LOCAL-SPIN GROUP MUTUAL EXCLUSION ALGORITHMS

by

Robert Danek

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

LOCAL-SPIN GROUP MUTUAL EXCLUSION ALGORITHMS

Robert Danek

Master of Science

Graduate Department of Computer Science

University of Toronto

2004

The group mutual exclusion (GME) problem is a variant of the mutual exclusion problem in which multiple processes that request the same session (and thus belong to the same "group") may be admitted to the critical section concurrently. We examine N-process group mutual exclusion under two different shared-memory models: the distributed shared-memory (DSM) model and the cache-coherent (CC) model. We prove that the remote memory reference (RMR) complexity of any GME algorithm in the DSM model is $\Omega(N)$, and we present and prove correct several local-spin GME algorithms whose RMR complexity matches this lower bound. We also present a 2-session local-spin GME algorithm for the CC model that beats the lower bound of the DSM model, and use it to construct an M-session GME algorithm. Each algorithm we present is in the form of a reduction to a FCFS abortable mutual exclusion algorithm.

Acknowledgements

Firstly, I would like to thank my supervisor, Vassos Hadzilacos, for the insight and feedback that he provided on my work. His numerous suggestions helped shape this thesis into its current form.

I would also like to thank Sam Toueg for the helpful comments that he provided as my second reader.

Finally, I would like to thank my family and friends for their encouragement and patience while I completed my degree. Without their support, I would not have been able to make it this far.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Mutual Exclusion Problem

The N-process mutual exclusion problem is a well-studied problem in the field of distributed computing. The motivation for the problem arises from the fact that there are certain resources in systems, called critical resources, that can only be accessed by at most one process at a time. Examples of such resources include a disk drive, a printer, or a "software" resource such as a data structure that can be modified by different processes. One can imagine the havoc that can be wreaked in a system if multiple processes were allowed to write to different parts of a disk, write different documents to a printer, or update the same data structure, at the same time.

To formalize the mutual exclusion problem, we assume that processes execute an algorithm structured as per Figure 1.1.

Figure 1.1 Structure of Algorithm Executed by Processes

```
loop
  Non-Critical Section (NCS)
  Trying Protocol (TP)
  Critical Section (CS)
  Exit Protocol (EP)
end loop
```

The CS is the section of the algorithm where a process accesses the critical resource.

We assume that the CS is finite and that a process that enters the CS eventually leaves it.

A solution to the N-process mutual exclusion (ME) problem consists of a trying and exit protocol (called the ME algorithm) so that the following properties are satisfied:

Mutual Exclusion: If a process p is in the CS, then no process $q \neq p$ is in the CS concurrently with p.

Lockout Freedom: If a process p enters the trying protocol, then p eventually enters the CS.

Bounded Exit: If a process enters the exit protocol, then the process returns to the NCS in a bounded number of its own steps.

We assume that the trying and exit protocols do not refer to any variables that are used inside the NCS or the CS.

An ME algorithm clearly guarantees that at most one process accesses a critical resource at a time. However, the order in which processes are admitted to the CS if more than one processes tries to enter the CS simultaneously is undefined. For example, suppose at some time t a process p tries entering the CS and remains in the trying protocol for a long time. Then, at some time t' long after t, a process $q \neq p$ attempts to enter the CS. It is possible for an ME algorithm to admit q into the CS before p.

The notion of one process attempting to enter the CS long after another is vague. However, despite this vagueness, one can intuitively see the scenario described above is unfair. Consider the situation of waiting in line at a bank or a grocery store: one expects that it is fair for those people who arrived in the line earlier to be served before those who arrived in the line at some later time. Similarly, processes that "arrive" (i.e., leave the NCS) earlier should be granted access to the CS earlier.

In order to formalize the concept of fairness, we first split the trying protocol into two parts: a doorway, and a waiting room. The doorway is a piece of code that a process

executes in a bounded number of its own steps. We now assume processes execute an algorithm structured as per Figure 1.2.

Figure 1.2 Structure of Algorithm Executed by Processes (Fair Case)

```
loop
  Non-Critical Section (NCS)
  Doorway
  Waiting Room
  Critical Section (CS)
  Exit Protocol (EP)
end loop
```

An ME algorithm that satisfies the following property is considered fair:

First-Come-First-Served (FCFS): If a process p finishes the doorway before a process $q \neq p$ starts the doorway, then p enters the CS before q enters the CS.

The FCFS property captures the notion that processes that attempt to enter the CS earlier than other processes will be granted access to the CS earlier. An ME algorithm satisfying the FCFS property is referred to as an FCFS ME algorithm.

We now return to our example of a line at a bank or grocery store. Besides the expectation of fairness in such lines, one also expects that a person can leave the line at any time prior to being served. For example, one might want to take advantage of this ability if one spends too long in the line without being served. Processes may want an analogous ability when attempting to access a critical resource. There may be a lot of processes simultaneously wanting to access the CS, and this may cause a long wait before a process is actually granted entry into the CS. In this case, a process should be able to somehow abort its attempt to enter the CS. ME algorithms that provide this ability are called abortable ME algorithms.

An abortable ME algorithm works as follows: when a process leaves the NCS, it first executes a trying protocol that consists of a doorway and waiting room (see Figure 1.2), where the waiting room is simply a busy-wait loop. (A busy-wait loop is an unbounded loop in which a process waits for one or more shared variables to acquire certain values.

Busy-wait loops are discussed more in the next chapter.) Once the process is in the busy-wait loop, it may decide to abort its attempt to enter the CS, at which point it abandons busy-waiting, executes a section of code referred to as the abort protocol, and then returns to the NCS. Otherwise, if the process succeeds in entering the CS, then the process proceeds through the CS and to the exit protocol, as with the other ME algorithms.

In addition to the ME properties, an abortable ME algorithm satisfies the following property:

Bounded Abort: If a process enters the abort protocol, then the process returns to the NCS in a bounded number of its own steps.

1.2 The Group Mutual Exclusion Problem

Up until now we have been considering resources that can only be accessed by at most one process at a time. It turns out there are other types of resources in systems that can be accessed by more than one process concurrently as long as all the processes accessing the resource belong to the same "group". One example of such a resource is provided by Joung [11]. A Computer Supported Cooperative Work application may make use of an electronic white board (the resource), to which multiple users (the processes) have access. When a user wants to share some piece of information about a certain topic (the group) with other users, he posts it to the white board, and any users interested in the same topic may access that information, and potentially add other pieces of information by writing to the white board. However, users interested in a different topic cannot use the white board until the current users are finished. Another example is provided by Hadzilacos [8]. Consider a server (the resource) that has the ability to cache some subset of data from a large distributed database. Clients (the processes) of this server that are interested in the subset of data (the group) that the server currently has cached can

access the server concurrently, but those clients interested in some other subset of data must wait until no other process is accessing the current data.

Clearly a mutual exclusion algorithm is sufficient for protecting access to these types of resources, but the drawback of using such an algorithm is that it does not admit any concurrency. This leads us to the definition of the N-process group mutual exclusion problem.

In the group mutual exclusion problem, originally defined by Joung [11], processes execute an algorithm structured as per Figure 1.1. Before a process leaves the NCS, it declares which session it wants to attend (i.e., to which "group" it wants to belong). For our purposes, this is done by simply writing an integer to a private variable. When a process is outside of the NCS, we say that the process is requesting the session whose value is written to that variable.

A solution to the N-process group mutual exclusion (GME) problem consists of a trying and exit protocol (called the GME algorithm) so that the following properties[1] are satisfied:

Mutual Exclusion: If a process p, requesting a session s, is in the CS, then no other process q, requesting a session $s' \neq s$, is in the CS concurrently with p.

Lockout Freedom: If a process p enters the trying protocol, then p eventually enters the CS.

Bounded Exit: If a process p enters the exit protocol, then p returns to the NCS in a bounded number of its own steps.

Concurrent Entering: If a process p is requesting a session s, and no other process is requesting a session $s' \neq s$, then p enters the CS in a bounded number of its own steps.

[1] Some of the property names are the same as those given for the mutual exclusion problem. When we refer to these properties, it will be clear from the context whether we are referring to the ME properties or the GME properties.

Similar to the ME problem, we assume that the trying and exit protocols do not refer to any variables used inside the NCS or CS.

GME algorithms guarantee that a resource is sufficiently protected while at the same time, under certain circumstances, allowing processes requesting the same session to access the resource concurrently. However, as with ME algorithms, there is no guarantee on the order in which processes are admitted to the CS.

Consider the FCFS property we defined in the context of the ME problem. The purpose of the FCFS property there is to capture the notion that it is fair for a process that attempts to enter the CS earlier than other processes to be granted entry into the CS before those other processes. An analogous notion of fairness in the context of the GME problem is for processes that request a session s to be granted entry into the CS earlier than processes that request a session $s' \neq s$ some time later. Assuming processes execute an algorithm structured as per Figure 1.2, this notion of fairness is captured by the following property, defined by Hadzilacos [8]:

First-Come-First-Served (FCFS): If a process p, requesting a session s, finishes the doorway before a process q, requesting a session $s' \neq s$, starts the doorway, then p enters the CS before q enters the CS.

A GME algorithm satisfying the above property is referred to as an FCFS GME algorithm.

The FCFS property only addresses the problem of fairness between processes that request different sessions, but says nothing about processes requesting the same session. To address the problem of fairness among all processes, one might naively consider adopting the FCFS property as it was stated in the context of the ME problem. However, this is not possible, since the FCFS property stated in that manner, without the embellishment given above, would conflict with the concurrent entering property, which is undesirable. To see the conflict with the concurrent entering property, consider the following scenario:

a process p finishes executing the doorway before another process q starts executing the doorway, where both processes request the same session, and all other processes are in the NCS. The FCFS property says that p must enter the CS before q, which means that q may have to wait until p enters the CS. This violates the concurrent entering property, which says that both p and q must enter the CS in a bounded number of their own steps.

Instead, we consider a notion of fairness among processes requesting the same session that works at an intuitive level as follows: suppose a process p requests some session at some time t, and long after t, another process q requests the same session. Now, suppose q enters the CS before p. Then it is fair that p should be able to enter the CS unimpeded.

The foregoing intuition is captured formally by the following property, defined by Jayanti et al. [10]:

First-In-First-Enabled (FIFE): If a process p, requesting session s, completes the doorway before a process $q \neq p$, also requesting session s, starts the doorway, and q enters the CS before p, then p enters the CS in a bounded number of its own steps.

Another concept of fairness in the context of the GME problem deals with the concurrent entering property. In its current form, the property implies that a process should be given unimpeded access to the CS as long as no other process requests a conflicting session. This allows for the situation where a process p requests a session s, and long after this, a process q, requesting a session $s' \neq s$, leaves the NCS and blocks p from entering the CS in a bounded number of its own steps. This situation is undesirable and unfair, because q did not leave the NCS until long after p.

A variant of the concurrent entering property that formalizes the above intuition is given below. This property was also defined by Jayanti et al. [10].

Strong Concurrent Entering: If a process p, requesting a session s, completes the doorway before any process q, requesting a session $s' \neq s$, starts the doorway, then p enters the CS within a bounded number of its own steps.

To recap: the most basic properties that any GME algorithm must satisfy are mutual exclusion, lockout freedom, bounded exit, and concurrent entering. The lower bound proof that we present in this thesis applies to such GME algorithms. In addition to the basic properties, we have also defined a number of fairness properties: FCFS, FIFE, and strong concurrent entering.

1.3 Models and Measures of Complexity

Our discussion here on models and measures of complexity is based on the discussion in Anderson et al. [2].

There are two major types of models under which group mutual exclusion and mutual exclusion algorithms may run. One is the message-passing model, and the other is the shared-memory model. In this thesis, we focus on shared-memory models.

The two main types of shared-memory models are the distributed shared-memory (DSM) model, and the cache-coherent (CC) model. The difference between the two models is with respect to where variables are physically stored and how processes access the variables.

Intuitively, under the DSM model, every variable is associated with exactly one process. Accessing a variable associated with another process causes the process to make a **remote memory reference**.

Under the CC model, variables are located in a global store that is not associated with any particular process. Every time a process reads a variable, it does so using a local copy (cached copy) of the variable. Whenever the cached variable is no longer valid — either because the process has never read the variable before, or because some other process overwrote it in the global store — the process makes a remote memory reference and copies the variable into its local store (i.e., it caches the variable). Also, every time a process writes a variable, the process writes the variable to the global store, which

involves making a remote memory reference. The reader is directed to Chapter 2 for a more detailed description of the two models.

An important measure of algorithmic complexity used in evaluating the performance of mutual exclusion or group mutual exclusion algorithms is the remote memory reference (RMR) complexity. This is a measure of the maximum number of remote memory references that a process makes between the time it leaves the NCS and the time it next returns to the NCS. Accessing a remote variable is much more expensive than accessing a local variable, and so the more remote memory references that are made by a process, the poorer the performance of that process.

We say that a ME or GME algorithm is **local-spin** under the DSM or CC model if the number of remote memory references that are made between the time a process leaves the NCS and the time it next returns to the NCS is bounded.

1.4 The Reduction Concept

At this point we consider whether there is any algorithmic relationship between group mutual exclusion and mutual exclusion. We can clearly use an algorithm that solves the group mutual exclusion problem to solve the mutual exclusion problem. Each process need only request a unique session (e.g., its process identifier). However, it is not so clear that one can take a mutual exclusion algorithm and somehow use it to construct a group mutual exclusion algorithm.

Making use of an algorithm that solves one problem A in the construction of an algorithm that solves another problem B is referred to as a **reduction** of B to A (or **transformation** of A to B).

It turns out that it is possible to reduce a group mutual exclusion algorithm to a FCFS abortable mutual exclusion algorithm. We explore this reduction in detail in this thesis.

1.5 Previous Work

Dijsktra [5] was the first to propose a solution to the mutual exclusion problem. However, his solution did not satisfy the lockout freedom property. Instead, it satisfied a weaker variant of this property, called deadlock freedom:

Deadlock Freedom: If a process p is stuck forever in its trying protocol, then there is some process that enters the CS infinitely often.

Knuth [13] was the first to provide a solution to the mutual exclusion problem that satisfied lockout freedom. Lamport presented his "bakery algorithm" in [14]. The "bakery algorithm" was the first mutual exclusion algorithm to satisfy the FCFS property. In addition to this, Lamport's algorithm did not assume atomic reads and writes; all earlier mutual exclusion algorithms assumed that variables could be read and written atomically, which is a form of mutual exclusion. Assuming mutual exclusion at a lower-level is theoretically undesirable when constructing a mutual exclusion algorithm.

Local-spin mutual exclusion algorithms were presented in [3] by Anderson, and [7] by Graunke and Thakkar. Both of these have $O(1)$ RMR complexity, but are local-spin only under the CC model. In [15] Mellor-Crummey and Scott presented a mutual exclusion algorithm that is local-spin under the DSM model. The RMR complexity of this algorithm is $O(1)$, but it has a drawback: the algorithm does not satisfy the bounded exit property; rather, it satisfies a weaker form of this property, called exit protocol termination:

Exit Protocol Termination: If a process p enters the exit protocol, then p eventually returns to the NCS.

Another undesirable property of all the preceding local-spin algorithms is that they make use of strong synchronization primitives, such as "fetch-and-store", "fetch-and-

add", and "compare-and-swap".[2]

Yang and Anderson [18] presented a mutual exclusion algorithm that is local-spin under the DSM model, has RMR complexity $O(\log N)$, and does not use any strong synchronization primitives — it uses only ordinary (atomic) read and write operations.

Abortable mutual exclusion algorithms were studied by Scott in [16] and Scott and Scherer in [17]. Jayanti [9] was the first to provide a FCFS abortable mutual exclusion algorithm that is local-spin under the DSM model. Jayanti's algorithm has RMR complexity $O(\min(k, \log N))$, where k is the point contention.[3]

Lower bounds on the RMR complexity for mutual exclusion algorithms were studied by Cypher [4] and by Anderson and Kim [1]. In particular, Cypher showed that the amortized lower bound for the number of remote memory references that an N-process mutual exclusion algorithm can make is $\Omega(\log \log N / \log \log \log N)$. Anderson and Kim improved on this by proving that the lower bound for the number of remote memory references a mutual exclusion algorithm can make is $\Omega(\log N / \log \log N)$. Both lower bounds apply to both the DSM and CC models, and to algorithms using atomic reads, writes, and a certain class of strong synchronization primitives, called comparison primitives, that includes "test-and-set" and "compare-and-swap" (but not primitives like "fetch-and-add").

The group mutual exclusion problem was first introduced and solved by Joung [11]. His solution does not satisfy any fairness properties, and is not local-spin. It also has the drawback that processes must request sessions from a finite set of known size.

Keane and Moir [12] almost succeeded in presenting a local-spin group mutual exclusion algorithm. Unlike Joung's algorithm, Keane and Moir's solution allowed for processes to request sessions from a set of unknown size. The problem with Keane and Moir's so-

[2] A detailed discussion of strong synchronization primitives is provided in Chapter 2.

[3] Intuitively, point contention is a measure of the number of processes that are simultaneously active at some point in the algorithm. The reader is directed to [9] and [2] for more details and references on this topic.

lution, pointed out by Hadzilacos [8], is that it does not satisfy the concurrent entering property, but rather a weaker property called the concurrent occupancy property:

Concurrent Occupancy: If a process p requests a session and no process requests a different session, then p eventually enters the CS (even if other processes do not leave the CS).

Hadzilacos [8] formally defined the concurrent entering property, and went on to define the FCFS property in the context of group mutual exclusion. He also presented a FCFS GME algorithm. His solution, like Keane and Moir's, allows for processes to request sessions from a set of unknown size, but unlike Keane and Moir's, is not local-spin.

In [10], Jayanti, Petrovic and Tan defined the FIFE property for the group mutual exclusion problem. (The FIFE property was originally defined in the context of the l-mutual exclusion by Fischer, et al. [6].) Jayanti also defined the strong concurrent entering property, and presented a group mutual exclusion algorithm that satisfies FCFS, FIFE, and strong concurrent entering. The algorithm they presented is in the form of a reduction to a FCFS abortable mutual exclusion algorithm, and allows for processes to request sessions from a set of unknown size. The algorithm is local-spin under the CC model (a fact overlooked in [10]) but not under the DSM model.

We know of no lower bounds research done for group mutual exclusion algorithms.

1.6 Contributions and Thesis Outline

We start in Chapter 2 by presenting some preliminaries, including a formal model of computation that will be used in the rest of the thesis for proving our results.

In Chapter 3, we show an $\Omega(N)$ lower bound on the RMR complexity of any N-process group mutual exclusion algorithm under the DSM model. This lower bound is strong in that it applies even if the number of sessions is known a priori and limited to two, and applies regardless of the synchronization primitives the algorithm may use.

Note that this result shows that GME is qualitatively different than ME since, as we have seen, there are ME algorithms whose RMR complexity under the DSM model is only $O(1)$ (if strong synchronization primitives are available) or only $O(\log N)$ (if only atomic read and write operations can be applied to shared memory).

In Chapters 4 - 6 we present a number of new local-spin group mutual exclusion algorithms. Each algorithm is in the form of a reduction to a FCFS abortable mutual exclusion algorithm. In particular, in Chapter 4, we first review the reduction by Jayanti et al. [10], and then present two variants of this algorithm that are local-spin under the DSM model. The first and simplest variant satisfies the FCFS and strong concurrent entering fairness properties, but not FIFE. The second variant is significantly more complex, and satisfies FCFS, strong concurrent entering, and FIFE.

An undesirable feature of the algorithms presented in Chapter 4 is that there are certain scenarios in which processes execute through the CS sequentially, even when it is possible for those processes to execute through the CS concurrently. We start Chapter 5 by explaining this problem in more detail, and then provide an algorithm that overcomes this problem at the expense of the fairness properties. This algorithm is presented in several different forms: the first variant uses COMPARE_AND_SWAP; the second variant, whose correctness we do not formally prove, uses no strong synchronization primitives but requires unbounded variables; the third variant, whose correctness we do prove, is similar to the second but uses only bounded variables.

The algorithms we present in Chapters 4 and 5 have RMR complexity $O(N)$ under the DSM model. Given the lower bound we prove in Chapter 3, it follows that these algorithms are optimal with respect to RMR complexity under the DSM model. In Chapter 6, we first present a 2-session GME algorithm that is local-spin under the CC model and has RMR complexity $O(f(N))$, where $f(N)$ is the RMR complexity of the FCFS abortable mutual exclusion algorithm used in the reduction. Given Jayanti's FCFS abortable ME algorithm in [9], which has RMR complexity $O(\min(k, \log N))$, our

2-session GME algorithm under the CC model beats the 2-session GME RMR lower bound of $\Omega(N)$ under the DSM model. This shows that the DSM and CC models are qualitatively different: it is not, in general, possible to translate an algorithm from the CC model to the DSM model, without at least some loss of efficiency (as measured by the number of remote memory references). We then use the 2-session algorithm as a building block for an M-session group mutual exclusion algorithm that is local-spin under the CC model and has RMR complexity $O(\log M \cdot f(N))$.

We conclude in Chapter 7.

Chapter 2

Preliminaries

In this chapter we outline the formal model of computation that we will use throughout this thesis, and describe the two memory models with which we are concerned. In addition, we provide some details about how our algorithms are structured and what features we make use of in our algorithms, and conclude by defining some terms and notation.

2.1 Formal Model

A **system** consists of a set of N processes and a set of shared variables (i.e., variables to which all processes in the system have access). A **process** is an automaton that has a **private state**, which is reflected by the values of a set of private variables (i.e., variables to which only that process has access). A **global state** of the system consists of a private state for each process and a value for each shared variable. An **initial private state** for a process p specifies an assignment of values to the private variables of p, and an **initial global state** for a system specifies an initial private state for each process in the system and an assignment of values to the shared variables.

Processes execute **actions** (sometimes called **steps**) based on their state. The action involves executing an operation on zero or one shared variables, possibly changing

its value, and possibly changing the private state of the process. **Operations** that can be performed on shared variables when executing an action include read, write, or read-modify-write. Read and write operations are self-explanatory; read-modify-write operations are explained in more detail below. Also, note that in executing an operation on a shared variable and changing the private state of the process, the global state is changed as well.

More formally, as part of a process i's specification, there is a function δ_i : **PRIVATE_STATE** $\mapsto$ **ACTION** that describes the action that i takes when it is in a given private state. We also define a function δ_{trans} : **STATE** $\times$ **ACTION** $\times$ **PROCESS** $\mapsto$ **STATE** that describes the new global state after a process performs an action. In particular, given a state s and an action a performed by a process p, the state $s' = \delta_{trans}(s, a, p)$ is a state that is identical to s except that the shared variable (if any) that was accessed by the action a and the private state of p are updated according to a.

A **run** (sometimes called an **execution**) of the system is an alternating sequence of global states and process identifiers that starts with an initial global state s_0: s_0, p_1, s_1, p_2, s_2, ... etc. Intuitively, a run describes an order in which processes execute actions and the sequence of states through which the system proceeds because of those actions. More formally, runs are infinite in length and must satisfy two conditions: they must be fair, and they must be valid. A **fair** run is a run in which every process executes an action infinitely often (i.e., it appears in the sequence infinitely often). (We do not consider systems in which processes "terminate". This is not a limitation since any system in which a process does "terminate" can be modeled by the process entering an empty infinite loop.) Given a global state s and process p, we use the notation $s(p)$ to denote the private state of process p contained within the global state s. A **valid** run is a run such that for each positive integer i, if $p_i = p$ and $a = \delta_p(s_{i-1}(p))$, then $s_i = \delta_{trans}(s_{i-1}, a, p)$: i.e., for each positive integer i, the triple s_{i-1}, p_i, s_i in the alternating sequence of global

states and processes that make up the run, "makes sense" according to the specification of the system.

A **point in a run** is simply an index of some element in the run.

An **algorithm** is a way of specifying the system (i.e., it is a specification of the automaton of each process in the system and the initial (global) state). A **run of an algorithm** $\mathbb{A}$ is shorthand for saying a run of the system specified by algorithm $\mathbb{A}$.

Processes execute actions atomically. At an intuitive level this means that no two processes can execute an action at the same time. This is formally captured in the preceding by the fact that executions are modeled as *sequences* of actions.

Also, processes execute actions asynchronously. At an intuitive level this means that there is a unbounded, but finite, amount of time between the time when a process executes an action and when it next executes an action. This is formally captured in the preceding by the fact that in any run, each process p must perform an infinite number of actions, yet there is no bound on the number of actions that can be performed by different processes in between a point when p performs one action and the point when p performs its next action.

The synchronization primitives that we will consider in this thesis are defined below. Shared variables are capitalized and private variables are in lower-case.

```
FETCH_AND_ADD(V1, v2 : integer) {
    old_val := V1;
    V1 := V1 + v2;
    return old_val;
}

ATOMIC_ADD(V1, v2 : integer) {
    V1 := V1 + v2;
}
```

```
COMPARE_AND_SWAP(V_1, v_2, v_3) {
    if V_1 = v_2 then
        V_1 := v_3;
        return true;
    else
        return false;
    end if
}
```

2.2 Memory Models

We use the abstraction of a **memory module** to help describe the semantics of the different memory models. Each process has exactly one memory module associated with it. Whenever a process accesses some shared variable, it does so by referencing some memory module (depending on the memory model, as described below). If a process accesses a variable by referencing its own memory module, then we say that the process makes a **local memory reference**; otherwise, we say that the process makes a **remote memory reference** (RMR).

