boolean array of size N. If *Choosing*$[i]$ is true, it indicates that process i is currently selecting its token number in the doorway. If *Choosing*$[i]$ is false, it means process i has already picked up its token number and finished executing the doorway.

The Lamport's Bakery Algorithm is presented in Figure 3.1. The doorway of the algorithm consists of lines 3-5. The waiting room is made up of lines 6-11.

At the beginning, process i sets *Choosing*$[i]$ to be true (line 3), indicating that it is starting to pick up its token number. Then process i accesses the token numbers of every other process and calculates its own token number to be one more than the largest token number of all other processes (line 4). Process i then sets *Choosing*$[i]$ to be false to indicate that it is done with picking a token number (line 5). In the waiting room, for each other process j, process i first waits for process j to select its token number (line 7). Then process i waits until it has the lower token number than process j or until process j enters the REMAINDER SECTION. After the loop, process i enters the CS. It is trivial to see that, the process that enters the CS has the lowest token number among all processes that are requesting the CS. When process i exits the CS, it sets *Token*$[i]$ to 0 to denote that it is not interested in entering the CS.

We note that in order to ensure the mutual exclusion property it is crucial that process i first waits until process j has selected its token number before comparing. Suppose we drop line 8 and two processes i and j are both requesting the CS. It is possible that process j gets a smaller token number but does not update *Token*$[j]$ yet. When process i finishes the doorway, process i will find out *Token*$[j]$ is 0 and then enters the CS. Once process j finishes the doorway, process j has the smaller token number and so, it will also enter the CS while process i is still in the CS, which creates a mutual exclusion violation. We also note that the token number is compared using *lexicographic order*. That is $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$ (a, b, c and d are all integers). If processes i and j have the same token number, then the process with the lower process identifier has the higher priority.

Lamport's Bakery Algorithm satisfies the mutual exclusion, the starvation freedom, the bounded exit and the FCFS properties. As Lamport pointed out in [15], the implementation of computing the maximum of token numbers of other processes is critical to the correctness. One simple valid implementation entails the process first recording all other token numbers into its local memory, one at a time, and then calculating the maximum value over all token numbers. A drawback of the Lamport's Bakery Algorithm is that the token number can grow without a bound. It is trivial to see that processes can always request the CS and increment the value of the token infinitely. Thus, the algorithm uses unbounded shared variables.

3.2 A GME Bakery Algorithm Using Fetch-and-Inc

3.2.1 Introduction and Model

Takamura and Igarashi did not correctly generalize Lamport's Bakery Algorithm [19] to solve GME because their algorithms do not have the concurrent entry property. Here we present a correct generalization of Lamport's Bakery Algorithm to a GME algorithm. The algorithm has $\Theta(N)$ shared space complexity and $O(N)$ RMR complexity. Other than ordinary read and write instructions, our algorithm uses the *Fetch-and-Inc* instruction.

Fetch-and-Inc is a kind of read-modify-write synchronization primitives. A synchronization primitive is considered as a special instruction that atomically modifies the content of a memory location. The instruction is defined as follows:

```
Fetch-and-Inc (Var: integer) {
    $old\_var := Var$
    $Var := Var + 1$
    return $old\_var$
}
```

In the execution of Fetch-and-Inc (*Var*), a process first reads the shared variable *Var*. Then it increments *Var* by one and returns the original value of *Var*. The internal procedure of the instruction is executed atomically, i.e., it is guaranteed that a process will not be interrupted while performing the Fetch-and-Inc procedure. This means that only one process can be executing the Fetch-and-Inc at any point of time.

Figure 3.2 A GME Bakery Algorithm Using Fetch-and-Inc

```
shared variables:
    Session: array[1..N] of integer, initially all 0
    Token: array[1..N] of integer, initially all 0
    Counter: integer, initially 1
private variables:
    mysession: integer, initially 0
1:  repeat
2:      REMAINDER SECTION
3:      $Session[i] := mysession$
4:      $Token[i] :=$ Fetch-and-Inc($Counter$)
5:      for $j := 1$ to $N$ do
6:          await $((Token[i] < Token[j]) \vee (Session[j] \in \{0, mysession\}))$
7:      end for
8:      CRITICAL SECTION
9:      $Token[i] := 0$
10:     $Session[i] := 0$
11: forever
```

3.2.2 Commentary of the algorithm

Figure 3.2 presents our algorithm. It uses three shared variables. *Session* is an integer array of size N. *Session*$[i]$ represents the session that process i is requesting. If *Session*$[i]$ is 0, that means process i is in the REMAINDER SECTION. *Token* is an array of integer of size N. *Token*$[i]$ stores the token number of process i. *Counter* is an integer that can be read and written by every process. Processes access *Counter* to get their token numbers. The doorway of the algorithm consists of lines 3-4. The waiting room is made up of lines 5-7.

At first, process i saves its session number in *Session*$[i]$ to indicate to other processes, which session process i is requesting (line 3). Then process i reads the value of *Counter* as its token number and increments the *Counter* so that later processes get larger token numbers (line 4). Next, for each process j that is not in the REMAINDER SECTION, process i waits until it has the smaller token number than that of process j or process j is requesting the same session as process i (line 6). After the loop, process i enters the CS. When process i exits the CS, it resets *Token*$[i]$ and *Session*$[i]$ to 0 to indicate that it is in the REMAINDER SECTION.

3.2.3 Proof of the Correctness

The algorithm satisfies the mutual exclusion property, concurrent entry property, starvation freedom property, bounded exit property and the FCFS property. Here we present a complete proof of correctness of the algorithm.