The two memory models that we are interested in are the **distributed shared-memory (DSM) model**, and the **cache-coherent (CC) model**.

If a system operates under the DSM model, then each shared variable is located within exactly one process's memory module. For example, consider a system that consists of two processes, p and q, and two shared variables, v_p and v_q. Assume v_p is stored in the memory module of process p, and v_q is stored in the memory module of process q. When p accesses v_p or q accesses v_q, the processes make local memory references. However, when p accesses v_q or q accesses v_p, the processes make remote memory references.

If a system operates under the CC model, then we assume the existence of an additional memory module, called the global memory module. In this case, all shared variables are located in the global memory module. Intuitively, the memory modules associated with the processes act as caches. When a process p executes a read or read-modify-write operation on some shared variable v, if v is not "cached" in p's memory

module, then p accesses the global memory module and copies v into its local memory module. A variable is not cached in p's memory module when p accesses v for the first time, or when the copy of v that p has in its cache is invalid. Process p's copy of v becomes invalid when a process $q \neq p$ overwrites v in the global memory module. When a process p executes a write or a successful read-modify-write (i.e., a read-modify-write in which the variable is actually written) on some shared variable v, then p updates both its local memory module and the global memory module (and therefore invalidates any copies of that variable in other processes' memory modules). We assume that the global memory module is remote to every process, and so a process p must make a remote memory reference whenever it writes some shared variable, tries to read a shared variable for the first time, or tries to read an invalid variable.

As an example of how things work under the CC model, consider a system that consists of two processes, p and q, and a shared variable v. Assume p reads v, and then q reads v. Both p and q make remote memory references because they are reading v for the first time. Now, assume p writes v, p reads v, and then q reads v. When p writes v, it makes a remote memory reference, since it must update the global memory module. Process p's subsequent read is a local memory reference since the copy of v in p's memory module is not invalid. However, q's read of v involves a remote memory reference, because p, by writing v, invalidated q's copy of v.

We say that a ME or GME algorithm is **local-spin** under the DSM or CC model if the number of remote memory references that are made between the time a process leaves the NCS and the time it next returns to the NCS is bounded.

2.3 Algorithm Preliminaries

Each group mutual exclusion algorithm that we present in this thesis is in the form of a reduction to a FCFS abortable mutual exclusion algorithm. We use the methods

MUTEXDOORWAY(), MUTEXWAITINGROOM(), MUTEXABORT(), and MUTEXEXIT() in our GME algorithms to denote the doorway, the waiting room, the abort protocol, and the exit protocol, respectively, of the underlying FCFS abortable mutual exclusion algorithm.

In order to satisfy the mutual exclusion property, it is clearly necessary for the algorithms to have a mechanism that can prevent processes from entering the CS. The mechanism we use for this purpose is a busy-wait loop, which is essentially a loop that can be executed an unbounded number of times. The busy-wait loops that we use are fairly small and consist of checking a boolean condition to know when to advance in the algorithm. We use the notation "**await** v" as shorthand for "**while** v **do** /* */ **od**", where v is a boolean expression.

Figure 2.1 Cobegin-Coend Block

```
cobegin
    M1();
    ||
    M2();
coend
```

Our algorithms also use cobegin-coend blocks. Consider the code fragment given in Figure 2.1. M1() and M2() are referred to as co-routines. (We sometimes refer to them just as methods when it is clear from the context that they are in fact co-routines.) Each co-routine in a cobegin-coend block is separated by the "||" symbol. When a process executes a cobegin-coend block it executes the co-routines within that block in a fair manner. ("Fair" here means that in any infinite execution both coroutines take infinitely many steps.) When one of the co-routines terminates, the process automatically aborts execution of all other co-routines in that cobegin-coend block and advances in the algorithm.

Finally, every process has a private variable called *mysession*. A process sets this variable before leaving the NCS, and no process ever sets *mysession* to 0 when leaving the NCS. A process indicates that it is not presently requesting a session by setting

mysession to 0.

2.4 Definitions of Common Terms and Notation

We define some terms that are used throughout the thesis.

Invocation: An **invocation** of a group mutual exclusion algorithm by some process p refers to an interval of time delimited by p's leaving the NCS and its subsequent return.

Active: We say that a process is **active** when it is outside of the NCS.

Doorway-Precede: Process p **doorway-precedes** another process q iff p finishes executing the doorway before q begins executing the doorway.

Doorway-Concurrent: Process p is **doorway-concurrent** with process q iff neither process doorway-precedes the other.

Note that in the preceding definitions to be strictly precise we ought to refer to the invocations that the processes are executing, rather than the processes themselves. For example, we should be saying that a particular invocation of p doorway-precedes (or is doorway-concurrent with) a particular invocation of q. However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run.

In the proofs of correctness of the algorithms, we use the following notation:

We say that p@i is true iff process p is about to execute line i. Also, p@{i..j} is true iff p is about to execute any line in the range i to j.

We say that a process p is **enabled to execute** some lines i..j iff p@{i..j}.

The notation $p.v$ denotes process p's private variable v.

The underlying FCFS abortable mutual exclusion algorithm is often referred to for simplicity as algorithm $\mathbb{A}$. For example, when we write "due to the FCFS property of $\mathbb{A}$," we are referring to the FCFS property of the underlying FCFS abortable mutual exclusion algorithm.

For $s \in \{0,1\}$, $\overline{s} = 1 - s$.

Chapter 3

GME Lower Bound Under the DSM Model

In this chapter we prove that under the DSM model any GME algorithm (i.e., an algorithm satisfying mutual exclusion, lockout freedom, bounded exit, and concurrent entering) must perform $\Omega(N)$ RMRs per invocation in the worst case. This lower bound holds even for the 2-session GME problem (i.e., when each process can request one of only two sessions), and even if processes can access shared variables using strong synchronization primitives.

We first give a proof sketch of the lower bound in order to convey some intuition, and follow this by presenting the formal proof.

3.1 Proof Sketch of Lower Bound

In the proof sketch we make the simplifying assumption that when a process busy-waits, it does so by spinning on a single variable in the process's local memory module. For example, a process might busy-wait on a boolean variable b by executing the statement **await** $\neg b$. Typically local-spin algorithms tend to have busy-wait loops that are of this form, or very similar. In fact, all the algorithms that we present in this thesis, with the

exception of Jayanti's algorithm, have busy-wait loops of this form. However, to make our result as general as possible, this assumption is not made in the formal version of the proof.

Theorem 3.1: Let $\mathbb{A}_{DSM}$ be a DSM local-spin algorithm for 2-session GME. Algorithm $\mathbb{A}_{DSM}$ must perform $\Omega(N)$ RMRs per invocation in the worst case.

Proof sketch: Without loss of generality, let s and $s' \neq s$ be the two sessions that processes executing $\mathbb{A}_{DSM}$ are allowed to request. Consider the following run: some process p requests session s, while no other process is active. Process p enters the CS (because $\mathbb{A}_{DSM}$ satisfies the lockout freedom property), after which point $N-1$ other processes enter the trying protocol, each requesting session s'. Since p is in the CS, and $\mathbb{A}_{DSM}$ satisfies the mutual exclusion property, none of the $N-1$ processes requesting session s' can enter the CS. There is no bound on the amount of time that p can spend in the CS, so we assume that p is in the CS until the $N-1$ processes in the trying protocol enter a busy-wait loop. Since $\mathbb{A}_{DSM}$ is an algorithm that is local-spin under the DSM model, this implies that there are at least $N-1$ distinct variables on which each process in the trying protocol is busy-waiting. For any such process to enter the CS, the local-spin variable it is busy-waiting on must be updated. Now, consider what happens when p leaves the CS and executes the exit protocol. Process p must update at least $N-1$ variables. If it updates any less, then we can complete the construction of this run in such a way that there will be at least one process in the trying protocol that does not enter the CS in a bounded number of its own steps after p is no longer active. This violates the concurrent entering property that $\mathbb{A}_{DSM}$ satisfies by assumption. □

3.2 Lower Bound Proof

Proof of Theorem 3.1:

As in the proof sketch, without loss of generality, let s and $s' \neq s$ be the two sessions

that processes executing $\mathbb{A}_{DSM}$ are allowed to request.

We want to construct a run of $\mathbb{A}_{DSM}$ such that $\Omega(N)$ remote memory references are made by some process during an invocation of $\mathbb{A}_{DSM}$. We call this our **goal run**. In constructing such a run, at different steps of our construction, we will admit "pseudo-runs" that are not necessarily fair (i.e., runs in which all processes execute steps infinitely often). We will show that certain properties must hold in these "pseudo-runs" by constructing proper runs (i.e., runs that are fair and valid) from them. This will eventually lead to the construction of our goal run, which is both fair and valid.

We begin by constructing a pseudo-run β, similar to the way that was done in the proof sketch, as follows:

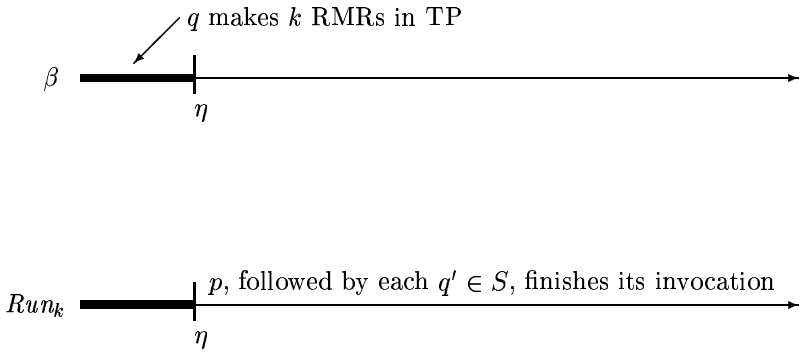
Assume a process p requests session s, and let S be the set of the remaining $N-1$ processes that request session s'. The processes in $S \cup \{p\}$ execute in the following order: p leaves the NCS, enters the trying protocol, and then enters the CS. While p is in the CS, all processes in S leave the NCS and enter the trying protocol. Then, $\forall q \in S$, q continually executes steps of $\mathbb{A}_{DSM}$. $\mathbb{A}_{DSM}$ is a correct GME algorithm, and so by the mutual exclusion property, while p is in the CS, no process in S can enter the CS. So $\forall q \in S$, q executes steps inside the trying protocol of $\mathbb{A}_{DSM}$. (Note that β is not fair because p takes finitely many steps.)

In the proof sketch, we assumed that there was a point after which all processes in S must enter a loop where each process busy-waits on some variable in its local memory module. Here, instead, we prove the more general claim that there is a point in β after which no process in S makes any remote memory references.

Claim 3.1.1: $\forall q \in S$, there is a point in β after which q makes no remote memory references.

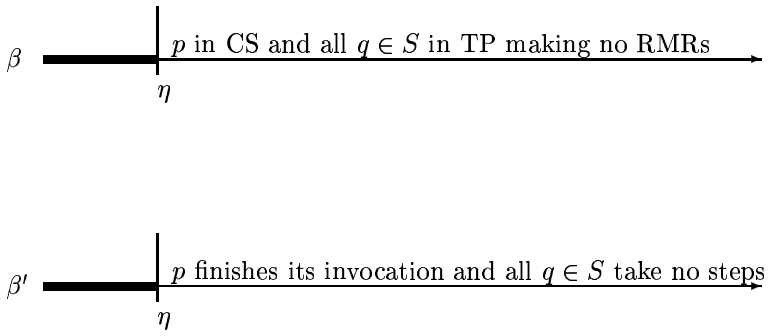
Proof: Suppose, by way of contradiction, that the claim is false. By this assumption, and the fact that every process in S continually executes steps in the trying protocol, there exists some $q \in S$ such that q makes a remote memory reference in the trying

protocol infinitely often. From this it follows that for any positive integer k, there is a point η in β such that q makes at least k remote memory references in the trying protocol before point η. We construct a run of $\mathbb{A}_{DSM}$, Run_k, identical to β up to point η, after which the following happen, in order: p leaves the CS, executes the exit protocol, and returns to the NCS, followed by all processes in S finishing execution of the trying protocol, executing passage through the CS, executing the exit protocol, and returning to the NCS. Runs β and Run_k are illustrated by the following diagram (note, in this diagram, and in the diagrams used elsewhere in this proof, thick lines represent where the runs are equivalent, and thin lines represent where the runs diverge):



Hence, $\forall k$, there is a run of $\mathbb{A}_{DSM}$, Run_k, in which some process makes at least k remote memory references in the trying protocol in a single invocation. This implies that the number of remote memory references per invocation is unbounded, contradicting that $\mathbb{A}_{DSM}$ is a local-spin algorithm. □ (Claim 3.1.1)

By Claim 3.1.1 there is a point η in pseudo-run β after which no process in S makes any remote memory references. We create a new pseudo-run, β', which is identical to β up to and including point η, after which the processes in S execute no steps, and p leaves the CS, executes the exit protocol, and returns to the NCS. We illustrate the two runs in the diagram below:

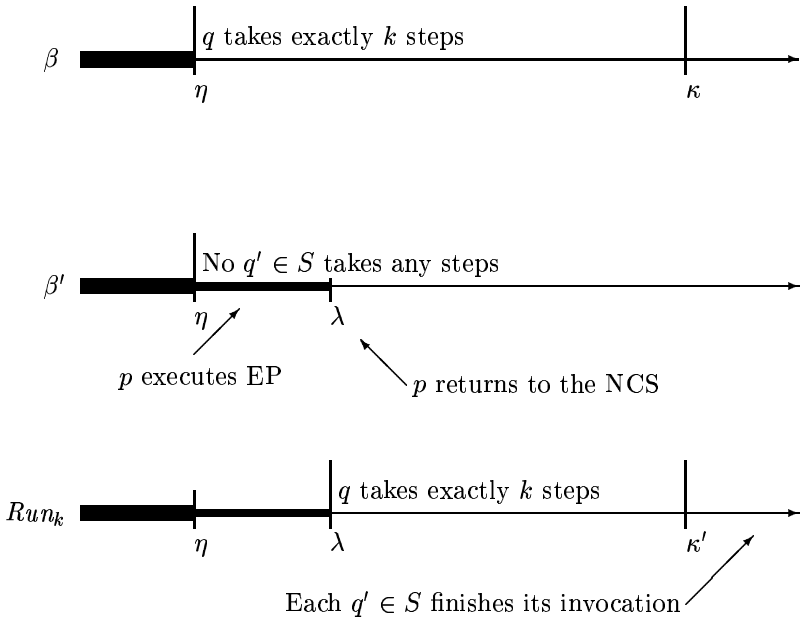


In the following claim we will show that in β', p makes $\Omega(N)$ remote memory references in the exit protocol. Once we have shown this, we will be able to construct the goal run from β' by simply letting all processes in S finish executing their invocation after p has finished the exit protocol.

Claim 3.1.2: In β', p makes at least $N-1$ remote memory references in the exit protocol.

Proof: Suppose, by way of contradiction, that p makes fewer than $N-1$ remote memory references in the exit protocol. By this assumption, the fact that processes are executing under the DSM model, and the fact that there are $N-1$ processes in S, it follows that there exists a process $q \in S$, such that in β', p does not reference any variable local to q during p's execution of the exit protocol. Now, consider some positive integer k. We construct a run of $\mathbb{A}_{DSM}$, Run_k[1], identical to β' up to and including the point λ where p returns to the NCS, after which the following occur, in order: q executes exactly k steps, and each process in S finishes executing the trying protocol, executes passage through the CS, executes the exit protocol, and returns to the NCS. Let κ be a point in β such that q executes exactly k steps in β between point η and κ; and let κ' be the point in Run_k after which q has finished executing the said k steps that it must execute but before the other processes in S start executing steps again. We illustrate the relationship between the different runs in the diagram below:

[1]Note that Run_k in this claim has no relation to Run_k from Claim 3.1.1



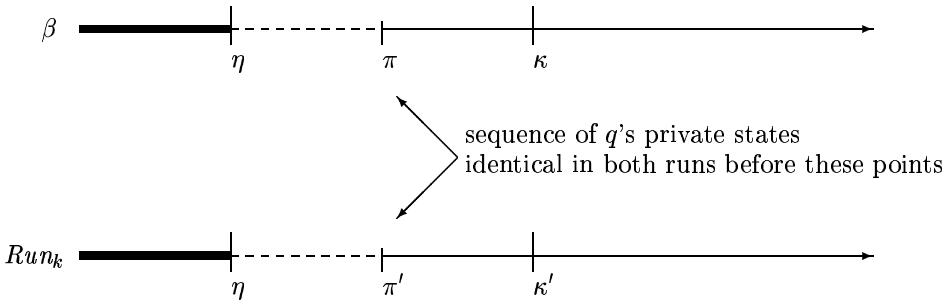
Before we can state our next sub-claim, we first need to define what we mean by a process's **sequence of private states** in a run. As defined in the formal model, a run is just an alternating sequence of global states and process identifiers that starts from some global state and satisfies the fairness and validity conditions. A process r's sequence of private states in a run is simply the sequence of global states that appear in the run, restricted to the private states of process r.

It turns out that q's sequence of private states is identical in both β and Run_k up to points κ and κ', respectively. This is formally proven in the next sub-claim.

Sub-Claim 3.1.2.1: The sequence of q's private states in β up to point κ is identical to the sequence of q's privates states in Run_k up to point κ'.

Proof: Suppose, by way of contradiction, that the sub-claim is false. Let π and π' be the points in β and Run_k, respectively, such that for the point $\pi - 1$ immediately before π, and for the point $\pi' - 1$ immediately before π', the sequence of q's private states in β up to point $\pi - 1$ is identical to the sequence of q's private states in Run_k up to point

$\pi' - 1$, but the sequence of q's private states in β up to point π is *not* identical to the sequence of q's private states in Run_k up to point π'. By the way Run_k is constructed from β', and, in turn, β, it follows that Run_k and β are the same up to and including point η, so π and π' can only occur after point η. Moreover, by the assumption that the sub-claim is false, π and π' must be before (or be identical to) κ and κ', respectively. This is illustrated by the following diagram. (Note, in this diagram, dashed lines are used to represent where the sequence of q's private states are identical in the two runs exclusive of the region where the two runs are identical. Thick lines still represent where the two runs are identical. Hence, the region covered by the thick lines and the dashed lines together represent where the sequence of q's private states are identical in the two runs.)



Now, consider the reason why q's private state in β and Run_k diverges at π and π'. Since $\mathbb{A}_{DSM}$ is deterministic, and both runs start from the same initial state, it follows that q's private state diverges in the two runs because q reads (either through a read operation, or the read portion of a read-modify-write operation) some variable v, and the value q reads is different in the two runs. We will refer to this read in the following as the **diverging read**. Since the value q reads from v is different in the two runs, it follows that there must be a last write to variable v by some process in either β or Run_k before the diverging read (i.e., before point π or π', respectively). Since β and Run_k are identical up to point η, the last write must occur after point η. We further claim

that this last write must be by a process other than q. To see this, suppose, by way of contradiction, that q performs the last write to v in one of the runs. From this, and the fact that q's sequence of private states in β and Run_k are identical prior to the diverging read, it must be the case that q's last write to v, before the diverging read, sets v to the same value in both β and Run_k. This contradicts that the value q reads from v in β is different from the value q reads from v in Run_k when it performs the diverging read.

Consider the cases:

Case 1: A last write to v occurs in β between points η and π.

Since q is in the trying protocol in β after point η, and q performs the diverging read of variable v in β after point η, it follows that v cannot be referenced in the CS or NCS. Hence, since p is in the CS in β after point η, it follows that p is not the process that performs the last write to v. This, and the fact that q does not perform the last write, imply that for some process $q' \in S \setminus \{q\}$, q' is the process that performs the last write. Now, by assumption, no process in S makes a remote memory reference in β after point η, and so v is local to q'. This implies that q accesses q''s memory module when q performs the diverging read of v in β after point η, contradicting that q does not make any remote memory references in β after point η.

Case 2: A last write to v occurs in Run_k between points η and π'.

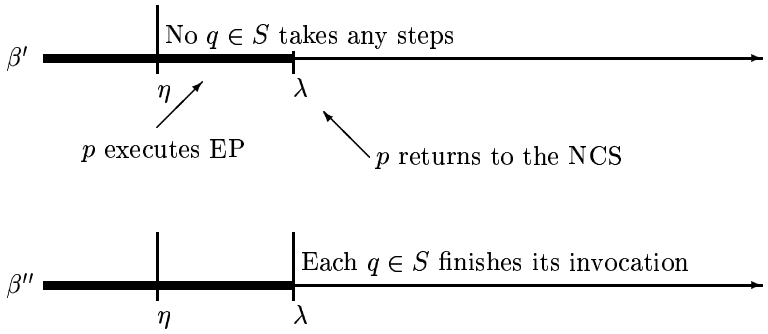
By our construction of Run_k, no process in $S \setminus \{q\}$ executes any steps in Run_k between points η and κ'. This, and the fact that the process q does not perform the last write, imply that p must be the process that performs the last write.

Also by our construction of Run_k, any steps that p executes in Run_k between points η and π' are part of the exit protocol. From this, the fact that p does not write any variable local to q when it executes the exit protocol, and the fact that p performs the last write to v in Run_k between points η and π', it follows that v is stored in the memory module of a process other than q. Thus q makes a remote memory reference when it performs the diverging read in β (which occurs after point η), contradicting that no process in S

makes a remote memory reference in β after point η. $\Box$ (Sub-Claim 3.1.2.1)

By Sub-Claim 3.1.2.1, and the fact that q does not enter the CS in β, q does not enter the CS in Run_k before point κ'. Hence, $\forall k$, there is a run of $\mathbb{A}_{DSM}$, Run_k, in which some process executes k steps in the trying protocol while there is no process requesting a conflicting session. This implies that the number of steps a process executes in the trying protocol, while no other process is requesting a conflicting session, is unbounded, contradicting that $\mathbb{A}_{DSM}$ satisfies concurrent entering. $\Box$ (Claim 3.1.2)

We now create a new run β'' that is identical to β' up to the point λ where p returns to the NCS, after which all processes in S finish executing the trying protocol, execute passage through the CS, execute the exit protocol, and finally return to the NCS. This new run is illustrated in the diagram below.



By Claim 3.1.2, p makes at least $N-1$ remote memory references in the exit protocol in run β', and so p must also do so in β''. Hence β'' is our goal run in which some process makes at least $N-1$ remote memory references in an invocation. From this it follows that $\mathbb{A}_{DSM}$ must perform $\Omega(N)$ remote memory references per invocation in the worst case. $\Box$

Corollary 3.2: Let $\mathbb{A}_{DSM}$ be a DSM local-spin algorithm for GME (for any number of sessions). Algorithm $\mathbb{A}_{DSM}$ must perform $\Omega(N)$ RMRs per invocation in the worst case.

Proof: Follows immediately from Theorem 3.1. $\Box$

Chapter 4

Fair Local-Spin GME

We start by reviewing Jayanti's [10] algorithm. This algorithm satisfies all the regular group mutual exclusion properties, and also satisfies the stronger fairness properties: FCFS, FIFE, and strong concurrent entering. We also discuss a drawback to this algorithm: the fact that it is not local-spin under the DSM model.

After reviewing Jayanti's algorithm, we present two versions of the algorithm that are local-spin under the DSM model. Our first version attempts to change Jayanti's algorithm as little as possible. In doing this, however, we end up with an algorithm that does not satisfy the FIFE property. To achieve FIFE in addition to the other properties, we need to develop some more complicated machinery. We do this in the second algorithm we present. Both algorithms have RMR complexity $O(N)$. In view of Corollary 3.2, this is the best possible RMR complexity (within a constant factor).

4.1 Jayanti's Algorithm

Jayanti's group mutual exclusion algorithm [10] is shown in Figure 4.1.[1] Jayanti's algorithm is in the form of a reduction to a FCFS abortable mutual exclusion algorithm, and satisfies the FCFS, FIFE, and strong concurrent entering properties. The algorithm conforms to the structure presented in Figure 1.2: the doorway consists of lines 3 to 11, the waiting room consists of lines 12 to 16, the CS is at line 17, and the exit protocol consists of lines 18 to line 26.

The algorithm works as follows: when a process p enters the doorway, it declares its intention to enter the CS by setting its copy of the *session* variable (i.e., *session*[p]) to the session that it is requesting (line 3). It then proceeds to execute the doorway of the underlying FCFS abortable mutual exclusion algorithm (MUTEXDOORWAY()) used in the reduction (line 4). After this, it checks to see if there are any processes requesting a conflicting session and sets the p^{th} row of *barricade* to *true* (lines 5..11). When this is done, p has completed the doorway. When p enters the waiting room it executes two co-routines concurrently: one is the waiting room of the underlying FCFS abortable mutual exclusion algorithm (MUTEXWAITINGROOM()), and the other is the method CONFLICT-FREE(). Process p completes the waiting room (of the GME algorithm) and enters the CS when one of the preceding co-routines terminates. (When the MUTEXWAITINGROOM() co-routine terminates, we say that the process is **mutex qualified**, and when the CONFLICT-FREE() co-routine terminates, we say that the process is **conflict-free qualified**.) If p detected no other processes requesting a conflicting session when it executed the doorway, then p immediately becomes conflict-free qualified because *conflict_set* $= \emptyset$. Otherwise, if the process detected conflict, the process will be blocked in the waiting room until the earliest time it can enter the CS without violating

[1] Jayanti presents two reductions in [10]; the algorithm in Figure 4.1 corresponds to his "bounded reduction". Also, we have changed the variable names to be more descriptive: in [10], *barricade* is *flag* and *conflict_set* is S_i.

the mutual exclusion property.

Figure 4.1 Jayanti's Algorithm; Process $p \in \{1..N\}$

```
shared variables:
    session: array[1..N] of integer
    barricade: array[1..N][1..N] of boolean

private variables:
    conflict_set: set of integer
    mysession: integer /* Session that p wants to attend; set before leaving NCS */
 1: loop
 2:    NCS
 3:    session[p] := mysession;
 4:    MutexDoorway();
 5:    conflict_set := ∅;
 6:    for j = 1 to N do
 7:       barricade[p][j] = true;
 8:       if session[j] ∉ {0, mysession} then
 9:          conflict_set := conflict_set ∪ {j};
10:       end if
11:    end for
12:    cobegin
13:          MutexWaitingRoom();
14:          ||
15:          conflict-free(); /* defined in Figure 4.2 */
16:    coend /* when one co-routine terminates, go to the next line */
17:    CS
18:    if mutex qualified then
19:       MutexExit();
20:    else
21:       MutexAbort();
22:    end if
23:    session[p] := 0;
24:    for j = 1 to N do
25:       barricade[j][p] := false;
26:    end for
27: end loop
```

Figure 4.2 Method conflict-free() for Algorithm in Figure 4.1

```
 1: repeat
 2:    for each j ∈ conflict_set do
 3:       if session[j] ∈ {0, mysession} then
 4:          conflict_set = conflict_set − {j};
 5:       end if
 6:       if barricade[p][j] = false then
 7:          conflict_set = conflict_set − {j};
 8:       end if
 9:    end for
10: until conflict_set = ∅
```

After p executes through the CS it executes one of MutexExit() or MutexAbort()

(lines 18..22). At this point one can see the reason why the FCFS mutual exclusion algorithm used in the reduction needs to be abortable. When p finished the waiting room, it did so because either MUTEXWAITINGROOM() or CONFLICT-FREE() terminated. If MUTEXWAITINGROOM() terminated, then the method MUTEXEXIT() executes after p has finished executing through the CS. Otherwise, if CONFLICT-FREE() terminated, then the execution of MUTEXWAITINGROOM() was aborted and MUTEXABORT() executes after p has finished executing through the CS. After this, p declares that it is no longer requesting entry into the CS by resetting its *session* variable (line 23), and then setting the p^{th} column of *barricade* to *false* for each process (lines 24..26). By resetting the *session* variable, any processes that execute through the doorway at some future point will no longer see p as a process that is potentially requesting a conflicting session. Furthermore, by resetting the *session* variable and by setting the p^{th} column of *barricade* to *false*, any other process that detected that p was requesting a conflicting session will now be able to remove p from its *conflict_set* (see lines 4 and 7 of CONFLICT-FREE()).

The reader is directed to [10] for a formal proof of correctness.

The drawback of Jayanti's algorithm is that it is not local-spin under the DSM model. This is due to the way the co-routine CONFLICT-FREE() is implemented. Under the DSM model, each *session* variable can be associated with at most one process. Since, in the CONFLICT-FREE() method, every process accesses every *session* variable an unbounded number of times, an arbitrary number of remote memory references can occur.

However, the algorithm is local-spin under the CC model, despite a statement to the contrary in [10]. Recall that under the CC model a process makes a remote memory reference reading a variable when it reads that variable for the first time or when the process's copy of the variable is invalid. (A process p's copy of a variable becomes invalid when another process writes to the variable after the last time p read the variable.)

Theorem 4.1: Jayanti's [10] algorithm is local-spin under the CC model.

Proof: By inspection we see that the only place in the algorithm where an unbounded

number of remote memory references can occur is in the method CONFLICT-FREE(). Thus, it suffices to show that the number of remote memory references that can occur in this method, due to a process reading the *session* and *barricade* variables, is bounded. This is done in the next two claims.

Claim 4.1.1: In a given invocation by a process p, for each process j, p makes at most three remote memory references to variable $session[j]$ while executing the method CONFLICT-FREE().

Proof: Suppose, by way of contradiction, that for some j, process p makes at least four remote memory references to variable $session[j]$ during its execution of the method CONFLICT-FREE(). Of these four remote memory references, the first may have been required to initially load the value of $session[j]$ into p's cache, but the remaining (at least three) were caused by invalidations due to writes into $session[j]$. Only process j writes into $session[j]$. So, while p was in CONFLICT-FREE(), i.e., after p finished the GME doorway, j wrote into $session[j]$ at least three times. Since j writes into $session[j]$ only twice during each invocation (in lines 3 and line 23), it follows that j executes an invocation in which it starts the GME doorway after p has finished the GME doorway, and enters the CS while p is still in the GME waiting room. This contradicts the FCFS property of the algorithm. □ (Claim 4.1.1)

Claim 4.1.2: In a given invocation by a process p, for each process j, p makes at most two remote memory references to variable $barricade[p][j]$ while executing the method CONFLICT-FREE().

Proof: Suppose, by way of contradiction, that for some j, process p makes at least three remote memory references to variable $barricade[p][j]$ during its execution of the method CONFLICT-FREE(). Only the first of these remote memory references is due to the variable being read for the first time; all the others must be because p's copy of the variable is invalid. Moreover, by inspecting the algorithm, we see that the only other process besides p that writes to $barricade[p][j]$ is process j when it executes

line 25. From this and the fact that p makes at least two remote memory references while executing CONFLICT-FREE() due to the variable $barricade[p][j]$ being invalid, it follows that j must write to $barricade[p][j]$ at least twice while p executes the method CONFLICT-FREE(). Since j writes *false* to $barricade[p][j]$ at line 25, when p makes the second remote memory reference reading $barricade[p][j]$, $barricade[p][j] = false$. But when p reads $barricade[p][j] = false$, it removes j from *conflict_set* and does not read $barricade[p][j]$ in any future iteration of the method CONFLICT-FREE() until the next invocation of the algorithm. This, in turn, implies that p does not make more than two remote memory references to $barricade[p][j]$, contradicting the assumption that p makes at least three such remote memory references. □ (Claim 4.1.2)

□

4.2 A FCFS Local-Spin GME Algorithm

As already discussed, Jayanti's algorithm is not local-spin under the DSM model. The reason is the unbounded number of remote memory references that can occur during a process's execution of the method CONFLICT-FREE(), which arises from the fact that each *session* variable can be stored in the memory module of at most one process.

As a naive first approach to converting the algorithm into one that is local-spin under the DSM model, we consider the effect of removing the entire "**if**" clause starting at line 3 of the CONFLICT-FREE() method in Figure 4.2. This eliminates the problem of unbounded remote memory references due to reads of the *session* variable. Moreover, $\forall p \geq 1$, $\forall j \geq 1$, we can clearly place the variable $barricade[p][j]$ in the memory module of process p. This makes all the references in the method CONFLICT-FREE() local. Now, notice that for any process $p \geq 1$, once a process j has set the variable $barricade[p][j] = false$, process p removes j from $p.conflict\_set$. In other words, there is no point for waiting until $conflict\_set = \emptyset$ any more; we need only wait until each $barricade[p][j]$ variable is

set to *false*.

It turns out that the preceding naive first approach works quite well. The algorithm in Figure 4.4 is the result of following this approach, and the updated CONFLICT-FREE() method is in Figure 4.5. The algorithm has RMR complexity $O(N)$ under the DSM model. The only drawback that this algorithm has is that it does not satisfy FIFE.

We proceed as follows: we give some definitions relative to the algorithm, describe why the algorithm violates FIFE, give a proof of correctness, and then give a proof for why the algorithm has RMR complexity $O(N)$. (As this algorithm is a variant of Jayanti's algorithm, the correctness proof has certain similarities to the correctness proof given in [10].)

4.2.1 Definitions

GME Doorway: The doorway consists of lines 3..11.

GME Waiting Room: The waiting room consists of lines 12..16

GME Exit Protocol: The exit protocol consists of lines 18..26

Mutex Qualified Process: Process p is **mutex qualified** iff $p@\{17..22\}$ and p reached line 17 because it completed execution of the method MUTEXWAITINGROOM().

Conflict-free Qualified Process: Process p is **conflict-free qualified** iff $p@\{17..18\}$ and p reached line 17 because it completed execution of the method CONFLICT-FREE().

As with the definitions of "doorway-precedes" and "doorway-concurrent" (see Section 2.4, page 21), in the preceding definitions to be strictly precise we ought to refer to the invocations that the processes are executing, rather than the processes themselves. For example, we should be saying that a particular invocation of p is mutex qualified (or conflict-free qualified). However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run.

Figure 4.3 Header for Algorithm in Figure 4.4

```
shared variables:
    session: array[1..N] of integer
    barricade: array[1..N][1..N] of boolean

/* barricade[p][j] is stored in memory module for process p for all j. */

private variables:
    conflict_set: set of integer
    mysession: integer /* Session that p wants to attend; set before leaving NCS */
```

Figure 4.4 A FCFS Local-Spin GME Algorithm; Process $p \in \{1..N\}$

```
 1: loop
 2:    NCS
 3:    session[p] := mysession;
 4:    MutexDoorway();
 5:    conflict_set := ∅;
 6:    for j = 1 to N do
 7:       barricade[p][j] = true;
 8:       if session[j] ∉ {0, mysession} then
 9:          conflict_set := conflict_set ∪ {j};
10:       end if
11:    end for
12:    cobegin
13:       MutexWaitingRoom();
14:       ||
15:       conflict-free(); /* defined in Figure 4.5 */
16:    coend /* when one co-routine terminates, go to the next line */
17:    CS
18:    session[p] := 0;
19:    for j = 1 to N do
20:       barricade[j][p] := false;
21:    end for
22:    if mutex qualified then
23:       MutexExit();
24:    else
25:       MutexAbort();
26:    end if
27: end loop
```

Figure 4.5 Method conflict-free() for Algorithms in Figures 4.4, 4.7

```
1: for each j ∈ conflict_set do
2:    await ¬barricade[p][j];
3: end for
```

4.2.2 FIFE Violation

The following describes a scenario in which FIFE is violated by the algorithm in Figure 4.4:

Let p and q be processes both requesting a session s, and let r be a process requesting a session $s' \neq s$. Assume r enters the CS. Process p then leaves the NCS and executes the GME doorway in its entirety. Process p adds r to $p.conflict\_set$ and blocks on $barricade[p][r]$ in the method CONFLICT-FREE(). Process r then executes the first line of the GME exit protocol, setting $session[r] = 0$. Process q then leaves the NCS and executes the GME doorway in its entirety. Note that p doorway-preceded q. In executing the GME doorway, q sees $session[r] = 0$, and hence does not add r to its $conflict\_set$. Since q's $conflict\_set$ is empty, it finishes executing the method CONFLICT-FREE() and enters the CS. However, p is still blocked on $barricade[p][r]$ since r has not executed any more lines of the GME exit protocol. Process p cannot proceed into the CS until after r executes the loop where $barricade[p][r]$ is assigned *false*. Thus p will not be able to enter the CS in a bounded number of its own steps, even after q is in the CS, violating the FIFE property.

4.2.3 Proof of Correctness

We start by proving an elementary lemma, followed by a corollary. The corollary will be used in proving that the algorithm satisfies the required properties.

Lemma 4.2: Suppose a process p assigns *true* to $barricade[p][q]$ at some time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $session[q] \neq 0$ is invariant in the interval $[t_1, t_2]$, then $barricade[p][q] = true$ is invariant in interval $[t_1, t_2]$.

Proof: Assume $session[q] \neq 0$ is invariant in the interval $[t_1, t_2]$. Then clearly $q@\{4..18\}$ is invariant in the interval $[t_1, t_2]$, and so q executes no part of its exit protocol in $[t_1, t_2]$. Since the only place where *false* can be assigned to $barricade[p][q]$ is in q's exit

protocol, at line 20, it must be the case that *false* is never assigned to $barricade[p][q]$ in $[t_1, t_2]$. Hence, because *true* is assigned to $barricade[p][q]$ at time t_1, it follows that $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. □

The following corollary gives a necessary condition for a process to become conflict-free qualified in a certain interval.

Corollary 4.3: Suppose a process p, requesting a session s_p, assigns *true* to $barricade[p][q]$ at line 7 at some time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $session[q] \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, then p cannot be conflict-free qualified in the interval $[t_1, t_2]$.

Proof: Assume $session[q] \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$. Suppose, by way of contradiction, that p is conflict-free qualified at some point in the interval $[t_1, t_2]$. Thus p finishes executing the method CONFLICT-FREE() by time t_2. Now, by our assumption that $session[q] \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 8 to true and thus adds q to $p.conflict\_set$ at line 9, for iteration $j = q$. Thus p busy-waits on $barricade[p][q]$ in the method CONFLICT-FREE() until it equals *false*. By Lemma 4.2, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Hence the method CONFLICT-FREE() cannot finish by time t_2, which contradicts the fact it does finish by time t_2. □

Mutual Exclusion

Theorem 4.4: The algorithm in Figure 4.4 satisfies the mutual exclusion property.

Proof: Let p be a process that requests session s_p and q be a process that requests session s_q, where $s_p \neq s_q$. Suppose, by way of contradiction, that p and q are in the CS simultaneously at time t. At most one process in the CS is mutex qualified, so assume, without loss of generality, that q is conflict-free qualified.

Claim 4.4.1: Process q begins the loop at line 6 before p executes line 3.

Proof: Let t_q be the time when q writes *true* to $barricade[q][p]$ at line 7. Now,

suppose, by way of contradiction, that the claim is false. Then $session[p] = s_p$ throughout the interval $[t_q, t]$. Hence, by Corollary 4.3, q cannot be conflict-free qualified in the interval $[t_q, t]$, which contradicts that q is in the CS conflict-free qualified at time t. □ (Claim 4.4.1)

Either p enters the CS mutex qualified, or it enters the CS conflict-free qualified. We consider these two cases, in turn:

Case 1: Process p is mutex qualified.

By Claim 4.4.1, q starts executing the loop at line 6 before p executes line 3, and so q will finish executing the method MUTEXDOORWAY() at line 4 before p starts executing the method MUTEXDOORWAY() at line 4. Hence, due to the FCFS property of $\mathbb{A}$, p cannot become mutex qualified until q executes the method MUTEXABORT() at line 25, contradicting the assumption that p and q are in the CS simultaneously at time t.

Case 2: Process p is conflict-free qualified.

By Claim 4.4.1, q starts executing the loop at line 6 before p executes line 3, and so q executes line 3 before p does. Since p and q are both conflict-free qualified, symmetrically, p executes line 3 before q does. This is a contradiction.

For p to be in the CS at time t, it must either be mutex qualified or conflict-free qualified. The preceding two cases contradict this. Hence mutual exclusion holds. □

Strong Concurrent Entering

Theorem 4.5: The algorithm in Figure 4.4 satisfies the strong concurrent entering property.

Proof: Let p be a process requesting session s_p that is in its GME waiting room at time t_1. Assume that at t_1 every active process that doorway-precedes or is doorway-concurrent with p requests session s_p. Now, let t_2 be the time when p starts its next invocation, if it exists. If there is no next invocation, then $t_2 = \infty$. We want to show that p will enter the CS in its current invocation in a bounded number of its own steps.

For this it suffices to show that $\forall j \in p.conflict\_set$, $barricade[p][j] = false$ during the interval $[t_1, t_2)$. Suppose that $q \in p.conflict\_set$. Then at line 8 at some time $t_q < t_1$, p read $session[q] \notin \{0, s_p\}$. This implies that at time t_q, $q@\{4..18\}$. This, in turn, implies that q doorway-precedes or is doorway-concurrent with p. By assumption, q is not active at time t_1, and so at some time in (t_q, t_1), q will execute the loop at line 19, assigning *false* to $barricade[p][q]$. Now, the only process that can set $barricade[p][q]$ to *true* is process p. Process p's last assignment of *true* to $barricade[p][q]$ occurred before t_q, and its next assignment occurs after t_2. It follows that $barricade[p][q] = false$ in $[t_1, t_2)$, as required. □

FCFS

Theorem 4.6: The algorithm in Figure 4.4 satisfies the FCFS property.

Proof: Assume p requests session s_p, q requests session s_q, $s_p \neq s_q$, and p doorway-precedes q. We must prove that q does not enter the CS before p enters the CS. Suppose, by way of contradiction, that it does. There are two cases, depending on how q enters the CS.

Case 1: Process q enters the CS mutex qualified before p.

Since p doorway-precedes q, p finishes executing the method MUTEXDOORWAY() at line 4 before q begins executing the method MUTEXDOORWAY() at line 4. Hence, by the FCFS property of $\mathbb{A}$, q cannot become mutex qualified until p executes one of MUTEXEXIT() or MUTEXABORT(). This contradicts the assumption that q enters the CS mutex qualified before p.

Case 2: Process q enters the CS conflict-free qualified before p.

Let t_q be the time when q assigns *true* to $barricade[q][p]$ at line 7, and let t_p be the time when p enters the CS. Since p doorway-precedes q, $session[p] \notin \{0, s_q\}$ is invariant in the interval $[t_q, t_p]$. By Corollary 4.3, q cannot be conflict-free qualified in the interval $[t_q, t_p]$. This implies that q cannot enter the CS conflict-free qualified before p, which

contradicts the assumption that it does. □

Lockout Freedom

Theorem 4.7: The algorithm in Figure 4.4 satisfies the lockout freedom property.

Proof: Suppose, by way of contradiction, that the algorithm does not satisfy the lockout freedom property. This means that there is an execution in which some process gets stuck forever in the GME waiting room when it tries to enter the CS. The following claim proves that this is impossible.

Claim 4.7.1: No process can be stuck forever in the method MUTEXWAITINGROOM().

Proof: $\mathbb{A}$ is an abortable mutual exclusion algorithm that satisfies the lockout freedom property. Hence the only way for a process to get stuck forever in MUTEXWAITINGROOM() is if some other process becomes mutex qualified and then does not execute the method MUTEXEXIT(); or, if some process starts executing MUTEXWAITINGROOM(), aborts execution of it, but then does not execute the method MUTEXABORT(). If a process becomes mutex qualified, then we see by examining the algorithm that it will eventually execute the method MUTEXEXIT(). Furthermore, the only way a process will abort execution of MUTEXWAITINGROOM() once started is if the process becomes conflict-free qualified. In this case, again by examining the algorithm, we see that such a process will eventually execute the method MUTEXABORT(). Hence no process can be stuck forever in the method MUTEXWAITINGROOM(). □ (Claim 4.7.1)

Claim 4.7.1 contradicts the fact that there is an execution in which some process is stuck forever in the GME waiting room. □

Bounded Exit

Theorem 4.8: The algorithm in Figure 4.4 satisfies the bounded exit property.

Proof: Since the underlying FCFS abortable ME algorithm satisfies the bounded exit

property, it is evident by inspection of the algorithm that there are no unbounded loops in the exit protocol. This implies that the bounded exit property holds. □

4.2.4 RMR Complexity

Theorem 4.9: The algorithm in Figure 4.4 has RMR complexity $O(N)$ under the DSM model.

Proof: The loops at lines 6 and 19 make exactly $N-1$ remote memory references. Also, there exists a FCFS abortable mutual exclusion algorithm that has RMR complexity $O(\min(k,\log N))$ [9], where k is the point contention. Outside of the said loops and the FCFS abortable mutual exclusion algorithm, there are no remote memory references. Hence the resulting algorithm makes $O(N)$ remote memory references. □

4.3 A FCFS and FIFE Local-Spin GME Algorithm

The algorithm that we gave in the preceding section is local-spin under the DSM model but violates FIFE. The scenario that violates FIFE involves a process p that doorway-precedes a process q, where both p and q request the same session, while a third process r, requesting a session that conflicts with p and q, is already in the CS. Upon leaving the CS, r "releases" q before it releases p, thus allowing q to enter the CS, while p cannot enter the CS in a bounded number of its own steps.

The FIFE-violating scenario just discussed would be avoided if q could somehow "capture" p before q entered the CS. This is the approach we use in the GME algorithm in Figure 4.7. Informally we say that a process q "captures" p if it modifies the shared variables in such a way that p can enter the CS in a bounded number of its own steps. Basically, the idea behind the capturing mechanism is for q to capture those processes that finish executing the doorway sufficiently ahead of it. So, if p doorway-precedes q, then q captures p before q enters the CS, which allows p to finish the method AWAIT-CAPTURE()

and enter the CS in a bounded number of its own steps. In more detail, the capturing mechanism operates as follows: a process q, when it enters the doorway (lines 4..6), reads the *statusCS* variable of every process into the private variable *l_status*. Then, just before entering the CS (lines 24..28), q checks each process p to see if p is requesting the same session as q, p was in the GME waiting room at the time when q started the doorway, and p is still in the GME waiting room. If these conditions hold, then q attempts to capture p by setting the variable $capture[p]$ to the value of $p.passage$ that q previously read.

The opening of a new pathway to the CS via the capturing mechanism is an obvious source of concern regarding ME: a captured process may be enabled to enter the CS at a time when a process requesting a different session is in the CS. Our algorithm naturally prohibits this, but as we will see later, some technical machinery must be developed to prove that ME is guaranteed.

Under the DSM model, the algorithm in Figure 4.7 has RMR complexity $O(N)$.

A drawback to this algorithm is that it uses an integer (*passage*) that increases without bound. However, the increase is linear in the number of CS invocations, so an integer of modest size (say 64 bits) would suffice in practice.

We proceed as follows: we give some notation and definitions relative to the algorithm, provide a line by line commentary of the algorithm, present a proof of correctness, and then supply a proof for why the algorithm has RMR complexity $O(N)$.

4.3.1 Definitions and Notation

When dealing with values of type *SessionNode* in the algorithm, we sometimes use the notation (a, b), where a is the value of the *session* field and b is the value of the *passage* field.

To refer to the fields of *SessionNode* variables separately, we use dot notation. That is, if v is a variable of type *SessionNode*, then $v.session$ refers to the value of the *session*

field, and $v.passage$ refers to the value of the *passage* field.

GME Doorway: The doorway consists of lines 3..16.

GME Waiting Room: The waiting room consists of lines 17..28

GME Exit Protocol: The exit protocol consists of lines 30..38

β-doorway-precede: Process p β-doorway-precedes another process q iff p executes line 16 before q executes line 7.

Mutex Qualified Process: Process p is **mutex qualified** iff $p@\{24..34\}$ and p reached line 24 because it completed execution of the method MUTEXWAITINGROOM().

Conflict-free Qualified Process: Process p is **conflict-free qualified** iff $p@\{24..30\}$ and p reached line 24 because it completed execution of the method CONFLICT-FREE().

Capture Qualified Process: Process p is **capture qualified** iff $p@\{24..30\}$ and p reached line 24 because it completed execution of the method AWAIT-CAPTURE().

Critical-section Qualified Process: Process p is **critical-section qualified** iff $p@\{24..29\}$. Intuitively, a process is critical-section qualified iff it can enter the CS in a bounded number of its own steps in any execution.

Figure 4.6 Header for Algorithm in Figure 4.7

```
type
    SessionNode = record session: integer; passage: integer end
shared variables:
    statusCS: array[1..N] of SessionNode init (0, −1)
    capture: array[1..N] of integer init 0
    barricade: array[1..N][1..N] of boolean
/* capture[p], barricade[p][j] are stored in process p's memory, for each j. */
private variables:
    mysession: integer /* Session that p wants to attend */
    conflict_set: set of integer
    passage: integer init 0
    l_status: array[1..N] of SessionNode /* for storing statusCS[j] locally */
```

Figure 4.7 A FCFS and FIFE Local-Spin GME Algorithm; Process $p \in \{1..N\}$

```
 1: loop
 2:   NCS
 3:   passage := passage + 1;
 4:   for j ∈ {1..N} \ {p} do
 5:     l_status[j] := statusCS[j];
 6:   end for
 7:   statusCS[p] := (mysession, −1);
 8:   MutexDoorway();
 9:   conflict_set := ∅;
10:   for j = 1 to N do
11:     barricade[p][j] := true;
12:     if statusCS[j].session ∉ {0, mysession} then
13:       conflict_set := conflict_set ∪ {j};
14:     end if
15:   end for
16:   statusCS[p] := (mysession, passage);
17:   cobegin
18:       MutexWaitingRoom();
19:       ||
20:       conflict-free(); /* defined in Figure 4.5 */
21:       ||
22:       await-capture(); /* defined in Figure 4.8 */
23:   coend /* when one co-routine terminates, go to the next line */
24:   for j ∈ {1..N} \ {p} do
25:     if ∃v ≠ −1 : statusCS[j] = l_status[j] = (mysession, v) then
26:       capture[j] := l_status[j].passage;
27:     end if
28:   end for
29:   CS
30:   statusCS[p] := (0, −1);
31:   for j = 1 to N do
32:     barricade[j][p] := false;
33:   end for
34:   if mutex qualified then
35:     MutexExit();
36:   else
37:     MutexAbort();
38:   end if
39: end loop
```

Figure 4.8 Method await-capture() for Algorithm in Figure 4.7

```
1: await capture[p] = passage;
```

Capture: Suppose a process p, requesting session s_p, executes line 26 at time t_2 for iteration $j = q$. Let t_1 be the time p last executed line 5 for iteration $j = q$. We say that process p **captures** process q at time t_2 iff $q@\{17..30\}$ is invariant in $[t_1, t_2]$.

As with the definitions of "doorway-precedes" and "doorway-concurrent" (see Section 2.4, page 21), in the preceding definitions to be strictly precise we ought to refer to

the invocations that the processes are executing, rather than the processes themselves. For example, we should be saying that a particular invocation of p is mutex qualified (or conflict-free qualified). However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run.