Lemma 3.2.1: At any point of time, no two processes will have the same none-zero value as their token number.

Proof. At line 4, processes get token numbers by executing Fetch-and-Inc. Since Fetch-and-Inc is considered as an atomic instruction, every process will read a unique value of *Counter* and increment it. Thus, the value of *Counter* is monotonically increasing after each execution of Fetch-and-Inc and so, no two processes will have the same none-zero value as their token numbers. □

Corollary 3.2.2: If process i finds *Session*$[j]$ is 0 at line 6. Then process j can not enter the CS with a conflicting session before process i exits the CS.

Proof. If process i finds out *Session*$[j]$ is 0. That means process j is in the REMAINDER SECTION at that time. Since the *Counter* is monotonically increasing, process j will get a larger token number than process i. Therefore, if process

j requests the CS with a conflicting session, it will be blocked at line 6 until process i exits the CS. □

Theorem 3.2.3: The algorithm in Figure 3.2 has the Mutual Exclusion property.

Proof. Assume process i and process j are in the CS at the same time with different sessions. If process i finds *Session*$[j]$ is 0 and passes process j at line 6, by Corollary 3.2.2, process j can not enter the CS with a different session before process i exits the CS. Thus, process i should pass process j because it has the smaller token number than process j. Since process j has the larger token number, it will be blocked by process i at line 6 until process i exits the CS, which contradicts with our assumption. □

Theorem 3.2.4: The algorithm in Figure 3.2 has the Concurrent Entry property.

Proof. Suppose a process i is requesting the CS, and every other process j is in the REMAINDER SECTION or requesting the same session as process i. For each other process j, at line 6, process i finds that *Session*$[j]$ is 0 or *mysession*. Hence, process i will not wait for any other process and enter the CS within a bounded number of its own steps. □

Theorem 3.2.5: The algorithm in Figure 3.2 has the Starvation Freedom property.

Proof. Suppose there exists a process i that is requesting the CS, but can not enter the CS forever. Process i must wait for a process j infinitely at line 6. Thus, process j must request a conflicting session than process i and must always have smaller token number than process i. Since process j exits the CS eventually and the *Counter* always increments, process j can not pick up a smaller token number than process i again. Thus, process i will find that process j has the session of 0 or a larger token number eventually and stop waiting immediately, which is a contradiction with our assumption. □

Theorem 3.2.6: The algorithm in Figure 3.2 has the Bounded Exit property.

Proof. Since the EXIT SECTION of the algorithm does not have any busy-wait loops, a process that enters the EXIT SECTION, finishes it within a bounded number of its own steps. □

Theorem 3.2.7: The algorithm in 3.2 has the FCFS property.

Proof. Assume processes i and j request different sessions and process i finished the doorway (line 4) before process j starts the doorway (line 3). It is trivial to see process j gets a larger token number than process i. Therefore, process j will wait for process i at line 6 until process i finishes the CS. □

3.2.4 Complexity Analysis

The algorithm has $\Theta(N)$ shared variable complexity because it only uses integer arrays of size N and single integer variable. One draw back of this algorithm is that the shared variable *Counter* and *Token* can grow unbounded. The shared variable *Token* used in Lamport's Bakery Algorithm is also unbounded; however, the value of *Token* will start over again from 1 if there is no contention. In our algorithm, the shared variable *Counter* will grow unbounded even in a rare contention environment.

It is trivial to see that the algorithm has unbounded RMR complexity under the DSM model because the process is not spinning on its local shared variables only (line 6). There is only one loop in the algorithm (line 6). Thus, there can only be a constant number of RMR in other lines under the CC model. In line 6, when a process i is busy waiting on *Token*$[j]$, it will either change to 0 or to a larger token number than *Token*$[i]$. If *Token*$[j]$ changes to a larger token number, it will cause process i to terminate the wait loop. If *Token*$[j]$ changes to 0, that means process j in the EXIT SECTION or REMAINDER SECTION, process j will sets its session to 0 and cause process i stop waiting. However, it is possible that *Token*$[j]$ changes to 0 and process j requests again with another conflicting session immediately. But, in this case, process j will get a larger token number than *Token*$[i]$. Hence, line 6 can only involve a maximum of six RMR (One for *Token*$[i]$, three for *Token*$[j]$, two for *Session*$[j]$). Since line 6 is enclosed within a for loop that can run a maximum of N times and line 3 and line 4 costs only two RMR, the ENTRY SECTION is of $O(N)$ RMR complexity. It is also easy to see that the EXIT SECTION entails just two RMR. Therefore, the total RMR complexity of the algorithm is $O(N)$.

A drawback of the algorithm is that it uses a strong synchronization primitive viz., Fetch-and-Inc instruction. In the next section, we give another GME Bakery algorithm that uses only simple read and write instructions.

3.3 A GME Bakery Algorithm Using Only Read/Write Instructions

3.3.1 Introduction

The Lamport's Bakery Algorithm does not use any powerful synchronization primitives. In the same spirit, we present another GME bakery algorithm that using only simple read and write instructions. The algorithm satisfies the ME, CE, BE, SF and FCFS properties. It has $O(N)$ RMR complexity in the CC model and $\Theta(N)$ shared space complexity.