4.3.2 Line by Line Commentary of the Algorithm

Line 3: Process p increments the value of its private variable *passage* upon leaving the NCS.

Line 4..6: Process p reads the *statusCS* variable of every other process into a private variable *l_status*. Process p will use *l_status* later to check whether there were any processes in the GME waiting room when p started the doorway.

Line 7: Process p declares that it is requesting session *mysession*, and sets $statusCS[p].passage = -1$. The latter indicates to other processes that p is not in the GME waiting room.

Line 8: Process p executes the method MUTEXDOORWAY() of the underlying mutual exclusion algorithm.

Line 10..15: Process p scans all processes to check if there are any processes requesting a different session. If there are, p adds the identifiers of these processes to its *conflict_set*. Then, when the method CONFLICT-FREE() executes, for each $j \in conflict\_set$, p will wait until $barricade[p][j] = false$.

Line 16: Process p sets $statusCS[p].passage = passage$. This serves two purposes: it indicates to other processes that p has finished the doorway and entered GME waiting room; and, it allows any process that may want to capture p after this point to read the value it should assign to $capture[p]$.

Line 17..23: Process p waits until it completes one of the co-routines MUTEXWAITINGROOM(), CONFLICT-FREE() or AWAIT-CAPTURE(), becoming, respec-

tively, mutex qualified, conflict-free qualified, or capture qualified, and proceeding to line 24.

Line 24..28: Process p attempts to capture those processes in the GME waiting room that β-doorway-precede p. This helps ensure the FIFE property.

Line 29: Process p executes the CS.

Line 30: Process p declares that it has left the CS.

Line 31..33: There may be other processes waiting for p to leave the CS. By assigning false to $barricade[j][p]$ for each process j, p ensures that any process j waiting for p does not have to wait for p any longer.

Line 34..38: If p is mutex qualified it executes the method MUTEXEXIT() before returning to the NCS. Otherwise, p previously aborted execution of the method MUTEXWAITINGROOM() at line 18, and so it executes the method MUTEXABORT() before returning to the NCS.

4.3.3 Proof of Correctness

The proof that the algorithm satisfies all the GME properties makes use of several crucial facts, which we prove in this section.

Our first goal is to show that if a process q is capture qualified, then there is some process p requesting the same session as q that captures q in q's current invocation before q is capture qualified. This is formally stated and proven in Lemma 4.12.

Lemma 4.10: If a process p captures a process q at some time t, then $q.passage = p.l\_status[q].passage$ at time t.

Proof: Assume p captures q at time t. Let t' be the time when p last executed line 5 for iteration $j = q$. By definition of capture, p executes line 26 for iteration $j = q$ at time t, and $q@\{17..30\}$ is invariant in $[t', t]$. Now, let x be the value of $q.passage$ that q assigns to $statusCS[q].passage$ when it executes line 16. By the fact that $q@\{17..30\}$ is invariant in $[t', t]$ and the fact that neither $statusCS[q]$ or $q.passage$ are updated while

$q@\{17..30\}$, $statusCS[q].passage = x$ and $q.passage = x$ are invariant in $[t', t]$. Hence, because p sets $p.l\_status[q].passage = statusCS[q].passage$ at time t', it follows that p sets $p.l\_status[q].passage = x$ at time t'. Furthermore, since no process alters $p.l\_status[q]$ in $(t', t]$, it follows that $p.l\_status[q].passage = x$ at time t. From this and the fact that $q.passage = x$ at time t, it follows that $q.passage = p.l\_status[q].passage$ at time t. □

Lemma 4.11: If a process p, requesting a session s_p, captures a process q, requesting session s_q, then $s_p = s_q$.

Proof: Assume that process p, requesting session s_p, captures process q, requesting session s_q. Let t_1 be the time when p sets $p.l\_status[q] = statusCS[q]$ at line 5, and t_2 be the time when p sets $capture[q] = p.l\_status[q].passage$ at line 26 (i.e., t_2 is the time when p captures q). Then, by definition of capture, $q@\{17..30\}$ is invariant in $[t_1, t_2]$. From this, and the fact that p must evaluate the condition at line 25 to be true, it follows that $statusCS[q].session = s_p$ is invariant in $[t_1, t_2]$. But $statusCS[q].session$ is the session that q is requesting, and so $s_q = s_p$. □

Lemma 4.12: If a process q requests a session s and is capture qualified during some invocation I_q, then there exists a process p, also requesting session s, that captures q while q is executing invocation I_q before q is capture qualified.

Proof: Assume that process q requests session s and is capture qualified during some invocation I_q. Then q reaches line 24 in I_q by finishing the method AWAIT-CAPTURE(). For q to finish the method AWAIT-CAPTURE(), it must be the case that q sees $capture[q] = q.passage$. Let v be the value of $q.passage$ (and $capture[q]$) when this happens.

The variable $q.passage = 0$ initially, q increments $q.passage$ at the start of every invocation of the algorithm at line 3, and no process ever decrements $q.passage$. From this it follows that $v > 0$.

Now, since $capture[q] = 0$ initially, and $v > 0$, it follows that there must exist some process p that sets $capture[q] = v$ prior to q being capture qualified. By examining the algorithm, we see that the only place where p can set $capture[q] = v$ is at line 26 (for

$j = q$). Let t_2 be the time when p executes line 26 (for $j = q$), and let t_1 be the time when p sets $p.l\_status[q] = statusCS[q]$ at line 5 prior to t_2.

Claim 4.12.1: $statusCS[q].passage = v$ at time t_1.

Proof: Suppose, by way of contradiction, that $statusCS[q].passage \neq v$ at time t_1. Then the value p assigns to $p.l\_status[q].passage$ at time t_1 is not equal to v, and so the value p assigns to $capture[q]$ at time t_2 is also not equal to v. This contradicts the fact that p sets $capture[q] = v$ at time t_2. □ (Claim 4.12.1)

Claim 4.12.2: $q@\{17..30\}$ at time t_1.

Proof: Suppose, by way of contradiction, that the claim is false. Then $statusCS[q].passage = -1$ at time t_1. This, together with Claim 4.12.1, implies $v = -1$. But this contradicts the fact that $v > 0$. □ (Claim 4.12.2)

Claim 4.12.3: $q.passage = v$ at time t_1.

Proof: By Claim 4.12.1, $statusCS[q].passage = v$ at time t_1, and by Claim 4.12.2, $q@\{17..30\}$ at time t_1. Since q sets $statusCS[q].passage = q.passage$ when it executes line 16, and no process changes either $q.passage$ or $statusCS[q].passage$ until after q executes line 30, it follows that $q.passage = v$ at time t_1. □ (Claim 4.12.3)

Claim 4.12.4: Process q is executing invocation I_q at time t_1.

Proof: Suppose, by way of contradiction, that the claim is false. Then, at time t_1, q is either executing an invocation that precedes I_q, or one that follows I_q. Since q is capture qualified in I_q after p sets $capture[q] = v$ at time t_2, it follows that at time t_1, q cannot be executing an invocation that follows I_q. Hence, at time t_1, q is executing an invocation I'_q that precedes I_q. Now, by Claim 4.12.3, $q.passage = v$ at time t_1. From this and the fact that $q.passage$ is incremented at line 3 at the start of every invocation, it follows that when q starts invocation I_q, $q.passage$ will be set to some value $v' > v$. So, when q reads $capture[q] = q.passage$ during invocation I_q, $q.passage = v'$. This contradicts the assumption that $q.passage = v$ at this time. □ (Claim 4.12.4)

Claim 4.12.5: $q@\{17..30\}$ is invariant in $[t_1, t_2]$.

Proof: By Claim 4.12.2, $q@\{17..30\}$ is true at time t_1, and by Claim 4.12.4, q is executing invocation I_q at time t_1. From these two facts, and the fact that q is capture qualified in invocation I_q after time t_2, it follows that $q@\{17..23\}$ is invariant in $[t_1, t_2]$. Hence $q@\{17..30\}$ is invariant in $[t_1, t_2]$. □ (Claim 4.12.5)

By Claim 4.12.5, p captures q at time t_2.

Now, Claims 4.12.4 and 4.12.5 immediately imply that q is executing invocation I_q at time t_2; the fact that p sets $capture[q] = v$ before q is capture qualified implies q is not yet capture qualified at time t_2; and Lemma 4.11 implies that p is requesting session s. These facts and the preceding discussion imply that p requests session s and captures q while q is executing invocation I_q before q is capture qualified. □

Lemma 4.13: If a process p captures a process q then (1) q β-doorway-precedes p, and (2) p is not mutex qualified.

Proof: Assume that p captures q.

(1): Let t_1 be the time when p executes line 5 for iteration $j = q$, and t_2 be the time when p executes line 26 for iteration $j = q$ (i.e., t_2 is the time when p captures q). $q@\{17..30\}$ is invariant in $[t_1, t_2]$, by definition of capture. This implies q executes line 16 before t_1. Clearly p executes line 7 after time t_1, and so q executes line 16 before p executes line 7. Hence q β-doorway-precedes p.

(2): Suppose, by way of contradiction, that p is mutex qualified. By (1), q β-doorway-precedes p. By $\mathbb{A}$'s FCFS property, p becomes mutex qualified after q has executed MUTEXEXIT() or MUTEXABORT(), and so p executes line 26 (for $j = q$) after $q@\{17..30\}$, contradicting that p captures q. □

In the above lemma, we assume that the invocation each process is executing is its "current" invocation. However, in what follows we will make use of the above lemma in situations where processes execute an invocation that is not necessarily the current invocation. This does not change the correctness of the lemma, as long as the invocation being executed is made explicit. For example, we can say that if a process p executing

invocation I_p captures a process q executing invocation I_q, then q (in invocation I_q) β-doorway-precedes p (in invocation I_p).

We now want to show that if a process q is capture qualified, then there exists a process r, requesting the same session as q, such that r is conflict-free qualified before q is capture qualified, and q β-doorway-precedes r. This fact will be important in the proof of mutual exclusion. To show that process r exists, we start by considering how q becomes capture qualified. By Lemma 4.12, there is a process p that captures q in q's current invocation before q is capture qualified, and by Lemma 4.13, q β-doorway-precedes p. If p is conflict-free qualified, then we are done. However, it is possible that p is also capture qualified, in which case we consider how p comes to be so. We continue in this manner, working backwards, in order to discover the sequence of captures that leads to q being capture qualified. It turns out that at some point in this backwards progression we will find the process r previously described. The following is a detailed proof of this.

We start by formalizing the concept of "the sequence of captures that leads to q being capture qualified". Assume that q executes some invocation I_q of the algorithm. A **sequence of (q, I_q)-captures** is a sequence $(q_0, I_{q_0}), (q_1, I_{q_1}), (q_2, I_{q_2}), ..., (q_k, I_{q_k})$ of process and invocation pairs such that (a) $(q_0, I_{q_0}) = (q, I_q)$, and (b) $\forall i : 0 \leq i < k$, q_{i+1}, while in invocation $I_{q_{i+1}}$, captures q_i, while in invocation I_{q_i}, before q_i is critical-section qualified. A **maximal sequence of (q, I_q)-captures** is a sequence of (q, I_q)-captures that is not a proper prefix of any sequence of (q, I_q)-captures.

Lemma 4.14: Suppose q executes some invocation I_q of the algorithm, and let $S_q = (q_0, I_{q_0}), (q_1, I_{q_1}), (q_2, I_{q_2}), ..., (q_k, I_{q_k})$ denote a sequence of (q, I_q)-captures. $\forall i, j : 0 \leq i < j \leq k$,

1. q_j is critical-section qualified (in invocation I_{q_j}) before q_i is critical-section qualified (in invocation I_{q_i}).

2. q_i (in invocation I_{q_i}) β-doorway-precedes q_j (in invocation I_{q_j}).

3. q_i (in invocation I_{q_i}) requests the same session as q_j (in invocation I_{q_j}).

Proof: We prove the result by induction on the value $l = j - i$. For convenience in this proof, rather than explicitly specify which invocation a process executes, we assume that a process executes the invocation with which that process is paired in the sequence S_q.

In the base case, $l = 1$ (i.e., $j = i + 1$), and so (q_i, I_{q_i}) and (q_j, I_{q_j}) are adjacent in S_q. According to the definition of a sequence of (q, I_q)-captures, q_j captures q_i before q_i is critical-section qualified. Hence:

1. In order for q_j to capture q_i, q_j must be critical-section qualified, and so q_j is critical-section qualified before q_i is critical-section qualified.

2. By Lemma 4.13, q_i β-doorway-precedes q_j.

3. By Lemma 4.11, q_i requests the same session as q_j.

For the induction step, assume that the lemma holds for all values smaller than l. We now prove the result for $l > 1$.

Consider the element $(q_{i+1}, I_{q_{i+1}})$ of S_q. The following must hold by the induction hypothesis:

1. q_j is critical-section qualified before q_{i+1} is critical-section qualified.

2. q_{i+1} β-doorway-precedes q_j.

3. q_{i+1} requests the same session as q_j.

Hence, the following results hold:

1. Since q_j is critical-section qualified before q_{i+1} is critical-section qualified, and q_{i+1} is critical-section qualified before q_i is critical-section qualified, it follows that q_j is critical-section qualified before q_i is critical-section qualified.

2. Since q_i β-doorway-precedes q_{i+1} and q_{i+1} β-doorway-precedes q_j, it follows that q_i β-doorway-precedes q_j.

3. Since q_i requests the same session as q_{i+1}, and q_{i+1} requests the same session as q_j, it follows that q_i requests the same session as q_j. □

Lemma 4.15: Suppose q executes some invocation I_q of the algorithm. A maximal sequence of (q, I_q)-captures has at most N elements.

Proof: Suppose, by way of contradiction, that the maximal sequence has more than N elements. Then some process r appears at least twice in the sequence. Let I_1 be the invocation r is paired with the first time it appears in the sequence, and I_2 be the invocation r is paired with the second time in appears in the sequence. By Lemma 4.14, r (in invocation I_2) is critical-section qualified before r (in invocation I_1) is critical section qualified, and r (in invocation I_1) β-doorway-precedes r (in invocation I_2). The first fact implies that r executes invocation I_2 before invocation I_1, but the second implies that r executes invocation I_1 before I_2. This is a contradiction. □

Lemma 4.16: Suppose q is capture qualified in some invocation I_q of the algorithm. Then a maximal sequence of (q, I_q)-captures has at least 2 elements.

Proof: Follows immediately from Lemma 4.12 and the definition of a maximal sequence of (q, I_q)-captures. □

Lemma 4.17: Suppose q executes some invocation I_q of the algorithm, and let $S_q = (q_0, I_{q_0}), (q_1, I_{q_1}), (q_2, I_{q_2}), ..., (q_k, I_{q_k})$ denote a maximal sequence of (q, I_q)-captures that has at least 2 elements. Then q_k is conflict-free qualified in invocation I_{q_k}.

Proof: Suppose, by way of contradiction, that q_k is not conflict-free qualified in invocation I_{q_k}. By the definition of a sequence of (q, I_q)-captures, q_k, during its execution of invocation I_{q_k}, captures q_{k-1} while q_{k-1} is executing invocation $I_{q_{k-1}}$ before q_{k-1} is critical-section qualified. From this and Lemma 4.13 it follows that q_k is not mutex qualified. Hence q_k is either conflict-free qualified or capture qualified. By our assumption, q_k

cannot be conflict-free qualified, and so it must be capture qualified. But this and Lemma 4.12 imply that S_q is not a maximal sequence of (q, I_q)-captures, which contradicts the assumption that it is such a sequence. □

Lemma 4.18: If q requests a session s and is capture qualified, then there exists some process r, also requesting session s, such that r is conflict-free qualified before q is capture qualified, and q β-doorway-precedes r.

Proof: Assume that q is capture qualified and requests a session s in some invocation I_q. Now, consider the maximal sequence of (q, I_q)-captures, $S_q = (q_0, I_{q_0}), (q_1, I_{q_1}), (q_2, I_{q_2}), ..., (q_k, I_{q_k})$. By Lemma 4.16, S_q has at least 2 elements, and thus, by Lemma 4.17, the process q_k is conflict-free qualified in invocation I_{q_k}. Moreover, by Lemma 4.14, the following three facts hold:

1. q_k is conflict-free qualified (in invocation I_{q_k}) before q is capture qualified (in invocation I_q).

2. q (in invocation I_q) β-doorway-precedes q_k (in invocation I_{q_k}).

3. q (in invocation I_q) requests the same session as q_k (in invocation I_{q_k}).

Thus q_k is the process r whose existence we sought to prove. □

Lemma 4.19: Suppose a process p assigns *true* to $barricade[p][q]$ at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q] \neq (0, -1)$ is invariant in the interval $[t_1, t_2]$, then $barricade[p][q] = true$ is invariant in $[t_1, t_2]$.

Proof: Assume that $statusCS[q] \neq (0, -1)$ is invariant in the interval $[t_1, t_2]$. Then clearly $q@\{8..30\}$ is invariant in the interval $[t_1, t_2]$, and so q executes no part of its exit protocol in $[t_1, t_2]$. Since the only place where *false* can be assigned to $barricade[p][q]$ is in q's exit protocol, at line 32, it must be the case that *false* is never assigned to $barricade[p][q]$ in $[t_1, t_2]$. Hence, because *true* is assigned to $barricade[p][q]$ at time t_1, it follows that $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. □

Corollary 4.20: Suppose a process p, requesting session s_p, assigns *true* to $barricade[p][q]$ at line 11 at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, then p cannot be conflict-free qualified in the interval $[t_1, t_2]$.

Proof: Assume that $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$. Suppose, by way of contradiction, that p is conflict-free qualified at some point in the interval $[t_1, t_2]$. Thus p finishes executing the method CONFLICT-FREE() by time t_2. Now, by our assumption that $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 12 to be true and thus adds q to $p.conflict\_set$ at line 13, for iteration $j = q$. Thus p busy-waits in the method CONFLICT-FREE() until $barricade[p][q] = false$. By Lemma 4.19, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Hence the method CONFLICT-FREE() cannot finish by time t_2, which contradicts the fact it does finish by time t_2. □

Mutual Exclusion

To prove that the algorithm satisfies the mutual exclusion property, we prove a slightly stronger fact which will also be useful later in the proof that the algorithm satisfies the FIFE property.

Lemma 4.21: If a process p requests some session s_p, and a process q requests some session $s_q \neq s_p$, then p and q cannot be simultaneously enabled to execute lines 24..30.

Proof: Assume that a process p requests some session s_p and a process q requests some session $s_q \neq s_p$. Suppose, by way of contradiction, that p and q are simultaneously enabled to execute lines 24..30 at some time t.

For a process to be enabled to execute lines 24..30, it must be either mutex qualified, capture qualified, or conflict-free qualified. We now consider the different ways in which p and q can be enabled to execute lines 24..30, and show that a contradiction arises in each case.

Case 1: Both processes are mutex qualified.

Due to the mutual exclusion property of $\mathbb{A}$, at most one process can be mutex qualified at a time. Hence this case is impossible.

Case 2: One process is capture qualified.

Without loss of generality, assume that q is capture qualified. By Lemma 4.18, there exists a process r, requesting session s_q, such that r is conflict-free qualified before q is capture qualified, and q β-doorway-precedes r.

Claim 4.21.1: Process r executes line 9 before p executes line 7.

Proof: Suppose, by way of contradiction, that the claim is false. Let t_a be the time when r sets $barricade[r][p] = true$ at line 11. By the assumption that p executes line 7 before r executes line 9, $statusCS[p].session \notin \{0, s_q\}$ in $[t_a, t]$. Thus, by Corollary 4.20, r cannot be conflict-free qualified in the interval $[t_a, t]$. From this and the fact that q is capture qualified after r is conflict-free qualified, it follows that q cannot be capture qualified in the interval $[t_a, t]$. This contradicts the fact that q is capture qualified at time t. □ (Claim 4.21.1)

Claim 4.21.2: q β-doorway-precedes p.

Proof: Process q β-doorway-precedes r, and so q executes line 16 before r executes line 7. This implies q executes line 16 before r executes line 9. Moreover, by Claim 4.21.1, r executes line 9 before p executes line 7. Hence, q executes line 16 before p executes line 7, and so q β-doorway-precedes p. □ (Claim 4.21.2)

Now consider the different ways in which p can be enabled to execute lines 24..30:

Case 2.1: Process p is mutex qualified.

By Claim 4.21.2, q finishes executing the method MUTEXDOORWAY() at line 8 before p starts executing the method MUTEXDOORWAY() at line 8. Thus, by the FCFS property of $\mathbb{A}$, p is mutex qualified after q executes the method MUTEXABORT() at line 37. This implies p is mutex qualified after q is no longer enabled to execute lines 24..30. This contradicts the assumption that p and q are simultaneously enabled to execute lines

24..30 at time t.

Case 2.2: Process p is conflict-free qualified.

Let t_a be the time when p sets $barricade[p][q] = true$ at line 11. Claim 4.21.2 implies that q executes line 16 before p executes line 7, and so q executes line 16 before time t_a. Hence $statusCS[q].session \notin \{0, s_p\}$ is invariant in $[t_a, t]$. By this fact and Corollary 4.20, p cannot be conflict-free qualified in the interval $[t_a, t]$, which contradicts the assumption that p is conflict-free qualified at time t.

Case 2.3: Process p is capture qualified.

By Claim 4.21.2, q β-doorway-precedes p. Since p and q are both capture qualified, symmetrically, p β-doorway-precedes q. This is a contradiction.

Case 3: One process is conflict-free qualified, and the other is not capture qualified.

Without loss of generality, assume that q is conflict-free qualified.

Claim 4.21.3: Process q starts executing the loop at line 10 before p executes line 7.

Proof: Suppose, by way of contradiction, that the claim is false. Let t_1 be the time when q writes *true* to $barricade[q][p]$ at line 11. By assumption, p executes line 7 before time t_1, and so $statusCS[p].session \notin \{0, s_q\}$ is invariant during the interval $[t_1, t]$. Thus, by Corollary 4.20, q cannot be conflict-free qualified in the interval $[t_1, t]$, contradicting the fact that q is conflict-free qualified at time t. □ (Claim 4.21.3)

Now consider the different ways in which p can be enabled to execute lines 24..30:

Case 3.1: Process p is mutex qualified.

By Claim 4.21.3, q will finish executing the method MUTEXDOORWAY() at line 8 before p starts executing the method MUTEXDOORWAY() at line 8. Hence, due to the FCFS property of $\mathbb{A}$, p cannot become mutex qualified until q executes the method MUTEXABORT() at line 37. This contradicts the assumption that p and q are simultaneously enabled to execute lines 24..30 at time t.

Case 3.2: Process p is conflict-free qualified.

By Claim 4.21.3, q starts executing the loop at line 10 before p executes line 7, and so q

executes line 7 before p does. Since p and q are both conflict-free qualified, symmetrically, p executes line 7 before q does. This is a contradiction.

Since each case leads to a contradiction, it follows that the lemma holds. □

Theorem 4.22: The algorithm in Figure 4.7 satisfies the mutual exclusion property.

Proof: Follows immediately from Lemma 4.21. □

Strong Concurrent Entering

Theorem 4.23: The algorithm in Figure 4.7 satisfies the strong concurrent entering property.

Proof: Let p be a process requesting session s_p that is in its GME waiting room at time t_1. Assume that at t_1 every active process that doorway-precedes or is doorway-concurrent with p requests session s_p. Now, let t_2 be the time when p starts its next invocation, if it exists. If there is no next invocation, then $t_2 = \infty$. We want to show that p will enter the CS in its current invocation in a bounded number of its own steps. For this it suffices to show that $\forall j \in p.conflict\_set$, $barricade[p][j] = false$ during the interval $[t_1, t_2)$. Suppose that $q \in p.conflict\_set$. Then at line 12 at some time $t_q < t_1$, p read $statusCS[q].session \notin \{0, s_p\}$. This implies that at time t_q, q@{8..30}. This, in turn, implies that q doorway-precedes or is doorway-concurrent with p. By assumption, q is not active at time t_1, and so at some time in (t_q, t_1), q will execute the loop at line 31, assigning *false* to $barricade[p][q]$. Now, the only process that can set $barricade[p][q]$ to *true* is process p. Process p's last assignment of *true* to $barricade[p][q]$ occurred before t_q, and its next assignment occurs after t_2. It follows that $barricade[p][q] = false$ in $[t_1, t_2)$, as required. □

Lockout Freedom

Theorem 4.24: The algorithm in Figure 4.7 satisfies the lockout freedom property.

Proof: Suppose, by way of contradiction, that the algorithm does not satisfy the

lockout freedom property. This means that there is an execution in which some process gets stuck forever in the GME waiting room when it tries to enter the CS. The following claim proves that this is impossible.

Claim 4.24.1: No process can be stuck forever in the method MUTEXWAITINGROOM().

Proof: $\mathbb{A}$ is an abortable mutual exclusion algorithm that satisfies the lockout freedom property. Hence the only way for a process to get stuck forever in MUTEXWAITINGROOM() is if some other process becomes mutex qualified and then does not execute the method MUTEXEXIT(); or, if some process starts executing MUTEXWAITINGROOM(), aborts execution of it, but then does not execute the method MUTEXABORT(). If a process becomes mutex qualified, then we see by examining the algorithm that it will eventually execute the method MUTEXEXIT(). Furthermore, the only way a process will abort execution of MUTEXWAITINGROOM() once started is if the process becomes conflict-free qualified or capture qualified. In this case, again by examining the algorithm, we see that such a process will eventually execute the method MUTEXABORT(). Hence no process can be stuck forever in the method MUTEXWAITINGROOM(). □ (Claim 4.24.1)

Claim 4.24.1 contradicts the fact that there is an execution in which some process is stuck forever in the GME waiting room. □

FCFS

Theorem 4.25: The algorithm in Figure 4.7 satisfies the FCFS property.

Proof: Assume that p requests session s_p, q requests session s_q, $s_p \neq s_q$, and p doorway-precedes q. We must prove that q does not enter the CS before p enters the CS. Suppose, by way of contradiction, that it does. There are three cases, depending on how q enters the CS.

Case 1: Process q enters the CS mutex qualified before p.

Since p doorway-precedes q, p finishes executing the method MUTEXDOORWAY() at line 8 before q begins executing the method MUTEXDOORWAY() at line 8. Hence, by the FCFS property of $\mathbb{A}$, q cannot be mutex qualified until p executes one of MUTEXEXIT() or MUTEXABORT(). This contradicts the assumption that q enters the CS mutex qualified before p.

Case 2: Process q enters the CS conflict-free qualified before p.

Let t_q be the time when q assigns *true* to $barricade[q][p]$ at line 11, and let t_p be the time when p enters the CS. Since p doorway-precedes q, it follows that $statusCS[p].session \notin \{0, s_q\}$ is invariant in the interval $[t_q, t_p]$. So, by Corollary 4.20, q cannot be conflict-free qualified in the interval $[t_q, t_p]$. This implies that q cannot enter the CS conflict-free qualified before p, which contradicts the assumption that it does.

Case 3: Process q enters the CS capture qualified before p.

By Lemma 4.18, there exists a process r, requesting session s_q, such that r is conflict-free qualified before q is capture qualified, and q β-doorway-precedes r. By definition of β-doorway-precedes, q executes line 16 before r executes line 7. Moreover, p doorway-precedes q, so p executes line 16 before q executes line 3. Hence p executes line 16 before r executes line 7.

Let t_1 be the time when r sets $barricade[r][p] = true$ at line 11, and let t_2 be the time when p enters the CS. Since p executes line 16 before r executes line 7, it must be the case that p executes line 16 before time t_1. From this it follows that $statusCS[p].session \notin \{0, s_q\}$ is invariant in $[t_1, t_2]$. Thus, by Corollary 4.20, r cannot be conflict-free qualified in $[t_1, t_2]$. From this and the fact that q is capture qualified after r is conflict-free qualified, it follows that q is capture qualified after t_2. Since t_2 is the time when p enters the CS, it must be the case that q is capture qualified after p enters the CS. This contradicts the assumption that q enters the CS capture qualified before p. □

FIFE

Theorem 4.26: The algorithm in Figure 4.7 satisfies the FIFE property.

Proof: Assume that p and q request session s, p doorway-precedes q, and q enters the CS while p is in the GME waiting room. We refer to the invocation of p in which the preceding is true as p's current invocation.

Our goal is to show that p enters the CS in a bounded number of its own steps. We first observe that if p does not detect conflict in the doorway (i.e., $p.conflict\_set = \emptyset$), then p will complete the method CONFLICT-FREE() and enter the CS in a bounded number of its own steps, and we are done. So, in the following, we assume that p does detect conflict (i.e., $p.conflict\_set \neq \emptyset$).

We will next show that q captures p before q enters the CS (Claim 4.26.1). This implies that q sets $capture[p] = p.passage$ before q enters the CS. We further show that no process can change the value of $capture[p]$ between the time when q writes to $capture[p]$ and the time when p enters the CS (Claim 4.26.2). These two claims imply that p is able to finish the method AWAIT-CAPTURE() and enter the CS in a bounded number of its own steps.

Claim 4.26.1: Process q captures p prior to q entering the CS.

Proof: Let t_1 be the time when q sets $l\_status[p] = statusCS[p]$ at line 5, and let t_3 be the time when q enters the CS.

Sub-Claim 4.26.1.1: $p@\{17..28\}$ is invariant in $[t_1, t_3]$.

Proof: Since p doorway-precedes q, it follows that p enters the GME waiting room before q executes line 3. This implies that p enters the GME waiting room before time t_1. Moreover, by assumption, p is still in the GME waiting room when q enters the CS at time t_3. Hence p is in the GME waiting room during the interval $[t_1, t_3]$. Thus, it must be the case that $p@\{17..28\}$ is invariant in $[t_1, t_3]$. □ (Sub-Claim 4.26.1.1)

Since $p@\{17..28\}$ at time t_1, it must be the case that for some $v \neq -1$, q sets $q.l\_status[p] = (s, v)$ at time t_1. This and Sub-Claim 4.26.1.1 imply that q evaluates

the condition at line 25 (for $j = p$) to be true. Hence q executes line 26 (for $j = p$) before entering the CS. Let t_2 be the time when q executes line 26 (for $j = p$). Now, to show that q captures p prior to q entering the CS we must show that $p@\{17..30\}$ is invariant in $[t_1, t_2]$. This follows from Sub-Claim 4.26.1.1 and the fact that $t_2 \in [t_1, t_3]$. □ (Claim 4.26.1)

Let t_q be the time when q captures p, and let t_p be the time when p enters the CS. The next claim guarantees p is able to enter the CS in a bounded number of its own steps.

Claim 4.26.2: $capture[p] = p.passage$ is invariant in $[t_q, t_p]$.

Proof: Suppose, by way of contradiction, that the claim is false. By Lemma 4.10 and the fact that q captures p at time t_q, $p.passage = q.l\_status[p].passage$ at time t_q. This implies that q sets $capture[p] = p.passage$ at time t_q. Let v be the value of $p.passage$ at this time. Clearly $p.passage = v$ is invariant in $[t_q, t_p]$. So, by the assumption that $capture[q] = p.passage$ is not invariant in $[t_q, t_p]$, and the fact that operations on $capture[p]$ are atomic, it must be the case that some process r sets $capture[p] = v'$, where $v' \neq v$, at some time $t''_r \in (t_q, t_p]$. The only place in the algorithm where r can do this is at line 26. Clearly the value of $r.l\_status[p].passage$ at this time must be v'. This implies that r previously set $r.l\_status[p].passage = v'$ when r executed line 5 for iteration $j = p$. Let t_r be the time when this happens.

Sub-Claim 4.26.2.1: At time t_r, p is executing an invocation that precedes its current invocation.

Proof: Suppose, by way of contradiction, that the claim is false. Then, at time t_r, p is either executing its current invocation, or an invocation that follows its current invocation. In the former case, where p is executing its current invocation, it must be the case that $v' = v$, which contradicts that $v' \neq v$. In the latter case, where p is executing an invocation that follows its current invocation, it must be the case that $t_r > t_p$, which contradicts the fact that $t_r < t''_r < t_p$. □ (Sub-Claim 4.26.2.1)

Hence, by Sub-Claim 4.26.2.1, at time t_r, p is executing some invocation I_p that occurs prior to its current invocation. Let t'_r be the time when r evaluates the condition at line 25 prior to time t''_r. Process r must evaluate the condition at line 25 to be true, and so p must still be executing invocation I_p at time t'_r.

To obtain a contradiction, we now show that q cannot be mutex qualified, conflict-free qualified, or capture qualified at time t_q.

Sub-Claim 4.26.2.2: $t_q \in (t'_r, t''_r)$.

Proof: Process p is executing its earlier invocation I_p at time t'_r, and clearly p is executing its current invocation at time t_q. So $t'_r < t_q$. This, together with the fact that $t''_r \in (t_q, t_p]$, implies that $t_q \in (t'_r, t''_r)$. □ (Sub-Claim 4.26.2.2)

Sub-Claim 4.26.2.3: Process r requests session s.

Proof: By Sub-Claim 4.26.2.2, $t_q \in (t'_r, t''_r)$. That is, q and r are simultaneously enabled to execute line 26. By Lemma 4.21, r requests the same session as q (i.e., session s). □ (Sub-Claim 4.26.2.3)

By assumption, when p is in the GME waiting room in its current invocation, $p.conflict\_set \neq \emptyset$. Let u be a process in $p.conflict\_set$. It must be the case that at some time t_u when p executes line 12 (for $j = u$), p sees $statusCS[u].session \notin \{0, s\}$. Clearly $u@\{8..30\}$ at time t_u, and u requests some session $s' \neq s$.

Sub-Claim 4.26.2.4: $t_u \in (t'_r, t''_r)$.

Proof: Since t_u is a time when p is executing its current invocation, and t'_r is a time when p is executing its earlier invocation I_p, it follows that $t_u > t'_r$. Furthermore, p doorway-precedes q, and so $t_u < t_q$. Since $t_q < t''_r$, it follows that $t_u < t''_r$. Hence $t_u \in (t'_r, t''_r)$. □ (Sub-Claim 4.26.2.4)

Sub-Claim 4.26.2.5: Process u is not mutex qualified, conflict-free qualified, or capture qualified at any point in $[t_u, t''_r)$.

Proof: Suppose, by way of contradiction, that the claim is false. $u@\{8..30\}$ at time t_u. So, by the assumption that u is either mutex qualified, conflict-free qualified, or capture

qualified at some point in $[t_u, t''_r)$, it follows that $u@\{24..30\}$ at some point in $[t_u, t''_r)$. This and Sub-Claim 4.26.2.4 imply that $u@\{24..30\}$ at some point in (t'_r, t''_r). In turn, this, the fact that u requests a session $s' \neq s$, and the fact that r requests session s (true by Sub-Claim 4.26.2.3), imply that there are two processes requesting different sessions enabled to execute in lines 24..30 at the same time. This contradicts Lemma 4.21. □ (Sub-Claim 4.26.2.5)

Sub-Claim 4.26.2.6: $u@\{8..23\}$ is invariant in $[t_u, t''_r)$.

Proof: $u@\{8..30\}$ is true at time t_u, and by Sub-Claim 4.26.2.5, u is not mutex qualified, conflict-free qualified, or capture qualified at any point in $[t_u, t''_r)$. Hence $u@\{8..23\}$ is invariant in $[t_u, t''_r)$. □ (Sub-Claim 4.26.2.6)

Sub-Claim 4.26.2.7: If p, in its current invocation, β-doorway-precedes some process z, requesting session s, then z is not conflict-free qualified at any point in (t'_r, t''_r).

Proof: Assume that process p, in its current invocation, β-doorway-precedes some process z, requesting session s. Suppose, by way of contradiction, that z is conflict-free qualified at some point in (t'_r, t''_r). Let t_1 be the time when z sets $barricade[z][u] = true$ at line 11. Time t_u is when p sees $statusCS[u].session \notin \{0, s\}$ at line 12, and p β-doorway-precedes z, so it must be the case that $t_u < t_1$. Moreover, by the assumption that z is conflict-free qualified in (t'_r, t''_r), $t_1 < t''_r$. So it must be the case that $t_1 \in (t_u, t''_r)$. Now, by Sub-Claim 4.26.2.6, $u@\{8..23\}$ is invariant in $[t_u, t''_r)$, and so $u@\{8..23\}$ is invariant in $[t_1, t''_r)$. This implies $statusCS[u].session \notin \{0, s\}$ is invariant in $[t_1, t''_r)$. Hence, by Corollary 4.20, z cannot be conflict-free qualified in $[t_1, t''_r)$. This contradicts the assumption that z is conflict-free qualified in (t'_r, t''_r). □ (Sub-Claim 4.26.2.7)

Sub-Claim 4.26.2.8: Process q is not conflict-free qualified at any point in (t'_r, t''_r).

Proof: Process p, in its current invocation, doorway-precedes q, and so p β-doorway--precedes q. Also, by assumption, q requests session s. Hence, by Sub-Claim 4.26.2.7, q is not conflict-free qualified at any point in (t'_r, t''_r). □ (Sub-Claim 4.26.2.8)

Sub-Claim 4.26.2.9: Process q is not capture qualified at any point in (t'_r, t''_r).

Proof: Suppose, by way of contradiction, that the claim is false. By Lemma 4.18, there exists a process z, requesting session s, such that z is conflict-free qualified before q is capture qualified, and q β-doorway-precedes z. We will prove the following three facts:

1. p (in its current invocation) β-doorway-precedes z.

2. z is conflict-free qualified after t'_r.

3. z is conflict-free qualified before t''_r.

By (1), the fact that z requests s, and Sub-Claim 4.26.2.7, it follows that z is not conflict-free qualified during (t'_r, t''_r). This contradicts (2) and (3). So it remains to prove (1)-(3).

(1) follows from the facts that p doorway-precedes q and that q β-doorway-precedes z.

(2) follows from (1), the fact that p executes line 16 after t_u and that t_u is after t'_r (Sub-Claim 4.26.2.4).

(3) follows from the fact that z is conflict-free qualified before q is capture qualified and the assumption that q is capture qualified during (t'_r, t''_r). □ (Sub-Claim 4.26.2.9)

Sub-Claim 4.26.2.10: Process q is not mutex qualified at any point in (t'_r, t''_r).

Proof: Suppose, by way of contradiction, that the claim is false. Then q can enter the CS mutex qualified. Process p doorway-precedes q, and so p finishes executing the method MUTEXDOORWAY() at line 8 before q starts executing the method MUTEXDOORWAY() at line 8. By the FCFS property of $\mathbb{A}$, q can be mutex qualified only after p executes either MUTEXEXIT() or MUTEXABORT() after line 34. But this implies that p enters the CS before q, which contradicts the assumption that p is in the GME waiting room when q enters the CS. □ (Sub-Claim 4.26.2.10)

By Sub-Claim 4.26.2.2, q captures p at time $t_q \in (t'_r, t''_r)$. Hence q must be mutex qualified, conflict-free qualified, or capture qualified at some point in (t'_r, t''_r), which contradicts Sub-Claims 4.26.2.8, 4.26.2.9, and 4.26.2.10 □ (Claim 4.26.2)

By Claim 4.26.2, $capture[p] = p.passage$ is invariant in $[t_q, t_p]$. Hence, p is able to finish the method AWAIT-CAPTURE() and enter the CS in a bounded number of its own steps in $[t_q, t_p]$. □

Bounded Exit

Theorem 4.27: The algorithm in Figure 4.7 satisfies the bounded exit property.

Proof: Since the underlying FCFS abortable ME algorithm satisfies the bounded exit property, it is evident by inspection of the algorithm that there are no unbounded loops in the exit protocol. This implies that the bounded exit property holds. □

4.3.4 RMR Complexity

Theorem 4.28: The algorithm in Figure 4.7 has RMR complexity $O(N)$ under the DSM model.

Proof: Each of the loops at lines 4, 10, 24, and 31 makes exactly $N - 1$ remote memory references. Also, there exists a FCFS abortable mutual exclusion algorithm that has RMR complexity $O(\min(k,\log\ N))$ [9], where k is the point contention. Outside of the said loops and the FCFS abortable mutual exclusion algorithm, there are no remote memory references. Hence the resulting algorithm makes $O(N)$ remote memory references. □

Chapter 5

High Concurrency Local-Spin GME

A drawback of the algorithms in the preceding chapter can be illustrated by the following scenario: Let S be a set of processes, and p be a process that is not in S. Suppose that the processes in $S \cup \{p\}$ all leave the NCS at the same time, and every process in S requests a session s, while p requests a session $s' \neq s$. Further suppose that p declares the session it is requesting (i.e., p sets the $session[p]$ variable at line 3 in Figure 4.4, or p sets the $statusCS[p].session$ variable at line 7 in Figure 4.7) before any process in S goes through the loop in the doorway that checks for processes requesting a conflicting session. This means that every process in S will detect conflict and cannot be conflict-free qualified until after p leaves the CS. Moreover, assume that no process in S β-doorway-precedes another process in S, so that there is no opportunity for any process in S to be capture qualified. Now, suppose that the processes in $S \cup \{p\}$ execute through the GME doorway and GME waiting room in such a way so that all processes in S become mutex qualified before p. This means that all the processes in S execute through the CS sequentially, despite the opportunity for concurrency in this scenario: all processes in S could be granted access to the CS simultaneously. However, the algorithms from the preceding chapter operate in a way that prevents this from happening.

In this chapter we present algorithms that address this drawback. They all use a

capturing mechanism to increase concurrency during periods of contention and avoid the scenario just described. (A capturing mechanism allows one process to capture another process requesting the same session to "help" it enter the CS. Informally, we say that a process q captures a process p if it modifies the shared variables in such a way that p can enter the CS in a bounded number of its own steps.) The increased concurrency is achieved at the expense of the stronger fairness properties (FCFS, FIFE, strong concurrent entering) that were satisfied by the algorithms in the preceding chapter.

The first algorithm we present makes use of COMPARE_AND_SWAP to implement the capturing mechanism; the second uses only reads and writes, but unbounded registers; and the third, and final, uses only reads and writes, but bounded registers. Moreover, all three algorithms in this chapter have RMR complexity $O(N)$. In view of Corollary 3.2, this is the best possible RMR complexity (within a constant factor).

5.1 An Algorithm That Uses Compare-And-Swap

The algorithm we present in this section, in Figure 5.2, uses COMPARE_AND_SWAP to implement the capturing mechanism. A process p requesting session s_p that invokes the capturing mechanism (lines 25..29) scans all processes to see if anyone is currently requesting entry into the CS for the same session. It detects this by checking whether a process j has its $statusCS[j]$ variable set to (REQ, s_p). If p finds such a process, it uses COMPARE_AND_SWAP to change $statusCS[j]$ to (IN, s_p). A captured process j that is stuck in its GME waiting room at lines 12..18 will be able to exit the waiting room by completing the method AWAIT-CAPTURE(), which simply waits for $statusCS[j]$ to become $(IN, j.mysession)$.

For a process to invoke the capturing mechanism, it needs to be mutex qualified. This prevents the capturing mechanism from being invoked by processes that have been previously captured. Without this condition, there exists an execution scenario as follows:

a process p_1 captures a process p_2, p_1 exits and re-enters the trying protocol, p_2 captures p_1, p_2 exits and re-enters the trying protocol, p_1 captures p_2, etc. This capturing cycle can continue ad infinitum, starving processes requesting sessions different from p_1 and p_2.

This capturing mechanism differs from the one used in the algorithm in Figure 4.7. In that algorithm, a process was restricted to capture only those processes that β-doorway-preceded it. In the algorithm presented here, there is no such restriction.

We proceed as follows: we give some definitions relative to the algorithm, provide a line by line commentary of the algorithm, present a proof of correctness, and then supply a proof for why the algorithm has RMR complexity $O(N)$.

5.1.1 Definitions

GME Doorway: The doorway consists of lines 3..11. Even though this algorithm is not FCFS, we define as the doorway an initial portion of the trying protocol that a process completes in a bounded number of its own steps.

GME Waiting Room: The waiting room consists of lines 12..29

GME Exit Protocol: The exit protocol consists of lines 31..39

Figure 5.1 Header for Algorithm in Figure 5.2

shared variables:

statusCS: **array**[1..N] **of** $\{(OUT, 0), (IN, integer), (REQ, integer)\}$ **init** $(OUT, 0)$
barricade: **array**[1..N][1..N] **of boolean init** *true*

/* *statusCS*[p], *barricade*[p][j] stored in process p's memory, for each j. */

private variables:

mysession: **integer** /* Session that p wants to attend */
conflict_set: **set of integer**

Figure 5.2 An Algorithm That Uses Compare-And-Swap; Process $p \in \{1..N\}$

```
 1: loop
 2:   NCS
 3:   statusCS[p] := (REQ, mysession);
 4:   MutexDoorway();
 5:   conflict_set := ∅;
 6:   for j = 1 to N do
 7:     barricade[p][j] := true;
 8:     if statusCS[j] ∉ {(OUT, 0), (REQ, mysession), (IN, mysession)} then
 9:       conflict_set := conflict_set ∪ {j};
10:     end if
11:   end for
12:   cobegin
13:       MutexWaitingRoom();
14:       ||
15:       conflict-free(); /* defined in Figure 5.3 */
16:       ||
17:       await-capture(); /* defined in Figure 5.4 */
18:   coend /* when one co-routine terminates, go to the next line */
19:   for j = 1 to N do
20:     barricade[p][j] := true;
21:     if ∃s ≠ mysession : statusCS[j] = (IN, s) then
22:       await ¬barricade[p][j];
23:     end if
24:   end for
25:   if mutex qualified then
26:     for j ∈ {1..N} \ {p} do
27:       compare_and_swap(statusCS[j],(REQ, mysession),(IN, mysession));
28:     end for
29:   end if
30:   CS
31:   statusCS[p] := (OUT, 0);
32:   for j = 1 to N do
33:     barricade[j][p] := false;
34:   end for
35:   if mutex qualified then
36:     MutexExit();
37:   else
38:     MutexAbort();
39:   end if
40: end loop
```

Figure 5.3 Method conflict-free() for Algorithm in Figures 5.2, 5.6, 5.9

```
1: for each j ∈ conflict_set do
2:   await ¬barricade[p][j];
3: end for
```

Figure 5.4 Method await-capture() for Algorithm in Figure 5.2

```
1: await statusCS[p] = (IN, mysession);
```

As with previous algorithms we occasionally use definitions in which, to be strictly

correct, we would need to refer to the invocations that the processes are executing. However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run. This comment applies to the following definitions.

Mutex Qualified Process: Process p is **mutex qualified** iff $p@\{19..35\}$ and p reached line 19 because it completed execution of the method MUTEXWAITINGROOM().

Conflict-free Qualified Process: Process p is **conflict-free qualified** iff $p@\{19..31\}$ and p reached line 19 because it completed execution of the method CONFLICT-FREE().

Capture Qualified Process: Process p is **capture qualified** iff $p@\{19..31\}$ and p reached line 19 because it completed execution of the method AWAIT-CAPTURE().

Critical-Section Qualified Process: Process p is **critical-section qualified** iff $p@\{25..30\}$. Intuitively, if a process p is critical-section qualified then it can enter the CS in a bounded number of its own steps.

Capture: We say that a process p **captures** another process q iff p's COMPARE_AND_SWAP at line 27 succeeds for iteration $j = q$.

Captured State: Process p, requesting session s, is in a **captured state** iff $statusCS[p] = (IN, s)$.

5.1.2 Line by Line Commentary of the Algorithm

Line 3: Process p declares that it is requesting session *mysession*.

Line 4: Process p executes the method MUTEXDOORWAY() of the underlying FCFS abortable mutual exclusion algorithm.

Line 5..11: Process p scans all processes to check if there are any other processes requesting a different session than p. If there is, p will add the identifiers of these processes to its *conflict_set*. When the method CONFLICT-FREE() executes, for each $j \in conflict\_set$, p will attempt to wait on the variable $barricade[p][j]$.

Line 12..18: Process p waits until it completes one of the co-routines contained within

these lines. After completing one of the co-routines p is either mutex qualified, conflict-free qualified, or capture qualified, and proceeds to line 19.

Line 19..24: Process p checks if there are any previously captured processes, requesting a session different from itself, still inside the CS. If there are, then p waits until they exit.

Line 25..29: If process p, requesting session s_p, is mutex qualified, then it will attempt to capture any process j requesting the same session as p. The use of COMPARE_AND_SWAP at line 27 is necessary to ensure that when p checks if j is requesting s_p, the assignment of (IN, s_p) to $statusCS[j]$ occurs atomically with the check.[1]

Line 30: Process p executes the CS.

Line 31: Process p declares that it has left the CS.

Line 32..34: There may be other processes waiting for p to leave the CS. By assigning *false* to $barricade[j][p]$ for each process j, p ensures that any process j waiting for p does not have to wait for p any longer.

Line 35..39: If p is mutex qualified, then p executes the method MUTEXEXIT() before returning to the NCS. Otherwise, if p is not mutex qualified, then p previously aborted execution of the method MUTEXWAITINGROOM() at line 13. For this reason p executes the method MUTEXABORT() before returning to the NCS.

5.1.3 Proof of Correctness

We begin with an elementary lemma and two corollaries that are used repeatedly in the proof of the algorithm's correctness.

Lemma 5.1: Suppose process p assigns *true* to $barricade[p][q]$ at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q] \neq (OUT, 0)$ is invariant in the interval $[t_1, t_2]$, then $barricade[p][q] = true$ is invariant in interval $[t_1, t_2]$.

Proof: Assume $statusCS[q] \neq (OUT, 0)$ is invariant in the interval $[t_1, t_2]$. Then clearly

[1] If p were to separate the check from the assignment, due to the asynchrony of processes, when p performs the assignment, j may have already returned to the NCS. This would lead to an incorrect algorithm.

$q@\{4..31\}$ is invariant in the interval $[t_1, t_2]$, and so q executes no part of its exit protocol in $[t_1, t_2]$. Since the only place where *false* can be assigned to $barricade[p][q]$ is in q's exit protocol, at line 33, it must be the case that *false* is never assigned to $barricade[p][q]$ in $[t_1, t_2]$. Hence, because *true* is assigned to $barricade[p][q]$ at time t_1, it follows that $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. □

The following corollary gives a necessary condition for a process to become conflict-free qualified in a certain time interval.

Corollary 5.2: Suppose process p, requesting session s_p, assigns *true* to $barricade[p][q]$ at line 7 at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q] \notin \{(OUT, 0), (REQ, s_p), (IN, s_p)\}$ is invariant in the interval $[t_1, t_2]$, then p cannot be conflict-free qualified in the interval $[t_1, t_2]$.