Figure 3.3 A GME Algorithm Using Only Read/Write Instructions

```
shared variables:
    Session: array[1..N] of integer, initially all 0
    Choosing: array[1..N] of Boolean, initially all false
    Token: array[1..N] of integer, initially all 0
private variables:
    mysession: integer, initially 0
 1: repeat
 2:     REMAINDER SECTION
 3:     $Session[i] := mysession$
 4:     $Choosing[i] :=$ true
 5:     $Token[i] := 1+$ max of other token numbers
 6:     $Choosing[i] :=$ false
 7:     for $j := 1$ to $N$ do
 8:         await $((Choosing[j] = \textbf{false}) \vee (Session[j] = mysession))$
 9:         await $(((Token[i], i) < (Token[j], j)) \vee (Session[j] \in \{0, mysession\}))$
10:     end for
11:     CRITICAL SECTION
12:     $Token[i] := 0$
13:     $Session[i] := 0$
14: forever
```

3.3.2 Commentary of the algorithm

The algorithm is presented in Figure 3.3. It uses three shared variables. The first one *Session* is an integer array of size N and *Session*$[i]$ indicates the session number that process i requests in the current invocation to enter the CS. If *Session*$[i]$ is 0, that means process i is in the REMAINDER SECTION. The second one is *Token*, an integer array of size N and *Token*$[i]$ represents the token number selected by process i. The third shared variable is *Choosing*, a Boolean array of size N and *Choosing*$[i]$ is true would indicate that process i is currently attempting to determine its token number. The *Session* array and the *Token* array are initialized to zero and the *Choosing* array is initialized to false. The doorway of the algorithm consists of lines 3-6 and the waiting room is made up of lines 7-10.

When a process leaves the REMAINDER SECTION, it first selects its session number and places it in *Session*$[i]$ (line 3). We assume that all session numbers are positive integers. Process i then sets *Choosing*$[i]$ variable to true to signal to other processes that it is currently attempting to select a token number (line 4). It then selects its token number to be one more than the maximum of the token numbers of all other processes and places it in *Token*$[i]$ (line 5). Next, process i sets *Choosing*$[i]$ to false to signal to other processes that it is done with selecting its token number (line 6).

In the waiting room, at line 8, for each other process j, process i checks to see whether process j is requesting the same session. If process j is requesting the same session, there is no problem. If process j is requesting a conflicting session, process i waits for process j to finish the doorway and set *Choosing*$[j]$ to be false. It is also possible that process j is in the REMAINDER SECTION, however, in that case, process j has already set *Choosing*$[j]$ to false, which prevent unnecessary wait. Then, process i waits for process j until either process j has a larger token number or it is in the REMAINDER SECTION or it is requesting the same session as process i (line 9). It is possible that two conflicting processes pick the same token number. In that case, we use the process identifier to resolve the ties (line 9). The relation "less than" among ordered pairs of integers is defined as in the Lamport's Bakery Algorithm. That is $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$ (a, b, c and d are all integers). For simplicity, if *Token*$[i] =$ *Token*$[j]$, we say the process with the smaller process identifier has the smaller token number. After the loop, process i enters the CS. It is easy to see that process i enters the CS if it has the smallest token number among all conflicting processes. In the EXIT SECTION, process i resets *Token*$[i]$ and *Session*$[i]$ to 0.

3.3.3 Proof of the Correctness

The algorithm satisfies the mutual exclusion property, concurrent entry property, starvation freedom property, bounded exit property and the FCFS property. Here we present a complete proof that the algorithm satisfies these properties.

Theorem 3.3.1: The algorithm in Figure 3.3 has the FCFS property.

Proof. Assume two conflicting processes i and j request the CS. Also assume that process i finished the doorway before process j starts the doorway. So, when process j is computing its token number, it will read the token number of process i and select a larger token number. Hence, when process j checks process i at line 9, since process j has a larger token number, it will wait for process i until it exits the CS. $\square$

Theorem 3.3.2: The algorithm in Figure 3.3 has the Mutual Exclusion property.

Proof. Suppose two conflicting processes i and j are in the CS at the same time with different sessions. By Theorem 3.3.1, process i and process j must be doorway-concurrent. Without loss of generality, we assume process i enters the CS first. Since processes i and j are doorway concurrent and they request different sessions, at line 8, process i waits for process j to finish the doorway. Since process i enter the CS first, it must have a smaller token number than process j. On the other hand, at line 9, process j waits for process i because it finds that process i has a smaller token number. Therefore, process j can not enter the CS until process i finishes the CS and resets its token number, which is contradicting with our assumption. $\square$

Theorem 3.3.3: The algorithm in Figure 3.3 has the Concurrent Entry property

Proof. Assume that process i requests a session s, and no other process requests a different session. This means, for every other process j, it holds that

Session$[j] \in \{0, \mathit{mysession}\}$. Since process i always checks whether *Session*$[j] \in \{0, \mathit{mysession}\}$ in busy-wait line 9, it can not wait on line 9. If process j has the session of 0, it is in the REMAINDER SECTION and so *Choosing*$[j]$ must be false. Thus, in line 8, process i can not be blocked by any non-conflicting process because it always find that either (*Choosing*$[i]$ $=$ false) or (*Session*$[j]$ $=$ *mysession*). Therefore, process i can not wait at any line if there is no conflicting process and so, it enters the CS within a bounded number of its own steps. □

Theorem 3.3.4: The algorithm in Figure 3.3 has the Bounded Exit property

Proof. Since the EXIT SECTION consists of two simple write instructions, a process that enters the EXIT SECTION will trivially finish it within a bounded number of its own steps. □

Theorem 3.3.5: The algorithm in Figure 3.3 has the Deadlock Freedom property

Proof. Suppose the algorithm does not satisfy the deadlock freedom property. Then there is an execution of the algorithm in which a nonempty set S of processes enter the ENTRY SECTION but none of them enters the CS, and no process enters the CS infinitely often. We observe that no process can wait at line 8 forever since every process j requests the CS will finish the doorway eventually and set *Choosing*$[j]$ to true. Therefore, processes in the set S must wait on line 9 forever. There exists a process i that has the smallest token number among all processes in S and that process i will pass all other processes at line 9 and enter the CS, which is a contradiction with our assumption. □

Theorem 3.3.6: The algorithm in Figure 3.3 has the Starvation Freedom property.