Proof: Assume $statusCS[q] \notin \{(OUT, 0), (REQ, s_p), (IN, s_p)\}$ is invariant in the interval $[t_1, t_2]$. Suppose, by way of contradiction, that p is conflict-free qualified at some point in the interval $[t_1, t_2]$. Thus p finishes executing the method CONFLICT-FREE() by time t_2. Now, by our assumption that $statusCS[q] \notin \{(OUT, 0), (REQ, s_p), (IN, s_p)\}$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 8 to true and thus adds q to $p.conflict\_set$ at line 9, for iteration $j = q$. Thus p busy-waits on $barricade[p][q]$ in the method CONFLICT-FREE() until it equals *false*. By Lemma 5.1, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Hence the method CONFLICT-FREE() cannot finish by time t_2, which contradicts the fact it does finish by time t_2. □

The following corollary gives a necessary condition for a process to be able to enter the CS in a certain time interval.

Corollary 5.3: Suppose process p, requesting session s_p, assigns *true* to $barricade[p][q]$ at line 20 at time t_1. Let t_2 be a time such that $t_2 \geq t_1$, and let s_q be a session such that $s_q \neq s_p$. If $statusCS[q] = (IN, s_q)$ is invariant in the interval $[t_1, t_2]$, then p cannot be critical-section qualified in the interval $[t_1, t_2]$.

Proof: Assume $statusCS[q] = (IN, s_q)$ is invariant in the interval $[t_1, t_2]$. Suppose,

by way of contradiction, that p is critical-section qualified at some point in the interval $[t_1, t_2]$. Thus p reaches line 25 by t_2. Now, by our assumption that $statusCS[q] = (IN, s_q)$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 21 to true, and thus busy-waits on $barricade[p][q]$ at line 22 until it equals *false*, for iteration $j = q$. By Lemma 5.1, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Thus p cannot reach line 25 by t_2, which contradicts the fact it does reach line 25 by t_2. □

Mutual Exclusion

Theorem 5.4: The algorithm in Figure 5.2 satisfies the mutual exclusion property.

Proof: Let p be a process that requests session s_p and q be a process that requests session s_q, where $s_p \neq s_q$. Suppose, by way of contradiction, that p and q are in the CS simultaneously at time t.

For a process to enter the CS it must become either mutex qualified, capture qualified, or conflict-free qualified. We now consider the different ways in which p and q could have entered the CS, and show that a contradiction arises in each case.

Case 1: Both processes are mutex qualified.

Due to the mutual exclusion property of $\mathbb{A}$, at most one process can be mutex qualified at a time. Hence this case is impossible.

Case 2: One process is capture qualified.

Without loss of generality, assume q is capture qualified.

Claim 5.4.1: Process p starts executing the loop at line 19 before q is captured.

Proof: Let time t_1 be the time when p assigns *true* to $barricade[p][q]$ at line 20. Suppose, by way of contradiction, that when p starts executing the loop at line 19, q has already been captured. Thus $statusCS[q] = (IN, s_q)$ is invariant in $[t_1, t]$. Hence by Corollary 5.3, p cannot be critical-section qualified in the interval $[t_1, t]$, contradicting the fact that p is in the CS at time t. □ (Claim 5.4.1)

Let r be the first process to capture q during q's current invocation. (Because pro-

cesses that capture q are mutex qualified, the ME property of $\mathbb{A}$ ensures that they capture q sequentially, so r is well defined.) So r is a mutex qualified process that requests s_q; moreover, by the choice of r, q cannot become capture qualified before being captured by r.

Now consider the different ways p could have entered the CS:

Case 2.1: Process p is mutex qualified.

Either p became mutex qualified before r became mutex qualified, or vice-versa. In the former case, r cannot become mutex qualified until after p exits the CS. Since q cannot enter the CS before r captures it, and r cannot capture q before r becomes mutex qualified, it follows p will exit the CS before q enters it. This contradicts our assumption that p and q are in the CS together at time t.

In the latter case, p becomes mutex qualified after r leaves the CS. Since r captures q before it leaves the CS, when p becomes mutex qualified and starts executing the loop at line 19, q will have been captured. This contradicts Claim 5.4.1.

Case 2.2: Process p is conflict-free qualified.

Claim 5.4.2: Process p starts executing the loop at line 6 before r writes (REQ, s_q) into $statusCS[r]$ at line 3.

Proof: Let t_1 be the time when p writes *true* to $barricade[p][r]$ at line 7, and let t_2 be the time when r exits the CS. Since r captures q before it exits the CS, it follows that if $t_2 < t_1$ then at the time when p reaches line 19, q will already have been captured. This would contradict Claim 5.4.1. Thus it must be the case $t_2 \geq t_1$. Now, suppose, by way of contradiction, that the claim is false. Hence $statusCS[r] \notin \{(OUT, 0), (REQ, s_p), (IN, s_p)\}$ is invariant in $[t_1, t_2]$. By Corollary 5.2, p cannot be conflict-free qualified in $[t_1, t_2]$. Since p is conflict-free qualified at time t by assumption, it must be the case that p is conflict-free qualified and reaches line 19 after time t_2. But this implies that q will have been captured when p reaches line 19, contradicting Claim 5.4.1. □ (Claim 5.4.2)

Claim 5.4.2 implies that p finishes executing the method MUTEXDOORWAY() at line 4

before r begins executing the method MUTEXDOORWAY() at line 4. Hence, by the FCFS property of $\mathbb{A}$, r becomes mutex qualified after p executes the method MUTEXABORT() at line 38. This and the facts that r captures q after becoming mutex qualified, and q enters the CS after being captured by r, imply that q enters the CS after p executes the method MUTEXABORT() at line 38. This contradicts the assumption that p and q are in the CS simultaneously at time t.

Case 2.3: Process p is capture qualified.

By Claim 5.4.1, p starts executing the loop at line 19 before q. Since p and q are both capture qualified, symmetrically, q starts executing the loop at line 19 before p. This is a contradiction.

Case 3: One process is conflict-free qualified, and the other is not capture qualified.

Without loss of generality, assume q is conflict-free qualified.

Claim 5.4.3: Process q starts executing the loop at line 6 before p writes (REQ, s_p) into $statusCS[p]$ at line 3.

Proof: Let t_1 be the time when q writes *true* to $barricade[q][p]$ at line 7. Suppose, by way of contradiction, that the claim is false. Hence $statusCS[p] \notin \{(OUT, 0), (REQ, s_q), (IN, s_q)\}$ is invariant during the interval $[t_1, t]$. By Corollary 5.2, q cannot be conflict-free qualified in the interval $[t_1, t]$, contradicting the fact that q is in the CS conflict-free qualified at time t. □ (Claim 5.4.3)

Now consider the different ways p could have entered the CS:

Case 3.1: Process p is mutex qualified.

By Claim 5.4.3, q starts executing the loop at line 6 before p executes line 3, and so q will finish executing the method MUTEXDOORWAY() at line 4 before p starts executing the method MUTEXDOORWAY() at line 4. Hence, due to the FCFS property of $\mathbb{A}$, p cannot become mutex qualified until q executes the method MUTEXABORT() at line 38, contradicting the assumption that p and q are in the CS simultaneously at time t.

Case 3.2: Process p is conflict-free qualified.

By Claim 5.4.3, q starts executing the loop at line 6 before p executes line 3, and so q executes line 3 before p does. Since p and q are both conflict-free qualified, symmetrically, p executes line 3 before q does. This is a contradiction.
Since each case leads to a contradiction, it follows that mutual exclusion holds. □

Lockout Freedom

Lemma 5.5: If process q is in a captured state then no process requesting a different session than q can be in a captured state at the same time.

Proof: Assume, by way of contradiction, that q and q' are requesting sessions s and s', respectively, where $s \neq s'$, and both are in a captured state at the same time. Since q and q' are in captured states, there must be mutex qualified processes u and u' that captured q and q', respectively, in their current invocations. Since processes that capture other processes are mutex qualified, and, by the mutual exclusion property of $\mathbb{A}$, at most one process can be mutex qualified at a time, it must be the case that one can totally order the times when some process captures another process. Because of this ability to totally order when processes capture other processes, we assume, without loss of generality, that u and u' are the first processes to capture q and q' in their current invocations. This implies q and q' cannot be in captured states in their current invocations before they are captured by u and u'. Now, by the mutual exclusion property of $\mathbb{A}$, at most one of u and u' can be mutex qualified at a time. Hence, it must be the case that either u is mutex qualified before u', or vice-versa. Without loss of generality, assume u is mutex qualified before u'.

We will now prove that u' will "notice" the captured process q and will not be able to become critical-section qualified until q has left the CS. So, u' will not capture q' at line 27 until after q has left the CS, contradicting that q and q' are in a captured state simultaneously.

More precisely, we define the following times: t_1 is the time when u captures q; t_2

is the time when u' becomes mutex qualified; t_3 is the time when u' writes *true* to $barricade[u'][q]$ at line 20; t_4 is the time when u' captures q'; and t_5 is the time when q and q' are simultaneously in a captured state.

Clearly $t_1 < t_2 < t_3 < t_4 \leq t_5$. Hence $statusCS[q] = (IN, s)$ during the interval $[t_3, t_4]$, since $statusCS[q] = (IN, s)$ during the interval $[t_1, t_5]$. It follows by Corollary 5.3 that u' cannot be critical-section qualified in the interval $[t_3, t_4]$. This means that $u'@\{25..30\}$ is false in the interval $[t_3, t_4]$. Since u' must reach line 25 to perform any capturing, it must be the case that u' cannot capture q' at time t_4. This contradicts the fact that u' captures q' at time t_4. □

Lemma 5.6: No process can wait forever at line 22.

Proof: Suppose, by way of contradiction, that p waits forever at line 22 on $barricade[p][q]$. Let s_p be the session that p is requesting and t_1 be the time when p writes *true* to $barricade[p][q]$ at line 20. At some time $t_2 > t_1$, p must have read $statusCS[q] = (IN, s_q)$ at line 21, where $s_q \neq s_p$. This implies that q is in a captured state and that $q@\{4..31\}$ at time t_2. Because $statusCS[q] = (IN, s_q)$, the method AWAIT-CAPTURE() will eventually terminate, and so q cannot block forever at lines 12..18.

Claim 5.6.1: Process q cannot wait forever at line 22.

Proof: Suppose, by way of contradiction, that q waits forever at line 22 on $barricade[q][q']$. Let t'_1 be the time when q writes *true* to $barricade[q][q']$ at line 20. At some time $t'_2 > t'_1$, q must have read $statusCS[q'] = (IN, s_{q'})$ at line 21, where $s_{q'} \neq s_q$. This implies that q' is in a captured state and $q'@\{4..31\}$ at time t'_2. For q to block forever on $barricade[q][q']$ after t'_2, it must be the case that *false* is never assigned to $barricade[q][q']$ after t'_1. Now, it is evident from examining the algorithm, that if q' reaches line 25 at or after t'_2, it will eventually execute its GME exit protocol in its entirety, and thus assign *false* to $barricade[q][q']$. Hence, to avoid contradiction, $q'@\{4..24\}$ must be invariant after and including time t'_2. That is, q' will be forever in a captured state requesting session $s_{q'}$ after and including t'_2. But we also know, by assumption, that

q will be forever in a captured state requesting session s_q at some time after t_2'. It follows q and q', requesting different sessions, will be in a captured state at the same time. This contradicts Lemma 5.5. □ (Claim 5.6.1)

By the preceding discussion, and Claim 5.6.1, q will not block forever on lines 12..18 or line 22. This, and the fact that $q@\{4..31\}$ at time t_2, imply that q will eventually execute its GME exit protocol at some time $t > t_2 > t_1$. That is, *false* will be assigned to $barricade[p][q]$ after time t_1. But this contradicts the fact that for p to be stuck forever at line 22, no process can write *false* to $barricade[p][q]$ after time t_1. □

Theorem 5.7: The algorithm in Figure 5.2 satisfies the lockout freedom property.

Proof: Suppose by way of contradiction that some set S of processes get stuck forever in their trying protocol. If they manage to reach line 19, then, by Lemma 5.6, they will eventually enter the CS. Hence they must be stuck forever at lines 12..18.

Claim 5.7.1: No process in S can be stuck forever in the method MUTEXWAITINGROOM().

Proof: $\mathbb{A}$ is an abortable mutual exclusion algorithm that satisfies the lockout freedom property. Hence, the only way for processes in S to get stuck forever in MUTEXWAITINGROOM() is if some process becomes mutex qualified and then does not execute the method MUTEXEXIT(); or, if some process starts executing MUTEXWAITINGROOM(), aborts execution of it, but then does not execute the method MUTEXABORT(). If a process becomes mutex qualified, then we know by Lemma 5.6 and by examining the algorithm it will eventually execute the method MUTEXEXIT(). Furthermore, the only way a process will abort execution of MUTEXWAITINGROOM() once started is if the process becomes conflict-free qualified or capture qualified. In this case, again by Lemma 5.6 and by examining the algorithm we see that such processes will eventually execute the method MUTEXABORT(). Hence no process in S can be stuck forever in the method MUTEXWAITINGROOM(). □ (Claim 5.7.1)

Claim 5.7.1 contradicts that processes in S are stuck forever at lines 12..18. Hence

lockout freedom holds. □

Concurrent Entering

Theorem 5.8: The algorithm in Figure 5.2 satisfies the concurrent entering property.

Proof: Let p be a process requesting session s_p that is in its GME waiting room at time t_1, and t_3 be the time when p enters the CS. We show that if there is no active process that is requesting a different session in the interval $[t_1, t_3]$, then p enters the CS in a bounded number of its own steps. There are two places in the GME waiting room where p can be waiting (i.e., executing a section of code that potentially could require an unbounded number of steps): $p@\{12..18\}$ and $p@22$. In each case we will show that p actually completes the corresponding section of code in a bounded number of steps during $[t_1, t_3]$:

Case 1: $p@\{12..18\}$.

In this case we will show that p completes the CONFLICT-FREE() method (and therefore the cobegin-coend statement in lines 12..18) in a bounded number of its own steps. Let t_2 be the time when p first reaches line 19. It suffices to prove that $\forall q \in p.conflict\_set$, $barricade[p][q] = false$ during the interval $[t_1, t_2]$. Suppose $q \in p.conflict\_set$. Then at line 8 at some time $t_q < t_1$, p read $statusCS[q] \notin \{(OUT, 0), (REQ, s_p), (IN, s_p)\}$. This implies that at time t_q, $q@\{4..31\}$. Since there are no active processes requesting a session other than s_p at time t_1, it follows that q executes its exit protocol and thus assigns $false$ to $barricade[p][q]$ in the interval (t_q, t_1). Now, the only process that can set $barricade[p][q]$ to $true$ is process p. Process p's last assignment of $true$ to $barricade[p][q]$ occurred at a time $t < t_q$, and its next assignment occurs at a time $t > t_2$. Therefore, $barricade[p][q] = false$ during $[t_1, t_2]$, as required.

Case 2: $p@22$

It suffices to prove that for any q such that $p@22$ in iteration $j = q$ of the loop starting at line 19, $barricade[p][q] = false$ during $[t_1, t_3]$. So, assume for some q that $p@22$

in iteration $j = q$. It follows that at some time $t_q < t_1$, p read $statusCS[q] = (IN, s_q)$ at line 21, where $s_p \neq s_q$. This implies that at time t_q, q@{4..31}. Since there can be no active processes requesting a session other than s_p at time t_1, it follows that q executes its exit protocol and thus assigns *false* to $barricade[p][q]$ in the interval (t_q, t_1). The only process that can set $barricade[p][q]$ to *true* is process p. Process p's last assignment of *true* to $barricade[p][q]$ occurred at a time $t < t_q$, and its next assignment cannot occur until a time $t > t_3$. Therefore, $barricade[p][q] = false$ during $[t_1, t_3]$, as required. □

Bounded Exit

Theorem 5.9: The algorithm in Figure 5.2 satisfies the bounded exit property.

Proof: Since the underlying FCFS abortable ME algorithm satisfies the bounded exit property, it is evident by inspection of the algorithm that there are no unbounded loops in the exit protocol. This implies that the bounded exit property holds. □

5.1.4 RMR Complexity

Theorem 5.10: The algorithm in Figure 5.2 has RMR complexity $O(N)$ under the DSM model.

Proof: Each of the loops at lines 6, 19, 26, and 32 makes exactly $N - 1$ remote memory references. Also, there exists a FCFS abortable mutual exclusion algorithm that has RMR complexity $O(\min(k, \log N))$ [9], where k is the point contention. Outside of the said loops and the FCFS abortable mutual exclusion algorithm, there are no remote memory references. Hence the resulting algorithm makes $O(N)$ remote memory references. □

5.2 An Algorithm With Only Read/Write Operations

An undesirable property of the algorithm in the preceding section is that it makes use of COMPARE_AND_SWAP in the capturing mechanism. The need for this type of instruction can be illustrated by considering the implementation of the capturing mechanism at an abstract level, as follows. Clearly a process that executes the capturing mechanism must only capture processes requesting the same session as itself. This means that a process p that is attempting to decide whether to capture a process q must first read the session that q is requesting. If q requests the same session as p, p will subsequently capture q. However, if the read and the capture are not one atomic operation, then process q has the opportunity to return to the NCS — or possibly proceed to another invocation of the algorithm — between the time when p reads q's session and the time when p captures q. Unless the capturing mechanism takes this into account, the algorithm may fail to satisfy any number of the GME properties.

Figure 5.5 Header for Algorithm in Figure 5.6

```
type
    SessionNode = record session: integer; passage: integer end

shared variables:
    statusCS: array[1..N] of SessionNode init (0,0)
    capture: array[1..N] of integer init 0
    barricade: array[1..N][1..N] of boolean

/* capture[p], barricade[p][j] is stored in process p's memory, for each j. */

private variables:
    mysession: integer /* Session that p wants to attend */
    conflict_set: set of integer
    passage: integer init 0
    l_status: SessionNode/* for storing statusCS[j] locally */
    l_capture: integer /* for storing capture[j] locally */
```

Figure 5.6 An Algorithm With Only Read/Write Operations; Process $p \in \{1..N\}$

```
 1: loop
 2:   NCS
 3:   passage := passage + 1;
 4:   statusCS[p] := (mysession, passage);
 5:   MutexDoorway();
 6:   conflict_set := ∅;
 7:   for j = 1 to N do
 8:     barricade[p][j] := true;
 9:     if statusCS[j].session ∉ {0, mysession} then
10:       conflict_set := conflict_set ∪ {j};
11:     end if
12:   end for
13:   cobegin
14:       MutexWaitingRoom();
15:       ||
16:       CONFLICT-FREE(); /* defined in Figure 5.3 */
17:       ||
18:       AWAIT-CAPTURE(); /* defined in Figure 5.7 */
19:   coend /* when one co-routine terminates, go to the next line */
20:   for j = 1 to N do
21:     barricade[p][j] := true;
22:     l_capture := capture[j];
23:     if ∃s ∉ {0, mysession} : statusCS[j] = (s, l_capture) then
24:       await ¬barricade[p][j];
25:     end if
26:   end for
27:   if mutex qualified then
28:     for j ∈ {1..N} \ {p} do
29:       l_status := statusCS[j];
30:       if l_status.session = mysession then
31:         capture[j] := l_status.passage;
32:       end if
33:     end for
34:   end if
35:   CS
36:   statusCS[p] := (0, 0);
37:   for j = 1 to N do
38:     barricade[j][p] := false;
39:   end for
40:   if mutex qualified then
41:     MutexExit();
42:   else
43:     MutexAbort();
44:   end if
45: end loop
```

Figure 5.7 Method AWAIT-CAPTURE() for Algorithm in Figures 5.6, 5.9

```
1: await capture[p] = passage;
```

The use of COMPARE_AND_SWAP makes the read of q's session and p's capture of

q atomic, thus avoiding the above problem. However, the preceding discussion also suggests an alternative way for implementing the capturing mechanism without using strong synchronization primitives: if, when p reads q's session, it also reads a unique id associated with the current invocation that q is executing, then when p captures q, it can also inform q that it is being captured for a particular invocation. This allows q to decide whether it was captured by p in the current invocation or in a prior invocation, and behave accordingly.

Process q can keep track of the unique id by making use of an unbounded variable that is incremented at the start of each invocation. We refer to this unique id as the "passage".

The algorithm in Figure 5.6 makes use of the preceding mechanism. We do not analyze this algorithm in any depth, since it turns out that by embellishing the capturing mechanism slightly more, we can eliminate the need for unbounded variables. The resulting algorithm is described and analyzed in detail in the next section.

5.3 An Algorithm With Bounded Variables

The algorithm we describe in this section (shown in Figure 5.9) is a subtle modification of the algorithm in the preceding section. The modification allows for the use of bounded instead of unbounded variables in the capturing mechanism. A more detailed description of this mechanism is given below.

The capturing mechanism is at lines 29..36 of the algorithm. A process p that executes the capturing mechanism scans all processes to see if anyone is currently requesting entry into the CS for the same session it requests. Process p does this by reading $statusCS[j]$ into a private variable, l_status, for each process $j \neq p$, and then checking to see if $l\_status.session = s_p$. If p finds such a process, it assigns to the variable $capture[j]$ the value of $l\_status.passage$. That is, p assigns to $capture[j]$ the value of $statusCS[j].passage$

that p reads prior to checking whether j requests the same session. This is the point at which we consider the "capture" to take place. Now, if a process j is stuck in its GME waiting room at lines 14..20 when all of this happens, then j can complete the method AWAIT-CAPTURE() (see Figure 5.7), because the value that p assigns to *capture*[j] is equal to the value of j.*passage*.

There are two conditions that must hold for p to invoke the capturing mechanism (see the conjunction at line 29). The first condition requires that p is mutex qualified. This condition was necessary in the preceding two algorithms to prevent starvation (as described in Section 5.1, page 71), and is necessary here for the same reason. The second condition requires that there be conflict. The presence of conflict ensures that when a process p_1 is about to capture a process p_2, p_2 cannot execute an unbounded number of invocations. This allows our capturing mechanism to use only bounded variables.

It is not immediately obvious that the algorithm behaves in the manner just described, due to the complex nature of what "capturing" means. The formal definition of what it means for one process to "capture" another is stated below.

The RMR complexity of this algorithm under the DSM model is $O(N)$.

Figure 5.8 Header for Algorithm in Figure 5.9

```
type
    SessionNode = record session: integer; passage: {0,1} end

shared variables:
    statusCS: array[1..N] of SessionNode init (0,0)
    capture: array[1..N] of {-1,0,1} init -1
    barricade: array[1..N][1..N] of boolean

/* capture[p], barricade[p][j] are stored in process p's memory, for each j. */

private variables:
    mysession: integer /* Session that p wants to attend */
    conflict_set: set of integer
    passage: {0,1} init 0 /* passage flag */
    l_status: SessionNode/* for storing statusCS[j] locally */
    l_capture: {-1,0,1} /* for storing capture[j] locally */
```

Figure 5.9 An Algorithm With Bounded Variables; Process $p \in \{1..N\}$

```
 1: loop
 2:   NCS
 3:   passage := ¬passage;
 4:   capture[p] := −1;
 5:   statusCS[p] := (mysession, passage);
 6:   MutexDoorway();
 7:   conflict_set := ∅;
 8:   for j = 1 to N do
 9:     barricade[p][j] := true;
10:     if statusCS[j].session ∉ {0, mysession} then
11:       conflict_set := conflict_set ∪ {j};
12:     end if
13:   end for
14:   cobegin
15:     MutexWaitingRoom();
16:     ||
17:     CONFLICT-FREE(); /* defined in Figure 5.3 */
18:     ||
19:     AWAIT-CAPTURE(); /* defined in Figure 5.7 */
20:   coend /* when one co-routine terminates, go to the next line */
21:   for j = 1 to N do
22:     barricade[p][j] := true;
23:     l_status := statusCS[j];
24:     l_capture := capture[j];
25:     if ∃s ∉ {0, mysession} : statusCS[j] = l_status = (s, l_capture) then
26:       await ¬barricade[p][j];
27:     end if
28:   end for
29:   if mutex qualified ∧ ∃j : statusCS[j].session ∉ {0, mysession} then
30:     for j ∈ {1..N} \ {p} do
31:       l_status := statusCS[j];
32:       if l_status.session = mysession then
33:         capture[j] := l_status.passage;
34:       end if
35:     end for
36:   end if
37:   CS
38:   statusCS[p] := (0, 0);
39:   for j = 1 to N do
40:     barricade[j][p] := false;
41:   end for
42:   if mutex qualified then
43:     MutexExit();
44:   else
45:     MutexAbort();
46:   end if
47: end loop
```

We proceed as follows: we give some notation and definitions relative to the algorithm, provide a line by line commentary of the algorithm, present a proof of correctness, and

then supply a proof for why the algorithm has RMR complexity $O(N)$.

5.3.1 Definitions and Notation

When dealing with values of type *SessionNode* in the algorithm, we sometimes use the notation (a, b), where a is the value of the *session* field, and b is the value of the *passage* field.

To refer to the fields of *SessionNode* variables separately, we use dot notation. That is, if v is a variable of type *SessionNode*, then $v.session$ refers to the value of the *session* field, and $v.passage$ refers to the value of the *passage* field.

GME Doorway: The doorway consists of lines 3..13. Even though this algorithm is not FCFS, we define as the doorway an initial portion of the trying protocol that a process completes in a bounded number of its own steps.

GME Waiting Room: The waiting room consists of lines 14..36

GME Exit Protocol: The exit protocol consists of lines 38..46

As with previous algorithms we occasionally use definitions in which, to be strictly correct, we would need to refer to the invocations that the processes are executing. However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run. This comment applies to the following definitions.

Mutex Qualified Process: Process p is **mutex qualified** iff $p@\{21..42\}$ and p reached line 21 because it completed execution of the method MUTEXWAITINGROOM().

Conflict-free Qualified Process: Process p is **conflict-free qualified** iff $p@\{21..38\}$ and p reached line 21 because it completed execution of the method CONFLICT-FREE().

Capture Qualified Process: Process p is **capture qualified** iff $p@\{21..38\}$ and p reached line 21 because it completed execution of the method AWAIT-CAPTURE().

Critical-Section Qualified Process: Process p is **critical-section qualified** iff $p@\{29..37\}$. Intuitively, if a process p is critical-section qualified then it can enter the

CS in a bounded number of its own steps.

Capture: Suppose a mutex qualified process p executes line 33 at time t for iteration $j = q$. Let time t' be the time when p last executed line 31 before t. We say that process p **captures** process q at time t iff $q@\{6..38\}$ is invariant in $[t', t]$.

We also define a different notion of capture, called β-capture. The reason for defining two separate notions of capture is because it turns out that a crucial lemma is easier to prove using the notion of β-capture. This will be explained in more detail when the correctness of the algorithm is proven below.

β-Capture: Suppose a mutex qualified process p executes line 33 at time t for iteration $j = q$. We say that process p **β-captures** process q at time t iff the following conditions hold at time t: (1) $q@\{5..38\}$, and (2) $p.l\_status.passage = q.passage$.

To see the difference between β-capture and regular capture, consider the following: suppose a process p executes line 31 at some time t', and p then β-captures a process q at some time $t > t'$. Then $q@\{5..38\}$ and $p.l\_status.passage = q.passage$ must be true at time t, but q may have executed any even number of invocations of the algorithm between time t' and t. (The number of invocations must be even since $p.l\_status.passage = q.passage$ and q complements its *passage* in each iteration.) However, if p captures q at time t, then $q@\{6..38\}$ must be invariant in $[t', t]$, and so q must still be in the same invocation at time t as it was at time t'. It is also true in this case that at time t, $p.l\_status.passage = q.passage$, but it is not part of the formal definition of capture because it follows from the fact that $q@\{6..38\}$ is invariant in $[t', t]$.

Captured State: Process p is in a **captured state** at time t iff there exists a mutex qualified process q that captures p at some time $t' \leq t$, and $p@\{6..38\}$ is invariant in the interval $[t', t]$. Intuitively, if p is in a captured state in some invocation I_p of the algorithm, then there is a mutex qualified process q that captures p while p is executing invocation I_p.

5.3.2 Line by Line Commentary of the Algorithm

Line 3: Process p toggles the value of its private variable *passage* upon leaving the NCS.

Line 4: Process p resets $capture[p]$ to -1 to indicate that no other process has captured p.

Line 5: Process p declares that it is requesting session *mysession*, and sets $statusCS[p].passage$ to the current value of *passage*.

Line 6: Process p executes the method MUTEXDOORWAY() of the underlying mutual exclusion algorithm.

Line 7..13: Process p scans all processes to check if there are any processes requesting a different session. If there are then p will add the identifiers of these processes to its *conflict_set*. When p executes the method CONFLICT-FREE(), p will attempt to wait on the variable $barricade[p][j]$ for each $j \in conflict\_set$.

Line 14..20: Process p waits until it completes one of the co-routines contained within these lines. After completing one of the co-routines, p is either mutex qualified, conflict-free qualified, or capture qualified, and proceeds to line 21.

Line 21..28: Process p checks if there are any previously captured processes, requesting a session different from itself, still inside the CS. If there are, then p waits until they exit.

Line 29..36: If process p, requesting session s_p, is mutex qualified, and there is conflict, then p will attempt to capture any process j also requesting session s_p.

Line 37: Process p executes the CS.

Line 38: Process p declares that it has left the CS.

Line 39..41: There may be other processes waiting for p to leave the CS. By assigning false to $barricade[j][p]$ for each process j, p ensures that any process j waiting for p does not have to wait for p any longer.

Line 42..46: If p is mutex qualified, then p executes the method MUTEXEXIT() before returning to the NCS. Otherwise, if p is not mutex qualified, then p previously aborted execution of the method MUTEXWAITINGROOM() at line 15. For this reason p executes

the method MUTEXABORT() before returning to the NCS.

5.3.3 Proof of Correctness

Before proving the four GME properties (mutual exclusion, lockout freedom, concurrent entering, and bounded exit), we need to prove the following essential facts:

1. While a process q is captured, any process requesting a session different from q will not be able to enter the CS.

2. If a process q is capture qualified in some invocation I_q, then there exists a mutex qualified process p requesting the same session as q that captures q while q is executing invocation I_q and before q is capture qualified.

3. If a process p evaluates the condition at line 25 to be true for some invocation $j = q$, and p subsequently blocks at line 26, then q is in a captured state.

The first two facts are used in our proof of mutual exclusion, and the third fact is used in our proof of both lockout freedom and concurrent entering. The first fact is formally stated and proven in two parts, in Lemma 5.12 and Corollary 5.15. It turns out the second fact is quite difficult to prove directly. We prove it in two steps. The first step involves proving a variant of Fact 2 where the word "capture" is replaced by "β-capture". This is done in Lemma 5.16. For the second step we prove that if a process p β-captures a process q, then p captures q. This is formally stated and proven in Lemma 5.17. Fact 3 is formally stated and proven in Lemma 5.20.

We start by proving an elementary Lemma that will be useful later.

Lemma 5.11: For any t, if p@{6..38} at time t, then $statusCS[p].passage = p.passage$ at time t.

Proof: Assume p@{6..38} at time t. Let v be the value p assigns to $p.passage$ when it executes line 3. Clearly p assigns v to $statusCS[p].passage$ when p executes

line 5, and furthermore, $p.passage$ remains unchanged after p executes line 3. Since neither $statusCS[p].passage$ or $p.passage$ is updated in lines 6..37, it follows that both $statusCS[p].passage = v$ and $p.passage = v$ at time t, and so $statusCS[p].passage = p.passage$ at time t. □

The next lemma says that between the time a process becomes captured and the time it leaves the CS, the passage flag of its $statusCS$ variable is fixed and equal to its capture flag. (Note that while $statusCS[q]$ is exclusively under the control of q, $capture[q]$ is also under the control of processes that capture q — see line 33.)

Lemma 5.12: If a process p captures a process q, requesting a session s_q, at some time t_1, and q's next execution of line 38 after t_1 occurs at some time t_2, then $statusCS[q] = (s_q, capture[q])$ is invariant during $[t_1, t_2)$.

Proof: Assume a process p captures a process q, requesting a session s_q, at time t_1. Further assume that q's next execution of line 38 occurs at time $t_2 > t_1$. Let t_0 be the last time p executes line 31 prior to t_1.

Claim 5.12.1: $q@\{6..38\}$ is invariant in $[t_0, t_2)$.

Proof: Process p captures q at time t_1, and so $q@\{6..38\}$ is invariant in $[t_0, t_1]$, by definition. Process q next executes line 38 after time t_1 at time t_2, by assumption. The result follows. □ (Claim 5.12.1)

Claim 5.12.2: $statusCS[q].session = s_q$ is invariant in $[t_0, t_2)$.

Proof: $statusCS[q].session$ can be updated only by q in line 5. This observation and Claim 5.12.1 imply the result. □ (Claim 5.12.2)

Let k be the value q assigned to $q.passage$ when q last executed line 3 prior to t_0.

Claim 5.12.3: $statusCS[q].passage = k$ is invariant in $[t_0, t_2)$.

Proof: $statusCS[q].passage$ can be updated only by q in line 5. This and Claim 5.12.1 imply the result. □ (Claim 5.12.3)

Claim 5.12.4: $capture[q] = k$ at time t_1.

Proof: Process p captures q at time t_1, and so p assigns to $capture[q]$ the value of $p.l\_status.passage$ at time t_1. $p.l\_status.passage$ was previously assigned the value of $statusCS[q].passage$ at time t_0. By Claim 5.12.3, this value is k. Hence the value p assigns to $capture[q]$ at time t_1 is k. □ (Claim 5.12.4)

Claim 5.12.5: $capture[q] = k$ is invariant in $[t_1, t_2)$.

Proof: Suppose, by way of contradiction, that $capture[q] = k$ is not invariant in the interval $[t_1, t_2)$. By Claim 5.12.4, $capture[q] = k$ at time t_1, and so there must be a mutex qualified process r that updates $capture[q]$ to some value other than k in (t_1, t_2). We know that p is mutex qualified, so, by the mutual exclusion property of $\mathbb{A}$, either r is mutex qualified before p, or after. In the former case, r will update $capture[q]$ before time t_1, which contradicts that it updates $capture[q]$ in (t_1, t_2). Hence r is mutex qualified after p, which means r is mutex qualified after t_1. Also, to update $capture[q]$ in (t_1, t_2), r must execute line 33 before t_2. It follows that r's execution of lines 31..33 occurs entirely within (t_1, t_2). From this and Claim 5.12.3, it must be the case that r sets $r.l\_status.passage = k$ when r executes line 31. This implies r sets $capture[q] = k$ when r executes line 33. This contradicts the fact that r updates $capture[q]$ to a value different than k. □ (Claim 5.12.5)

Claims 5.12.2, 5.12.3, and 5.12.5 imply that $statusCS[q] = (s_q, capture[q])$ is invariant in $[t_1, t_2)$. □

The next lemma gives a sufficient condition for when $barricade[p][q] = true$. Two corollaries to this lemma give us sufficient conditions for when a process cannot be conflict-free qualified or critical-section qualified.

Lemma 5.13: Suppose a process p assigns $true$ to $barricade[p][q]$ at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q] \neq (0, 0)$ is invariant in the interval $[t_1, t_2]$, then $barricade[p][q] = true$ is invariant in interval $[t_1, t_2]$.

Proof: Assume $statusCS[q] \neq (0, 0)$ is invariant in the interval $[t_1, t_2]$. Then clearly $q@\{6..38\}$ is invariant in the interval $[t_1, t_2]$, and so q executes no part of its exit protocol

in $[t_1, t_2]$. Since the only place where *false* can be assigned to $barricade[p][q]$ is in q's exit protocol, at line 40, it must be the case that *false* is never assigned to $barricade[p][q]$ in $[t_1, t_2]$. Hence, because *true* is assigned to $barricade[p][q]$ at time t_1, it follows that $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. □

Corollary 5.14: Suppose a process p, requesting session s_p, assigns *true* to $barricade[p][q]$ at line 9 at time t_1. Let t_2 be a time such that $t_2 \geq t_1$. If $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, then p cannot be conflict-free qualified in the interval $[t_1, t_2]$.

Proof: Assume $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$. Suppose, by way of contradiction, that p is conflict-free qualified at some point in the interval $[t_1, t_2]$. Thus p finishes executing the method CONFLICT-FREE() by time t_2. Now, by our assumption that $statusCS[q].session \notin \{0, s_p\}$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 10 to be true and thus adds q to $p.conflict\_set$ at line 11, for iteration $j = q$. Thus p busy-waits in the method CONFLICT-FREE() until $barricade[p][q] = false$. By Lemma 5.13, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Hence the method CONFLICT-FREE() cannot finish by time t_2, which contradicts the fact it does finish by time t_2. □

Corollary 5.15: Suppose a process p, requesting session s_p, assigns *true* to $barricade[p][q]$ at line 22 at time t_1. Let t_2 be a time such that $t_2 \geq t_1$, and let s_q be a session such that $s_q \notin \{0, s_p\}$. If $statusCS[q] = (s_q, capture[q])$ is invariant in the interval $[t_1, t_2]$, then p cannot be critical-section qualified in the interval $[t_1, t_2]$.

Proof: Assume $statusCS[q] = (s_q, capture[q])$ is invariant in the interval $[t_1, t_2]$. Suppose, by way of contradiction, that p is critical-section qualified at some point in the interval $[t_1, t_2]$. Thus p reaches line 29 by t_2. Now, by our assumption that $statusCS[q] = (s_q, capture[q])$ is invariant in the interval $[t_1, t_2]$, p evaluates the condition at line 25 to be true, and thus busy-waits at line 26 until $barricade[p][q] = false$, for iteration $j = q$. By Lemma 5.13, however, $barricade[p][q] = true$ is invariant in $[t_1, t_2]$. Thus p cannot

reach line 29 by t_2, which contradicts the fact it does reach line 29 by t_2. □

The following lemma shows that for a process to be capture qualified there must be some other process that β-captures it.

Lemma 5.16: If a process q is capture qualified during some invocation I_q, then there exists a process p that β-captures q before q is capture qualified and while q is executing invocation I_q.

Proof: Assume q is capture qualified during some invocation I_q. Then q reaches line 21 when the method AWAIT-CAPTURE() terminates. For this to happen, it must be the case that q reads $capture[q] = q.passage$ sometime during the execution of the method AWAIT-CAPTURE(). Since q assigns $capture[q] = -1$ at line 4, and $q.passage \in \{0, 1\}$, it follows that there must be some process that sets $capture[q] = q.passage$ after q executes line 4 but before q is capture qualified. That is, there is a process p that sets $capture[q] = q.passage$ when $q@\{5..20\}$. The only place in the algorithm where p can set $capture[q] = q.passage$ is at line 33. It follows that p β-captures q before q is capture qualified and while q is executing invocation I_q. □

Consider an arbitrary execution of the algorithm. Since only one process can be mutex qualified at a given time, by the mutual exclusion property of $\mathbb{A}$, we can totally order the times in this execution when processes are mutex qualified. Furthermore, for a process p to β-capture another process, p must be mutex qualified. Hence, we can also totally order the times when some process β-captures another process. This allows us to prove our next lemma.

Lemma 5.17: If a process p β-captures a process q at a time t, then p captures q at time t.

Proof: Suppose, by way of contradiction, that the lemma is false. Thus, there is a run α where the precondition of the lemma holds but the consequent is violated.

By the discussion preceding the lemma, we can totally order the times in α when β-captures occur. Let $t_1 < t_2 < t_3 < \ldots$ be this ordering, and let k be the smallest

positive integer such that a capture does not occur at time t_k. Let p be the process that performs the β-capture at time t_k, and let q be the process that is β-captured at time t_k. (That is, p β-captures q at time t_k.) By definition of k, p does not capture q at time t_k.

Since process p β-captures process q at time t_k, it follows that p executes line 33 at time t_k, by definition of β-capture. Let v be the value of $p.l\_status.passage$ that p assigns to $capture[q]$ at this time. Note that v is also the value of $q.passage$ at time t_k, by definition of β-capture.

Let t' be the last time prior to t_k when p executes line 31.

Claim 5.17.1: $q@\{6..38\}$ is true at time t'.

Proof: Suppose the claim is false. Then $statusCS[q] = (0, 0)$ when p executes line 31, and so $l\_status.session = 0$ when p evaluates the condition at line 32. Since no process sets $mysession = 0$, it follows that p will evaluate the condition at line 32 to be false. Hence p does not execute line 33 at time t_k, which contradicts that p does execute line 33 at time t_k. □ (Claim 5.17.1)

To reach a contradiction, and hence complete the proof of the lemma, we show, with the following claim, that p does in fact capture q at time t_k.

Claim 5.17.2: $q@\{6..38\}$ is invariant in $[t', t_k]$.

Proof: Suppose, by way of contradiction, that the claim is false. By Claim 5.17.1, $q@\{6..38\}$ is true at time t', and we know $q@\{5..38\}$ is true at time t_k, by definition of β-capture. So, by the assumption that $q@\{6..38\}$ is not invariant in $[t', t_k]$, it must be the case that q executes different invocations at time t' and time t_k. We denote the invocation q executes at time t' by $I_{t'}$, and the invocation q executes at time t_k by I_t. Now, the value that p assigns to $l\_status.passage$ at time t' must be v; if not, then p will not set $capture[q] = v$ at time t_k, which contradicts the assumption that it does. Hence $statusCS[q].passage = v$ at time t', and so by Lemma 5.11, $q.passage = v$ at time t'. Now, by examining the algorithm, we see that at the start of every invocation of the algorithm, at line 3, $passage$ is changed to $\overline{passage}$. We further note that line 3 is the only place in

the algorithm where *passage* is changed. This, and the fact that $q.passage = v$ at time t' and time t_k, imply that there exists an invocation of q, I_{t_mid}, that occurs after $I_{t'}$ but before I_t. There are three cases, depending on how q is qualified to enter the CS in I_{t_mid}. We show that a contradiction arises in each case.

Case 1: Process q executes through the CS in I_{t_mid} mutex qualified.

The mutual exclusion property of $\mathbb{A}$ and the fact that p is mutex qualified during $[t', t_k]$ imply that q cannot be mutex qualified in $[t', t_k]$. From this and the fact that I_{t_mid} is contained entirely in the interval (t', t_k), it follows that q cannot execute through the CS in I_{t_mid} mutex qualified. This contradicts the assumption that it does.

Case 2: Process q executes through the CS in I_{t_mid} capture qualified.

By Lemma 5.16, there exists a process u that β-captures q while q is executing invocation I_{t_mid}, but before q is capture qualified.

First, we note that u cannot be p. This is because for p to β-capture q while q is executing I_{t_mid}, p must execute line 33 at some point in the interval (t', t_k). But p executes line 31 at time t' and next executes line 33 at time t_k, which makes the preceding impossible.

Now, for u to β-capture q, u must be mutex qualified, and so u is mutex qualified during I_{t_mid}. Since p is mutex qualified during $[t', t_k]$, it follows that u cannot be mutex qualified in $[t', t_k]$. But I_{t_mid} occurs entirely during (t', t_k), and so u cannot be mutex qualified during I_{t_mid}. This contradicts the fact that it is.

Case 3: Process q executes through the CS in I_{t_mid} conflict-free qualified.

First, assume that process p requests a session s_p in the invocation in which it β-captures q.

Now, we know p evaluates the condition at line 29 to be true. So there must be some process r, requesting a session $s_r \neq s_p$, such that $r@\{6..38\}$ when p checks if $statusCS[r].session \notin \{0, s_p\}$. Let t'' be the time when p does this check.

We will prove that during invocation I_{t_mid} that r is outside of the NCS and that q requests a session that conflicts with the session that process r requests. We will further prove that q "notices" this fact, and thus does not become conflict-free qualified, providing us with the desired contradiction.

Sub-Claim 5.17.2.1: Process p executes line 5 before r executes line 7.

Proof: Suppose the claim is false. Then p starts the method MUTEXDOORWAY() at line 6 after r finishes the method MUTEXDOORWAY() at line 6. By the FCFS property of $\mathbb{A}$, p is mutex qualified after r executes MUTEXEXIT() or MUTEXABORT(). This implies that $r@\{6..38\}$ is not true when p is mutex qualified. This contradicts the fact $r@\{6..38\}$ at time t'', a time when p is mutex qualified. □ (Sub-Claim 5.17.2.1)

Sub-Claim 5.17.2.2: Process r is not mutex qualified at any point in $[t'', t_k]$.

Proof: Since p is mutex qualified during $[t'', t_k]$, it follows from the mutual exclusion property of $\mathbb{A}$ that r cannot be mutex qualified at any point in $[t'', t_k]$. □ (Sub-Claim 5.17.2.2) □

Sub-Claim 5.17.2.3: Process r is not conflict-free qualified at any point in $[t'', t_k]$.

Proof: Suppose, by way of contradiction, that r is conflict-free qualified at some point in $[t'', t_k]$. Let t_a be the time when r executes line 9 for iteration $j = p$. By our assumption that r is conflict-free qualified at some point in $[t'', t_k]$, $t_a < t_k$. From this and Sub-Claim 5.17.2.1, $p@\{6..38\}$ is true in $[t_a, t_k]$. This implies $statusCS[p].session \notin \{0, s_r\}$ during $[t_a, t_k]$. It then follows by Corollary 5.14 that r cannot be conflict-free qualified in $[t_a, t_k]$, and so r cannot be conflict-free qualified in $[t'', t_k]$. □ (Sub-Claim 5.17.2.3)

Sub-Claim 5.17.2.4: Process r is not capture qualified at any point in $[t'', t_k]$.

Proof: Suppose, by way of contradiction, that r is capture qualified at some point in $[t'', t_k]$. By Lemma 5.16, there is some mutex qualified process u that β-captures r in r's current invocation before r is capture qualified. Either u is mutex qualified before p, or after p.

In the latter case, where u is mutex qualified after p, it follows that u is mutex qualified at some time after t_k, and so the time when u β-captures r is after t_k. Since r is capture qualified after u β-captures r, it follows that r is not capture qualified in $[t'', t_k]$. This contradicts the assumption that r is capture qualified in $[t'', t_k]$.

Now consider the former case, where u is mutex qualified before p. Let t_u be the time when u β-captures r, and let t_p be the time when p executes line 22 for iteration $j = r$. Since u is mutex qualified before p, u β-captures r before p is mutex qualified. From this, and the fact that p executes line 22 after p is mutex qualified, it follows that $t_u < t_p$. Also, it is clear that $t_p < t''$, since p evaluates the condition at line 29 after it executes line 22. As well, $t'' < t_k$, since p executes line 33 after it evaluates the condition at line 29. Hence $t_u < t_p < t'' < t_k$.

Recall our assumption that k is the smallest positive integer such that the k^{th} β-capture in α occurs at some time t_k, and yet a capture does not occur at time t_k. Either $k = 1$ or $k > 1$.

Consider the case where $k = 1$. Process u cannot β-capture r at time t_u, since $t_u < t_k$ and t_k is the first time in α some process β-captures another process. This contradicts that u β-captures r at time t_u.

Consider the case where $k > 1$. Since $t_u < t_k$, it follows that for some positive $m < k$, t_u is the m^{th} time in α some process β-captures another process. So, by the minimality of k, it follows that u captures r at time t_u. Now, $r@\{6..38\}$ at time t'', and so r has yet to execute line 38 at time t''. From this and Lemma 5.12, it must be the case that $statusCS[r] = (s_r, capture[r])$ is invariant in $[t_u, t'']$. Since $t_p \in [t_u, t'']$, it must also be the case that $statusCS[r] = (s_r, capture[r])$ is invariant in $[t_p, t'']$. Hence, by Corollary 5.15, p cannot be critical-section qualified in $[t_p, t'']$. This contradicts the fact that p is at line 29 at time t''. $\square$ (Sub-Claim 5.17.2.4)

Sub-Claim 5.17.2.5: Process q requests session s_p during invocation I_{t_mid}.

Proof: Suppose, by way of contradiction, that q requests some session $s_q \neq s_p$ during I_{t_mid}. Let t_a be the time q executes line 9 in invocation I_{t_mid} for iteration $j = p$. Since invocation I_{t_mid} occurs entirely in (t', t_k), it must be the case that $t_a \in (t', t_k)$. From this and the fact that $statusCS[p].session \notin \{0, s_q\}$ during $[t', t_k]$, it follows that $statusCS[p].session \notin \{0, s_q\}$ during $[t_a, t_k]$. Thus, by Corollary 5.14, q cannot be conflict-free qualified in $[t_a, t_k]$. This contradicts the fact that q must execute through the CS as a conflict-free qualified process in invocation I_{t_mid}. □ (Sub-Claim 5.17.2.5)

The fact that $r@\{6..38\}$ is true at time t'', coupled with Sub-Claims 5.17.2.2, 5.17.2.3, and 5.17.2.4 implies that $r@\{6..20\}$ in $[t'', t_k]$. Now, let t_a be the time q executes line 9 in invocation I_{t_mid} for iteration $j = r$. Since invocation I_{t_mid} occurs entirely during (t', t_k), it must be the case that $t_a \in (t', t_k)$. From this and the fact that $r@\{6..20\}$ in $[t'', t_k]$, it follows that $r@\{6..20\}$ in $[t_a, t_k]$. This, and the fact that $s_p \neq s_r$, imply that $statusCS[r].session \notin \{0, s_p\}$ during $[t_a, t_k]$. From this, Sub-Claim 5.17.2.5, and Corollary 5.14, it follows that q cannot be conflict-free qualified in $[t_a, t_k]$. This contradicts that q executes through the CS in I_{t_mid} as a conflict-free qualified process.

Cases 1, 2, and 3 all lead to a contradiction, and so the claim holds. □ (Claim 5.17.2)

By Claim 5.17.2, p captures q at time t_k. This contradicts the fact that p does not capture q at time t_k. □

Lemma 5.18: If a process p, requesting session s_p, captures another process q, requesting session s_q, then $s_p = s_q$ (i.e., p requests the same session as q).

Proof: Assume p captures another process q. Let t_1 be the time when p executes line 31, and t_2 be the time when p executes line 33. That is, t_2 is the time when p captures q. $q@\{6..38\}$ is invariant in $[t_1, t_2]$, by definition of capture. From this and the fact that p evaluates the condition at line 32 to be true, $statusCS[q].session = s_p$ is invariant in $[t_1, t_2]$. But $statusCS[q].session$ is the session q is requesting, and so $s_q = s_p$. □

Lemma 5.19: If a process q, requesting session s, is capture qualified during some invocation I_q, then there exists a process p, also requesting session s, that captures q before q is capture qualified and while q is executing invocation I_q.

Proof: Assume a process q is capture qualified during some invocation I_q. By Lemma 5.16, there exists a process p that β-captures q at some time t before q is capture qualified and while q is executing invocation I_q. By Lemma 5.17, p captures q at time t, and by Lemma 5.18, p and q request the same session. The result follows. □

Lemma 5.20: If a process p evaluates the condition at line 25 to be true at some time t for iteration $j = q$, and $barricade[p][q] = true$ at time t, then q is in a captured state at time t.