Proof. Lamport proved [16] that if an algorithm satisfies the deadlock freedom property and the FCFS property, then it necessarily satisfies the starvation freedom property. Hence, combining Theorem 3.3.1 and Theorem 3.3.5, we know the algorithm has the starvation freedom property. □

3.3.4 Complexity Analysis

It is trivial to see that the algorithm has $\Theta(N)$ shared space complexity. However, like Lamport's Bakery Algorithm, it uses unbounded shared registers.

We now analyze the RMR complexity of the algorithm.

In the DSM model, since we assign *Session*$[i]$, *Choosing*$[i]$,*Token*$[i]$ to process i, it is easy to see the algorithm has unbounded RMR complexity as the busy-wait is not exclusively on local shared variables.

We now analyze the RMR complexity in the CC model, in the waiting room, there are only two loops viz., the busy-wait loops in line 8 and line 9. In line 8, when process i is busy waiting for a process j, if *Choosing*$[j]$ changes to false, then process i will immediately terminate the wait. It is possible that process j requests another conflicting session and sets the *Choosing*$[j]$ to true again. In that case, since process i doorway-proceeds process j, process j will get a larger token number than process i. So, process j can not enter the CS to finish that invocation and therefore can not change the *Session* again until process i finishes the CS. Thus, line 8 can only involve a maximum of five RMR (three for *Choosing*$[j]$ and two for *Session*$[j]$). Similarly in line 9, when process i is busy-waiting on *Token*$[j]$, if *Token*$[j]$ changes to a larger token number, process i will stop waiting. If *Token*$[j]$ changes to 0, that means process j is in the REMAINDER SECTION or EXIT SECTION. That means process j sets *Session*$[j]$ to 0 eventually and terminate the waiting of process i. It is also possible that process j starts a new invocation and sets the *Session*$[j]$ to another conflicting session. In that case, since process i doorway-proceeds process j, process j will get a larger token number than process j, which will also terminate the wait. Hence, line 9 involves a maximum of six RMR (one for *Token*$[i]$, three for *Token*$[j]$, two for *Session*$[j]$). There are only constant number of RMR in line 8 and line 9. As these two lines are enclosed within a for loop that can run a maximum of N times, it follows that the entire waiting room is of $O(N)$ RMR complexity. Note that the doorway made up of line 3 through line 6 involves only constant number of RMR, except for the implicit loop in line 5. The implicit loop in line 5 has $O(N)$ RMR complexity as it involves inspecting the token numbers of all other processes. Finally, it is easy to see that the EXIT SECTION consisting of lines 12-13 involves exactly two RMR. Hence, the overall RMR complexity of this algorithm in the CC model is $O(N)$.

Chapter 4

A Flaw in the Literature

Hadzilacos developed [9] the first GME algorithm that satisfies the FCFS property. As mentioned before, he developed it by using a modular composition of a "FCFS algorithm" and a "ME algorithm". His algorithm is of $\Theta(N^2)$ shared space complexity and is claimed to have $O(N)$ RMR complexity. Hadzilacos also posed as an open problem the development of a GME algorithm with linear time and space while using only bounded shared variables and only simple read and write instructions. A few years later, Jayanti et al. [11] came up with a modification to the "FCFS algorithm" to reduce the shared space complexity while retaining the overall structure of the Hadzilacos's algorithm. Their algorithm has $\Theta(N)$ shared space complexity and they inherited the claim that it has $O(N)$ RMR complexity. The algorithm of Hadzilacos [9] as well as that of Jayanti et al. [11] use the same "ME Algorithm", which is called the "One-bit Algoirthm" for classical mutual exclusion problem. The "One-bit Algorithm" is originally discovered by Burns [5] and Lamport [16]. In this chapter, we first show the "One-bit Algorithm" used by Hadzilacos and Jayanti et al. is of $\Omega(N^2)$ RMR complexity. Thus, the open problem of linear time and space GME algorithm is not truly solved by Jayanti et al.

Taubenfeld [20] developed the Black-White Bakery Algorithm to solve classical mutual exclusion problem while using only bounded shared variables. Here we give an exposition of his Black-White Bakery Algorithm. We leave it as an open problem to investigate whether it is possible to bound the shared variables in our GME Bakery Algorithm by generalizing the Black-White Bakery Algorithm.

4.1 A Flaw in the Literature

As we showed in Chapter 2, the algorithms by Hadzilacos [9] and Jayanti et al. [11] satisfy the FCFS property. However, it is possible that while process i is in the middle of the doorway, a conflicting process j enters its doorway. Under this circumstance, we say that processes i and j are doorway-concurrent. In this case, the FCFS property does not dedicate as to who should get into the CS first. In fact, in their algorithm presented in Figure 2.9 and Figure 2.10, if two conflicting processes i and j record the *turn-snap* before each other updates their *Turn*, both processes i and j can get out of the "FCFS Algorithm" and can potentially enter the CS simultaneously. In order to prevent this from happening, we need the "ME Algorithm". In the worst case, consider a situation where all the n processes are doorway concurrent and are requesting different sessions. Then all of them can get out of the "FCFS Algorithm" and move on the "ME Algorithm". In order to simplify the analysis, we consider the "ME Algorithm" alone.

4.1.1 One-bit Mutual Exclusion Algorithm

The "ME Algorithm" is independently discovered by Burns [5] and Lamport [16]. The algorithm is the first mutual exclusion algorithm that uses the minimum possible shared space. Every process only uses one-bit shared variable to guarantee the mutual exclusion. Thus, it is also called the "One-bit" mutual exclusion algorithm. Figure 4.1 presents the code of the algorithm for process i.

The algorithm only uses one shared variable *Competing*, a Boolean array of size N, where N is the total number of processes. Every process owns a single a single-writer multi-reader shared bit *Competing*$[i]$. Only process i can write into *Competing*$[i]$; other processes can read it. If *Competing*$[i] =$ true, process i is competing for entry into the CS with other processes.

While entering the ENTRY SECTION, process i sets its bit to true and checks all processes with smaller-identifier (lines 3-5). If any of them, say process j, is found to have set its bit *Competing*$[j]$ to true, then process i resets *Competing*$[i]$ to false (line 6), allowing processes with smaller-identifier to make progress. Later, process i waits for process j's bit to become false (line 7) and then restarts the whole competition by going to line 3. Having checked all processes with smaller-identifiers, process i checks processes with larger-identifier and waits for each of them to set its bit to false (lines 11-13). This time, however, while process i is waiting it does not set its bit to false. After that, process i enters the CS. It turns out that this simple algorithm guarantees the mutual exclusion by using only one-bit shared variable.

The basic idea behind the algorithm is easy to understand. At any point of time, the process with the smallest process number has the highest priority to make progress because it will not set its *Competing* to false before it enters the CS. Note that the "goto" statement of line 8 is crucial for the mutual exclusion property. To see that, suppose we drop the "goto" and substitute line 8 with (*Competing*$[i]$ = true). Consider the following sequence of execution of three processes $i, j, k (i < j < k)$ are requesting the CS.

1) Processes j and k sets their *Competing* bit to true.
2) Process k checks all processes with smaller-identifiers and finds that only *Competing*$[j]$ is true. Since process j has a smaller-identifier, process k sets its *Competing*$[k]$ to false and waits for *Competing*$[j]$ to become false.
3) Process i starts requesting and sets its bit to true
4) Process j checks all processes with smaller-identifiers and finds that process i's bit is set to true. So process j sets *Competing*$[j]$ to false and waits for *Competing*$[i]$ to become false.
5) Process i finds no processes with smaller-identifier is requesting and all processes with larger-identifier sets their bits to false. Process i enters the CS.
6) Process k now finds *Competing*$[j]$ to be false. Since we substitute the "goto" with a new line 8. Process k sets its bit to true and finish checking processes with smaller-identifiers (process k checked process i before checking process j).
7) Process k finds that no process with larger-identifier is requesting the CS. So, process k enters the CS now.
8) Processes i and k are now in the CS at the same time.

On the other hand, if we keep the line of "goto", process k will restart checking processes with lower-identifier after it terminates the waiting for process j. Hence, process k will find out that process i's bit is set to true and then wait for it.

The One-bit mutual exclusion algorithm has the property of ME and DF. However, it does not satisfy the starvation freedom because process with smaller-identifier always has higher priority. For example, when a process p is wanting to enter the CS, it is possible that a process q with a smaller-identifier requests the CS again and again. It is possible that process p waits for process q forever. This can happen if every time process p is scheduled, it finds process q with smaller-identifier has its competing bit sets to true.

Figure 4.1 One-bit Mutual Exclusion Algorithm

```
shared variables:
    Competing: array[1..N] of Boolean, initially all false
1:  repeat
2:      REMAINDER SECTION
3:      Competing[$i$] := true
4:      for $j := 1$ to $i - 1$ do
5:          if ((Competing[$j$] = true) then
6:              Competing[$i$] := false
7:              await (($\neg$Competing[$j$])
8:              goto line 3
9:          end if
10:     end for
11:     for $j := i + 1$ to $N$ do
12:         await (($\neg$Competing[$j$])
13:     end for
14:     CRITICAL SECTION
15:     Competing[$i$] := false
16: forever
```

4.1.2 RMR Complexity Analysis

The RMR complexity of the algorithm is unbounded in both DSM and CC model. This is because the algorithm does not satisfy the starvation freedom property. Accordingly, there is no bound on the maximum number of RMR that a process makes before entering the CS.

In Hadzilacos's algorithm, the modular composition of the "FCFS Algorithm" and the "One-bit" mutual exclusion algorithm guarantee the starvation freedom property. So, we need to analyze the RMR complexity of the one-bit algorithm under the assumption that it has starvation freedom. In this section, we show that the One-bit mutual exclusion algorithm is of $\Omega(N^2)$ RMR complexity when used in the GME algorithm of Hadzilacos.

Assume N processes requesting N different sessions enter the ENTRY SECTION at about the same time in Hadzilacos' algorithm (Figure 2.8). Since all processes are doorway-concurrent, they pass the "FCFS Algorithm" and enter the One-bit mutual exclusion algorithm. Consider the following sequence of events.

1) Process N sets its competing bit to true.

2) Process $(N-1)$ sets its bit to true.

3) Process N checks all processes with lower-identifier and finds that process $(N-1)$'s bit is set. So, process N sets its bit to false and waits for *Competing*$[N-1]$ to become false.

4) Process $(N-2)$ sets its bit to true.

5) Process $(N-1)$ checks all processes with lower-identifier and finds that process $(N-2)$'s bit is set. So, process $(N-1)$ sets its bit to false and waits for *Competing*$[N-2]$ to become false.

6) Process N now finds *Competing*$[N-1]$ to be false and so restarts the competition by setting its bit to true.

7) Process N checks all processes with lower-identifier and finds that process $(N-2)$'s bit is set. So, process N sets its bit to false and waits for *Competing*$[N-2]$ to become false.

8) Process $(N-3)$ sets its bit to true.

9) Process $(N-2)$ checks all processes with lower-identifier and finds that process $(N-3)$'s bit is set. So process $(N-2)$ sets its bit false and waits for *Competing*$[N-3]$ to become false.

10) Process N now finds *Competing*$[N-2]$ to be false and so restarts the competition by setting its bit to true.

11) Process N checks all processes with lower-identifier and finds that process $(N-3)$'s bit is set. So, process N sets its bit to false and waits for *Competing*$[N-3]$ to become false.

12) .

13) .

14) .

15) Process N checks all processes with lower-identifier and finds that process 1's bit is set. So, process N sets its bit to false and waits for *Competing*$[1]$ to become false.

16) Process 1 checks all processes with larger-identifier and finds that all the bits are false and enters the CS.

17) Process 1 exits the CS and sets its bit to false.

Noted that during the above sequence of events, process N got blocked once by each one of the processes with lower-identifier. At the end of the above sequence, process 1's request is satisfied. Also, at the end of the above sequence, the *Competing* bits of process 2 through N are all false. Now, we can iterate the same sequence of the events, but this time with only process 2 through N participating. We can recursively iterate the same sequence of events again and again until finally we have only process N participating.

The net effect is that in this worst scenario, process N got blocked by process $(N-1)$ a total of $(N-1)$ times, by process $(N-2)$ a total of $(N-2)$ times and so on. So, the total number of times, process N gets blocked by some other process is

$$\sum_{k=1}^{N-1} k = \frac{N(N-1)}{2} = \Omega(N^2)$$

In the CC model, at least one remote memory reference is involved each time a process gets blocked and hence the RMR complexity of the One-bit mutual exclusion algorithm when used as part of Hadzilacos's algorithm is of $\Omega(N^2)$ in the CC Model. It was erroneously claimed in [9] and tacitly inherited in [11] that this algorithm is of $O(N)$ RMR. That claim is clearly wrong as illustrated above. Hence the problem of developing a linear time ($O(N)$ RMR complexity) and linear space ($O(N)$ shared space) algorithm for the group mutual exclusion problem, originally posed by Hadzilacos, is still open. In the remainder of this chapter, we develop such an algorithm.

4.2 Bounding Lamport's Bakery Algorithm

4.2.1 Introduction

The Lamport's Bakery Algorithm [16] is one of the best-known mutual exclusion algorithms. However, the algorithm uses unbounded shared variables. Several attempts [10, 21, 22] have been made to bound the shared space of the Lamport's Bakery Algorithm. However, solutions proposed are quite complicated [10] or even incorrect [21, 22]. In 2004, Taubenfeld [20] came up with a simple and elegant modification of the Lamport's Bakery Algorithm called **Black-White Bakery Algorithm** that uses only bounded shard variables. In this section, we introduce an exposition of the Black-White Bakery Algorithm.

4.2.2 Black-White Bakery Algorithm

Figure 4.2 depicts the algorithm. The doorway consists of lines 3-6. The waiting room is made up of lines 7-14. The algorithm uses a multi-writer multi-reader shared bit variable called *GlobalColor* that can only be black or white. Every process gets a token in each invocation, which consists of two parts: the token-color and the token-number. The algorithm bounds the growth of token numbers by coloring the token with the colors *black* or *white*. When a process enters the ENTRY SECTION, it first picks up its token-color by reading the *GlobalColor*. Then, process i selects its token-number to be one more than the largest token number of processes that have the same token-color. Next, process i waits until its colored token is the "lowest" and then it enters the CS. The order between tokens is defined as follows: If process i and process j have different token-colors, the token whose token-color is *different* from the value of the *GlobalColor* is smaller. If process i and process j have the same token color, the token with the smaller token-number is smaller. It is possible that processes have the same token-color and token-number. In that case, the process with the smaller identifier is deemed to have the smaller token. Once process i exits the CS, it updates the *GlobalColor* to be the opposite of its own token-color. By doing this, process i give priority to processes that have the same token-color as process i.

Figure 4.2 Black-White Bakery Algorithm

```
shared variables:
    Choosing: array[1...N] of Boolean, initially all false
    GlobalColor: a bit of type {black, white}, initialized arbitrarily
    Token-Color: array[1...N] of type {black, white}, initialized arbitrarily
    Token-Number: array[1...N] of integer, initially all 0

1:  repeat
2:      REMAINDER SECTION
3:      Choosing[i] := true
4:      Token-Color[i] := GlobalColor
5:      Token-Number[i] := 1+ max of token numbers of processes with the same
        color
6:      Choosing[i] := false
7:      for j := 1 to N do
8:          await Choosing[j] = false
9:          if (Token-Color[i] = Token-Color[j]) then
10:             await (((Token-Number[i], i) < (Token-Number[j], j))∨
                        (Token-Color[i] ≠ Token-Color[j]) ∨ (Token-Number[j] = 0)
11:         else
12:             await ((Token-Color[i] ≠ GlobalColor)∨
                        (Token-Color[i] = Token-Color[j]) ∨ (Token-Number[j] = 0)
13:         end if
14:     end for
15:     CRITICAL SECTION
16:     if Token-Color[i] = black then
17:         GlobalColor := white
18:     else
19:         GlobalColor := black
20:     end if
21:     Token-Number[i] := 0
22: forever
```

We now take a more detailed view of the algorithm. At first, in the ENTRY SECTION, process i sets *Choosing*$[i]$ to be true to indicate that it enters the doorway (line 3). Then process i selects its token-color by reading the shared bit *GlobalColor* and saving it into *Token-Color*$[i]$ (line 4). Next, process i selects a number that is one more than the largest token-number of processes that have the same token color (line 5). Later, process i sets *Choosing*$[j]$ to be false to indicate that it has finished the doorway. In the waiting room, for each process j, process i first waits for process j to finish the doorway in case it has started it (line 8). Then, process i compares the token-color of process j with that of itself (line 9). If process j has the same token-color, at line 10, process i waits until it notices that either (i) process i has the smaller token number or (ii) process j has reentered the ENTRY SECTION with the opposite token-color to that of process i or (iii) process j has got out of the CS and entered the REMAINDERS SECTION. If process j has different token-color than process i, at line 12, process i waits until either (i) process i's token-color is different with *GlobalColor* or (ii) process j has reentered the ENTRY SECTION with same token-color as process i's or (iii) process j has got out of the CS and entered the REMAINDER SECTION. After checking all other processes, process i enters the CS. When it exits the CS, process i sets the *GlobalColor* to the opposite of its own token-color (lines 16-21) and reset its token-number to 0.

It is trivial to see that if a process enters the CS, then processes with the different token-color than process i can not enter the CS before all processes having the same token-color with process i enter the CS. This crucial property ensures the mutual exclusion property and the bounded shared variable. Note that the second condition in line 10 and line 12 are essential to the correctness of the algorithm. For example, assume a process i is waiting for a process j at line 10 or 12. If process j reenters the ENTRY SECTION and gets a token-color again, it is possible that process j gets a new token-color that is different than previous token-color. In that case, process i should not wait for process j any more because process i doorway-precedes process j. Moreover, it is possible that process i will still wait for process j, if process i does not recheck process

j's token color, and this will result in a deadlock situation. Thus, by checking the second condition in line 10 and line 12, process i will detect process j has reentered the ENTRY SECTION and terminate the wait.

The Black-White Bakery Algorithm satisfies the properties of ME, DF and FCFS. It is not difficult to see that the algorithm is of $\Theta(N)$ shared space complexity. Note that when the *GlobalColor* changes, the token-numbers start all over again from 1. Therefore, the maximum value of token-number can have without a change is N and so the algorithm uses bounded shared variables. In the DSM model, the algorithm has unbounded RMR complexity because it is busy-waiting on remote shared variable. Under the CC model, since the busy-wait loops in lines 8, 10 and 12 can only involve constant number of RMR, the outer loop in the ENTRY SECTION is of $O(N)$ RMR complexity, and hence the RMR complexity of the algorithm is $O(N)$.

The Black-White Bakery Algorithm uses a simple and elegant manner to solve the disadvantage of unbound shared variable for Lamport's Bakery Algorithm. Since our GME Bakery Algorithm inherits the same disadvantage of Lamport's Bakery Algorithm, it might be possible to bound our GME Bakery Algorithm using similar technique. We leave it as an open problem to investigate whether it is possible to bound the shared variables in our GME Bakery Algorithm by using color mechanism like the Black-White Bakery Algorithm.

Chapter 5

Conclusion

5.1 Summary

Dijkstra first introduces the classical mutual exclusion problem in 1965 [7]. In 1998, Joung [12] gave the first statement of the group mutual exclusion problem and proposed a solution. Later, it has been extensively studied and a large number of algorithms have been developed.

Lamport's Bakery Algorithm is one of the best-known algorithm for classical mutual exclusion problem. In this thesis we give two simple and elegant algorithms that generalize Lamport's Bakery Algorithm to solve GME problem. Our first algorithm uses synchronization primitive Fetch-and-Inc. Our second algorithm uses only simple read and write operations. To the best of our knowledge, our algorithm is the simplest and elegant generalization of Lamport's Bakery Algorithm for the GME problem. Takamura and Igarashi [19] made an attempt to generalize the well-known Lamport's Bakery Algorithm to solve GME. They presented three different algorithms in that paper. Their first algorithm does not satisfy the starvation freedom property and concurrent entry property. Their second and third algorithms, apart from being quite complicated, do not satisfy the concurrent entry property and bounded exit property. All three of their algorithms satisfy the concurrent occupancy property.

Several GME algorithms have been developed while using using bounded shared variables. Joung's GME algorithm satisfies the four basic properties: ME, CE, BE and SF. However, it does not have the FCFS property. His algorithm has $\Theta(N)$ shared space complexity and unbounded RMR complexity in both DSM and CC model. Joung stated the concurrent entry property informally. Later, Keane and Moir [13] gave a precise definition of the concurrent entry property and devised a GME algorithm that satisfies it. However, Hadzilacos [9] pointed out that the formulation of the concurrent entry property by Keane and Moir does not correctly capture Joung's intent. Hadzilacos used the term "concurrent occupancy" to denote the property formulated by Keane and Moir and reformulated the concurrent entry property to capture Joung's original intent. Hadzilacos [9] gave a nice algorithm that satisfies the concurrent entry property as well as the FCFS property using bounded shared variables and only simple read and write operations. His algorithm can be thought of as a modular composition of two independent algorithms, one, the "FCFS algorithm" and the other, the "ME algorithm". The "ME algorithm" is based on the "One-bit Algorithm" for classical mutual exclusion problem that was independently developed by Burns [5] and Lamport [16]. The algorithm is of $\Theta(N^2)$ shared space complexity and unbounded RMR complexity in the DSM model. Hadzilacos claimed that it has $O(N)$ RMR complexity in the CC model. Also, Hadzilacos left as an open problem the development of an algorithm satisfying the properties of ME, CE, BE, SF and FCFS for the GME problem that runs in linear time (RMR complexity) and space (shared space complexity) while using only bounded shared variables and simple read and write instructions.

A couple of year later, Jayanti et al. [11] presented an algorithm as a solution to the above-mentioned open problem. They came up with a modification of the FCFS algorithm to reduce the shared space complexity while retaining the overall structure of Hadzilacos's algorithm and the "One-bit Algorithm". Although they did not explicitly claim so, their algorithm is considered to be of linear space and time complexity (for example, see the citation in [4]).

We demonstrate that both algorithms by Hadzilacos [9] and Jayanti et al. [11] are of $\Omega(N^2)$ RMR complexity in the CC model. Therefore the open problem of development of a linear time and space GME algorithms proposed by Hadzilacos is still open.

5.2 Open Problems

In a recent paper, Attiya, Hendler and Woefel [3] provided a $\Omega(\log N)$ RMR bound for the classical mutual exclusion problem in both DSM and CC models. These lower bounds are valid even if the algorithms are allowed to use *Compare&Swap* primitive and *load-link*(LL)/*store-conditional*(SC) variables in addition to simple read and write operations. Since the classical mutual exclusion problem is a special case of GME, the lower bound on RMR complexity for group mutual exclusion problem is $\Omega(\log N)$ as well.

Bhatt and Huang [4] presented an algorithm for GME problem whose RMR complexity matches the lower bound. So their algorithm is optimal with respect to RMR complexity. However, it may be possible to beat the lower bound if the algorithm is allowed to use more powerful synchronization primitives such as *Fetch&Add*. In fact, several constant RMR algorithms [2, 8, 18] have been developed for the classical mutual exclusion problem using this approach.

We leave it as an open problem to investigate whether it is possible to create a GME algorithm that has constant RMR complexity in the CC model using more powerful synchronization primitives.

Danek and Hadzilacos [6] proved a lower bound of $\Omega(N)$ on the RMR complexity of local-spin GME algorithms under the DSM model regardless of how powerful synchronization primitives it may use. They also gave a local-spin GME algorithm whose RMR complexity matches the lower bound. In this sense, their algorithm is an optimal algorithm. However, their algorithm uses $\Theta(N^2)$ shared space. It is still unknown whether there exists a local-spin GME algorithm has both linear time complexity and linear space complexity in the DSM model.

Bibliography

[1] K. Alagarsamy and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. *Unpublished manuscript*, 1999.

[2] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6–16, 1990.

[3] H. Attiya, D. Hendler, and P. Woelfel. Tight rmr lower bounds for mutual exclusion and other problems. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 217–226. ACM, 2008.

[4] V. Bhatt and C. Huang. Group mutual exclusion in o (log n) rmr. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 45–54. ACM, 2010.

[5] J. Burns. *Complexity of Communication Among Asynchronous Parallel Processes*. PhD thesis, Georgia Institute of Technology, January 1981.

[6] R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *DISC*, volume 3274 of *LNCS*, pages 71–85. Springer, 2004.

[7] E. Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*, pages 289–294. Springer, 2001.

[8] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[9] V. Hadzilacos. A note on group mutual exclusion. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 100–106. ACM, 2001.

[10] P. Jayanti, K. Tan, G. Friedland, and A. Katz. Bounding lamports bakery algorithm. In *SOFSEM 2001: Theory and Practice of Informatics*, pages 261–270. Springer, 2001.

[11] S. Jayanti, P.and Petrovic and K. Tan. Fair group mutual exclusion. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 275–284. ACM, 2003.

[12] Y. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[13] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32. ACM, 1999.

[14] D. Knuth. Additional comments on a problem in concurrent programming control. *Commun. ACM*, 9(5):321–322, May 1966.

[15] L. Lamport. A bug in the bakery algorithm. Technical report, Technical Report CA–7704–0611, Massachusette computer associates, inc, 1977.

[16] L. Lamport. The mutual exclusion problem: Parts i and ii. *Journal of the ACM (JACM)*, 33(2):313–348, 1986.

[17] E. Lycklama and V. Hadzilacos. *A first-come-first-served mutual exclusion algorithm with small communication variables*. ACM Transactions of Programming Languages and Systems 13(4), (1991), 558-576.

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

[19] M. Takamura and Y. Igarashi. Group mutual exclusion algorithms based on ticket orders. In *Computing and Combinatorics*, pages 232–241. Springer, 2003.

[20] G. Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. In *Distributed Computing*, pages 56–70. Springer, 2004.

[21] S. Vijayaraghavan. A variant of the bakery algorithm with bounded val-

ues as a solution to abrahams concurrent programming problem. *Proc. of Design, Analysis and Simulation of Distributed Systems*, 2003.

[22] T. Woo. A note on lamport's mutual exclusion algorithm. *Operating Systems Review*, 24(4):78–80, 1990.