Proof: Assume process p evaluates the condition at line 25 to be true at some time t for iteration $j = q$, and $barricade[p][q] = true$ at time t. Let t_1 be the time p executes 22, t_2 be the time p executes line 23, and t_3 be the time p executes line 24. Clearly $t_1 < t_2 < t_3 < t$.

Claim 5.20.1: $q@\{6..38\}$ is invariant in $[t_2, t]$.

Proof: Suppose, by way of contradiction, that the claim is false. Now, since p evaluates the condition at line 25 to be true at time t, it must be the case that $q@\{6..38\}$ is true at time t_2. For the same reason, $q@\{6..38\}$ is true at time t. From these facts, and the assumption that $q@\{6..38\}$ is not invariant in $[t_2, t]$, it must be the case that q executes the exit protocol in its entirety at least once in (t_2, t). This implies q will set $barricade[p][q] = false$ at some time in (t_2, t). Since $barricade[p][q] = true$ at time t, by assumption, there is a process that sets $barricade[p][q] = true$ at some time in (t_2, t) after q sets $barricade[p][q] = false$. But the only process that can set $barricade[p][q] = true$ is process p, and p cannot set $barricade[p][q] = true$ at any point in $[t_2, t]$. This contradicts the fact that some process sets $barricade[p][q] = true$ in (t_2, t). □ (Claim 5.20.1)

Let t' be the last time prior to t_2 when q executes line 4.

Claim 5.20.2: $q@\{5..38\}$ is invariant in $(t', t]$.

Proof: Time t' is the last time prior to t_2 when q executes line 4, by assumption. Furthermore, by Claim 5.20.1, $q@\{6..38\}$ is invariant in $[t_2, t]$. Thus $q@\{5..38\}$ is invariant in $(t', t]$. □ (Claim 5.20.2)

Let v be the value p reads into *l_capture* from $capture[q]$ at time t_3. By the assumption that p evaluates the condition at line 25 to be true at time t, and the fact that $statusCS[q].passage \in \{0, 1\}$ is invariant throughout the algorithm's execution, it follows that $v \in \{0, 1\}$. From this and the fact that q sets $capture[q] = -1$ at time t', it follows that there must exist a process r that sets $capture[q] = v$ by executing line 33 at some time $t'' \in (t', t_3)$.

Claim 5.20.3: $q@\{5..38\}$ at time t''.

Proof: Since $t'' \in (t', t_3)$, clearly $t'' \in (t', t]$. The result follows from this and Claim 5.20.2. □ (Claim 5.20.3)

Claim 5.20.4: $q.passage = v$ at time t''.

Proof: By Claim 5.20.1, $q@\{6..38\}$ at time t_2, and so, by Lemma 5.11, $statusCS[q].passage = q.passage$ at time t_2. Since p assigns $p.l\_status.passage$ the value of $statusCS[q].passage$ at time t_2, and p evaluates the condition at line 25 to be true at time t, it follows that $q.passage = v$ at time t_2. Furthermore, $q.passage$ is invariant as long as $q@\{5..38\}$, and so, by Claim 5.20.2 and the fact that $t_2 \in (t', t]$, $q.passage = v$ in $(t', t]$. Since $t'' \in (t', t]$, it follows that $q.passage = v$ at time t''. □ (Claim 5.20.4)

Process r sets $capture[q] = v$ at time t'', by assumption. That is, $r.l\_status.passage = v$ at time t''. This and Claim 5.20.4 imply that $r.l\_status.passage = q.passage$ at time t''. From this, and Claim 5.20.3, it follows that r β-captures q at time t''. By Lemma 5.17, r captures q at time t''. Hence $q@\{6..38\}$ at time t'', by definition of capture. From this, Claim 5.20.2, and the fact that $t'' \in (t', t]$, it follows that $q@\{6..38\}$ is invariant in $[t'', t]$. Hence q is in a captured state at time t. □

Mutual Exclusion

Theorem 5.21: The algorithm in Figure 5.9 satisfies the mutual exclusion property.

Proof: Let p be a process that requests a session s_p and q be a process that requests a session s_q, where $s_p \neq s_q$. Suppose, by way of contradiction, that p and q are in the CS simultaneously at time t.

For a process to enter the CS, it must be either mutex qualified, capture qualified, or conflict-free qualified. We now consider the different ways in which p and q could have entered the CS, and show that a contradiction arises in each case.

Case 1: Both processes are mutex qualified.

Due to the mutual exclusion property of $\mathbb{A}$, at most one process can be mutex qualified at a time. Hence this case is impossible.

Case 2: One process is capture qualified.

Without loss of generality, assume q is capture qualified. By Lemma 5.19, there exists a mutex qualified process r, requesting session s_q, that captures q before q is capture qualified and while q is executing its current invocation. Let time t_r be the time when r captures q. By Lemma 5.12, $statusCS[q] = (s_q, capture[q])$ is invariant in $[t_r, t]$.

Claim 5.21.1: Process p starts executing the loop at line 21 before q is captured by r.

Proof: Suppose, by way of contradiction, that when p starts executing the loop at line 21, q has already been captured by r. Let time t_1 be the time when p assigns *true* to $barricade[p][q]$ at line 22. By assumption, q is captured before t_1, and so $t_1 \in [t_r, t]$. From this and the fact that $statusCS[q] = (s_q, capture[q])$ is invariant in $[t_r, t]$, it follows that $statusCS[q] = (s_q, capture[q])$ is invariant in $[t_1, t]$. Thus by Corollary 5.15, p cannot be critical-section qualified in the interval $[t_1, t]$, contradicting the fact that p is in the CS at time t. □ (Claim 5.21.1)

Now consider the different ways p enters the CS:

Case 2.1: Process p is mutex qualified.

Either p is mutex qualified before r, or after. In the former case, r is mutex qualified after p exits the CS. Now, we know q enters the CS after q is capture qualified, and q is capture qualified after r captures q. So q enters the CS after r captures q. This, together with the fact that r captures q after r is mutex qualified, implies that q enters the CS after r is mutex qualified. This, in turn, implies that q enters the CS after p exits the CS. This contradicts our assumption that p and q are in the CS together at time t.

In the latter case, p is mutex qualified after r leaves the CS. Since r captures q before r leaves the CS, when p is mutex qualified and executes the loop at line 21, q will have been captured. This contradicts Claim 5.21.1.

Case 2.2: Process p is conflict-free qualified.

Claim 5.21.2: Process p starts executing the loop at line 8 before r executes line 5 of the invocation of r in which r captures q.

Proof: Suppose, by way of contradiction, that the claim is false. Let t_1 be the time when p writes *true* to $barricade[p][r]$ at line 9, and let t_2 be the time when r exits the CS. Since r captures q before it exits the CS, it follows that if $t_2 < t_1$ then at the time when p reaches line 21, q will already have been captured, which contradicts Claim 5.21.1. Thus it must be the case $t_2 \geq t_1$. Also, by the assumption that r executes line 5 before p starts executing the loop at line 8, and the fact that r requests session $s_q \neq s_p$, it follows that $statusCS[r].session \notin \{0, s_p\}$ is invariant in $[t_1, t_2]$. Hence, by Corollary 5.14, p cannot be conflict-free qualified in $[t_1, t_2]$. Since p is conflict-free qualified at time t by assumption, it must be the case that p is conflict-free qualified and reaches line 21 after time t_2. But this implies that q is captured when p reaches line 21, contradicting Claim 5.21.1. □ (Claim 5.21.2)

Claim 5.21.2 implies that p finishes executing the method MUTEXDOORWAY() at line 6 before r begins executing the method MUTEXDOORWAY() at line 6. Hence, by the FCFS property of $\mathbb{A}$, r is mutex qualified after p executes the method MUTEXABORT() at line 45. Now, we know q enters the CS after q is capture qualified, and q is capture

qualified after r captures q. We also know that r captures q after r is mutex qualified. This implies q enters the CS after r is mutex qualified. But r is mutex qualified after p executes line 45, and so q enters the CS after p leaves the CS. This contradicts the assumption that p and q are in the CS simultaneously at time t.

Case 2.3: Process p is capture qualified.

By Claim 5.21.1, p starts executing the loop at line 21 before q. Since p and q are both capture qualified, symmetrically, q starts executing the loop at line 21 before p. This is a contradiction.

Case 3: One process is conflict-free qualified, and the other is not capture qualified.

Without loss of generality, assume q is conflict-free qualified.

Claim 5.21.3: Process q starts executing the loop at line 8 before p executes line 5 of its current invocation.

Proof: Suppose, by way of contradiction, that the claim is false. Let t_1 be the time when q writes *true* to $barricade[q][p]$ at line 9. By assumption, p executes line 5 before t_1, and so $statusCS[p].session \notin \{0, s_q\}$ is invariant during the interval $[t_1, t]$. Thus, by Corollary 5.14, q cannot be conflict-free qualified in the interval $[t_1, t]$, contradicting the fact that q is in the CS conflict-free qualified at time t. □ (Claim 5.21.3)

Now consider the different ways p enters the CS:

Case 3.1: Process p is mutex qualified.

By Claim 5.21.3, q starts executing the loop at line 8 before p executes line 5, and so q will finish executing the method MUTEXDOORWAY() at line 6 before p starts executing the method MUTEXDOORWAY() at line 6. Hence, due to the FCFS property of $\mathbb{A}$, p cannot become mutex qualified until q executes the method MUTEXABORT() at line 45, contradicting the assumption that p and q are in the CS simultaneously at time t.

Case 3.2: Process p is conflict-free qualified.

By Claim 5.21.3, q starts executing the loop at line 8 before p executes line 5, and so q executes line 5 before p does. Since p and q are both conflict-free qualified, symmetrically,

p executes line 5 before q does. This is a contradiction.

Since each case leads to a contradiction, it follows that mutual exclusion holds. □

Lockout Freedom

To prove the lockout freedom property we first prove some elementary lemmas. The first of these states that processes in a captured state cannot block in the AWAIT-CAPTURE() method.

Lemma 5.22: If a process q is in a captured state during some invocation I_q, then q cannot block forever in the method AWAIT-CAPTURE() during invocation I_q.

Proof: Assume q is in a captured state during some invocation I_q at some time t. Suppose, by way of contradiction, that q blocks forever in the method AWAIT-CAPTURE() during invocation I_q. By the assumption that q is in a captured state at time t, there exists a process p that captured q while q was executing invocation I_q at some time $t_1 \leq t$. By the assumption that q blocks forever in the method AWAIT-CAPTURE(), the time when q next executes line 38 after time t_1 is ∞. Thus, by Lemma 5.12, $statusCS[q] = (s_q, capture[q])$ is invariant during $[t_1, \infty)$. Now, $q@\{6..38\}$ at time t_1 by definition of capture, and so, because of the fact that q never executes line 38 after t_1, it follows that $q@\{6..38\}$ is invariant in $[t_1, \infty)$. By this fact and Lemma 5.11, $statusCS[q].passage = q.passage$ is invariant in $[t_1, \infty)$. This, and the fact that $statusCS[q].passage = capture[q]$ is invariant in $[t_1, \infty)$, imply that $capture[q] = q.passage$ is invariant in $[t_1, \infty)$. But the method AWAIT-CAPTURE() terminates when $capture[q] = q.passage$, and so q cannot block forever in the method AWAIT-CAPTURE() during invocation I_q. This contradicts the assumption that q does block forever in the method AWAIT-CAPTURE(). □

We will next show that all processes that are simultaneously in a captured state request the same session.

Lemma 5.23: If process q is in a captured state then no process requesting a different session than q can be in a captured state at the same time.

Proof: Assume, by way of contradiction, that q and q' are requesting sessions s and s', respectively, where $s \neq s'$, and both are in a captured state at the same time. Since q and q' are in captured states, there must be mutex qualified processes u and u' that captured q and q', respectively, in their current invocations. Since processes that capture other processes are mutex qualified, and, by the mutual exclusion property of $\mathbb{A}$, at most one process can be mutex qualified at a time, it must be the case that one can totally order the times when some process captures another process. Because of this ability to totally order when processes capture other processes, we assume, without loss of generality, that u and u' are the first processes to capture q and q' in their current invocations. This implies q and q' cannot be in captured states in their current invocations before they are captured by u and u'. Now, by the mutual exclusion property of $\mathbb{A}$, at most one of u and u' can be mutex qualified at a time. Hence, it must be the case that either u is mutex qualified before u', or vice-versa. Without loss of generality, assume u is mutex qualified before u'.

We will now prove that u' will "notice" the captured process q and will not be able to become critical-section qualified until q has left the CS. So, u' will not capture q' at line 33 until after q has left the CS, contradicting that q and q' are in a captured state simultaneously.

More precisely, we define the following times: t_1 is the time when u captures q; t_2 is the time when u' becomes mutex qualified; t_3 is the time when u' writes *true* to $barricade[u'][q]$ at line 22; t_4 is the time when u' captures q'; and t_5 is the time when q and q' are simultaneously in a captured state.

Clearly $t_1 < t_2 < t_3 < t_4 \leq t_5$, and $q@\{6..38\}$ is invariant in $[t_1, t_5]$. By Lemma 5.12, $statusCS[q] = (s, capture[q])$ is invariant in the interval $[t_1, t_5]$, and thus in the interval $[t_3, t_4]$. Now, by Lemma 5.18, u' requests session $s' \neq s$. From this, the fact that $statusCS[q] = (s, capture[q])$ in $[t_3, t_4]$, and Corollary 5.15, it follows that u' cannot be critical-section qualified at any point in $[t_3, t_4]$. This means that $u'@\{29..37\}$ is false in

the interval $[t_3, t_4]$. Since u' must reach line 29 to perform any capturing, it must be the case that u' cannot capture q' at time t_4. This contradicts the fact that u' captures q' at time t_4. □

Lemma 5.24: If, for some process p, p@26 at some time t for iteration $j = q$, and $barricade[p][q] = true$ at time t, then p evaluated the condition at line 25 to be true at some time $t' < t$ and $barricade[p][q] = true$ at time t'.

Proof: Assume p@26 at some time t, for $j = q$, and $barricade[p][q] = true$ at time t. Then clearly p evaluated the condition at line 25 to be true at some time $t' < t$. Now, suppose, by way of contradiction, that $barricade[p][q] = false$ at time t'. Since $barricade[p][q] = true$ at time t by assumption, there is some process that sets $barricade[p][q] = true$ at some time in the interval $(t', t]$. The only process that can do this, however, is process p, and p has no opportunity to set $barricade[p][q] = true$ in lines 25..26. This contradicts the fact that some process sets $barricade[p][q] = true$ in $(t', t]$. □

Lemma 5.25: No process can wait forever at line 26.

Proof: Suppose, by way of contradiction, that p waits forever at line 26 on $barricade[p][q]$. Let s_p be the session that p is requesting and t_1 be the time when p writes *true* to $barricade[p][q]$ at line 22.

We will show that eventually q will execute its exit protocol, setting $barricade[p][q] = false$, at some time after t_1, and thus contradict that p waits forever on $barricade[p][q]$ at line 26.

By Lemma 5.24 and the assumption that p waits forever at line 26, at some time $t_2 > t_1$, p evaluates the condition at line 25 to be true for iteration $j = q$, and $barricade[p][q] = true$ at time t_2.Thus, by Lemma 5.20, q is in a captured state at t_2, and so q@{6..38} at t_2. By Lemma 5.22 and the fact that q is in a captured state at t_2, q cannot block forever in the method AWAIT-CAPTURE(). So q cannot block forever at lines 14..20.

Claim 5.25.1: Process q does not wait forever at line 26.

Proof: Suppose, by way of contradiction, that q waits forever at line 26 on

$barricade[q][q']$. Let t'_1 be the time when q writes *true* to $barricade[q][q']$ at line 22. By Lemma 5.24 and the assumption that q waits forever at line 26, at some time $t'_2 > t'_1$, q evaluates the condition at line 25 to be true for iteration $j = q'$, and $barricade[q][q'] = true$ at time t'_2. Thus, by Lemma 5.20, q' is in a captured state at time t'_2, and so $q'@\{6..38\}$ at time t'_2. For q to block forever on $barricade[q][q']$ after t'_2, it must be the case that *false* is never assigned to $barricade[q][q']$ after t'_1. This is because only q can set $barricade[q][q'] = true$, and it does not do so anywhere in lines 23..26. Now, it is evident from examining the algorithm that if q' reaches line 29 at or after t'_2, it will eventually execute its GME exit protocol in its entirety, and thus assign *false* to $barricade[q][q']$. Hence, to avoid contradiction, $q'@\{6..28\}$ must be invariant from time t'_2 and on. That is, q' will be forever in a captured state requesting session $s_{q'}$ from time t'_2 and on. But we also know, by assumption, that q will be forever in a captured state requesting session s_q at some time after t'_2. It follows that q and q', requesting different sessions, will be in a captured state at the same time. This contradicts Lemma 5.23. □ (Claim 5.25.1)

By the preceding discussion, and Claim 5.25.1, q will not block forever at lines 14..20 or line 26. This, together with the fact that $q@\{6..38\}$ at time t_2, implies that q will eventually execute its GME exit protocol at some time $t > t_2 > t_1$. That is, *false* will be assigned to $barricade[p][q]$ after time t_1. But this contradicts the fact that for p to be stuck forever at line 26, no process can write *false* to $barricade[p][q]$ after time t_1. □

Theorem 5.26: The algorithm in Figure 5.9 satisfies the lockout freedom property.

Proof: Suppose, by way of contradiction, that some set S of processes get stuck forever in their trying protocol. If they manage to reach line 21, then by Lemma 5.25 they will eventually enter the CS. Hence they must be stuck forever at lines 14..20.

Claim 5.26.1: No process in S can be stuck forever in the method MUTEXWAITINGROOM().

Proof: $\mathbb{A}$ is an abortable mutual exclusion algorithm that satisfies the lockout freedom property. Hence the only way for processes in S to get stuck forever

in MUTEXWAITINGROOM() is if some process becomes mutex qualified and then does not execute the method MUTEXEXIT(); or, if some process starts executing MUTEXWAITINGROOM(), aborts execution of it, but then does not execute the method MUTEXABORT(). If a process becomes mutex qualified, then we know by Lemma 5.25 and by examining the algorithm it will eventually execute the method MUTEXEXIT(). Furthermore, the only way a process will abort execution of MUTEXWAITINGROOM() once started is if the process becomes conflict-free qualified or capture qualified. In this case, again by Lemma 5.25 and by examining the algorithm we see that such processes will eventually execute the method MUTEXABORT(). Hence no process in S can be stuck forever in the method MUTEXWAITINGROOM(). □ (Claim 5.26.1)

Claim 5.26.1 contradicts that processes in S are stuck forever at lines 14..20. Hence lockout freedom holds. □

Concurrent Entering

Theorem 5.27: The algorithm in Figure 5.9 satisfies the concurrent entering property.

Proof: Let p be a process requesting session s_p that is in its GME waiting room at time t_1, and t_3 be the time when p enters the CS. We show that if there is no active process that is requesting a different session in the interval $[t_1, t_3]$, then p enters the CS in a bounded number of its own steps. There are two places in the GME waiting room where p can be waiting (i.e., executing a section of code that potentially could require an unbounded number of steps): $p@\{14..20\}$ and $p@26$. In each case we will show that p actually completes the corresponding section of code in a bounded number of steps during $[t_1, t_3]$.

Case 1: $p@\{14..20\}$.

In this case we will show that p completes the CONFLICT-FREE() method (and therefore the cobegin-coend statement in lines 14..20) in a bounded number of its own steps. Let t_2 be the time when p first reaches line 21. It suffices to prove that $\forall q \in p.conflict\_set$,

$barricade[p][q] = false$ during the interval $[t_1, t_2]$. Suppose $q \in p.conflict\_set$. Then at line 10 at some time $t_q < t_1$, p read $statusCS[q].session \notin \{0, s_p\}$. This implies that at time t_q, q@{6..38}. Since there can be no active processes requesting a session other than s_p at time t_1, it follows that q executes its exit protocol and thus assigns *false* to $barricade[p][q]$ in the interval (t_q, t_1). Now, the only process that can set $barricade[p][q]$ to *true* is process p. Process p's last assignment of *true* to $barricade[p][q]$ occurred at a time $t < t_q$, and its next assignment occurs at a time $t > t_2$. Therefore, $barricade[p][q] = false$ during $[t_1, t_2]$, as required.

Case 2: p@26

It suffices to prove that for any process q such that p@26 in iteration $j = q$ of the loop starting at line 21, $barricade[p][q] = false$ during $[t_1, t_3]$. So, assume for some q that p@26 in iteration $j = q$. It follows that at some time $t_q < t_1$, p evaluated the condition at line 25 to be true. We claim that $barricade[p][q] = false$ at some time during $[t_q, t_1)$. This is obvious if $barricade[p][q] = false$ at time t_q. If not, by Lemma 5.20, q is in a captured state at time t_q. This means that q@{6..38} at time t_q. Since there can be no active processes requesting a session other than s_p at time t_1, it follows that q executes the exit protocol in its entirety and thus assigns *false* to $barricade[p][q]$ in the interval (t_q, t_1).

Hence $barricade[p][q] = false$ at some time in the interval $[t_q, t_1)$. Now, the only process that can set $barricade[p][q] = true$ is process p. Process p's last assignment of *true* to $barricade[p][q]$ occurs before t_q, and its next assignment cannot occur until after t_3. Therefore $barricade[p][q] = false$ during $[t_1, t_3]$, as required. □

Bounded Exit

Theorem 5.28: The algorithm in Figure 5.9 satisfies the bounded exit property.

Proof: Since the underlying FCFS abortable ME algorithm satisfies the bounded exit property, it is evident by inspection of the algorithm that there are no unbounded loops

in the exit protocol. This implies that the bounded exit property holds. □

5.3.4 RMR Complexity

Theorem 5.29: The algorithm in Figure 5.9 has RMR complexity $O(N)$ under the DSM model.

Proof: Each of the loops at lines 8, 21, 30, and 39 makes exactly $N-1$ remote memory references. Also, there exists a FCFS abortable mutual exclusion algorithm that has RMR complexity $O(\min(k,\log\ N))$ [9], where k is the point contention. Outside of the said loops and the FCFS abortable mutual exclusion algorithm, there are no remote memory references. Hence the resulting algorithm makes $O(N)$ remote memory references. □

Chapter 6

Local-Spin GME Under The CC Model

Up until this point we have only considered the group mutual exclusion problem under the DSM model. In this chapter, we explore the group mutual exclusion problem under the CC model. In particular, we are interested in whether the algorithms in the preceding chapters, all of which have RMR complexity $O(N)$ under the DSM model, are optimal with respect to RMR complexity under the CC model. It turns out that this is not the case, because, as we will show, there is an M-session group mutual exclusion algorithm that has RMR complexity $o(N)$ under the CC model for certain values of M.

Recall that in the CC model each process has a (local) cache associated with it and there is a global store that is remote to all processes. Whenever a process accesses the global store, the process makes a remote memory reference. In particular, whenever a process reads a variable that is not in its cache or reads a variable that is invalid (i.e., a variable for which there exists a newer copy in the global store), it must copy the variable from the global store to its cache. Whenever a process reads a variable that it has already cached, and the variable is not invalid, the process simply reads the variable from the cache. Furthermore, whenever a process writes a variable, it updates both its cache and

the global store, and, as a side-effect of this update, it invalidates the variable in other processes' caches. (This is the only means by which a variable in a process's cache can become invalid.)

The algorithms that we develop in this chapter make use of the read-modify-write primitives COMPARE_AND_SWAP, ATOMIC_ADD, and FETCH_AND_ADD. We make the assumption that a failed COMPARE_AND_SWAP (i.e., a COMPARE_AND_SWAP in which the shared variable being read is not modified) does not result in any variables being invalidated. This assumption is only needed when reasoning about the RMR complexity results in this chapter. The correctness of the algorithms does not depend on it.

In this chapter we develop an M-session GME algorithm under the CC Model that is in the form of a reduction to a FCFS abortable mutual exclusion algorithm, $\mathbb{A}$. We develop this algorithm in two steps. In the first step we develop a 2-session GME algorithm, and show that the RMR complexity of this algorithm is $O(f(N))$, where $f(N)$ is the RMR complexity of $\mathbb{A}$. Since there exists a FCFS abortable mutual exclusion algorithm with RMR complexity $O(\min(k, \log N))$ [9], where k is the point contention, it follows that there exists a 2-session GME algorithm that also has RMR complexity $O(\min(k, \log N))$. This result beats the $\Omega(N)$ RMR lower bound for 2-session GME under the DSM model proven in Theorem 3.1.

In the second step we create an arbitration tree of height $\lfloor \log M \rfloor$ that uses the 2-session algorithm as a building block. The "tree" is actually presented in the form of a recursive algorithm. This results in an M-session GME algorithm with RMR complexity $O(\log M \cdot f(N))$.[1]

[1] This two-step approach is similar to the one taken by Yang and Anderson [18] in developing their N-process mutual exclusion algorithm. (Yang and Anderson use as a building block a 2-process ME algorithm with $O(1)$ RMR complexity and then create an arbitration tree of height $\lfloor \log N \rfloor$, so that their resulting N-process algorithm has RMR complexity $O(\log N)$.)

6.1 2-Session GME With Low RMR Complexity

Figure 6.1 2-Session GME Algorithm; Process $p \in \{1..N\}$

```
type
    GateNode= record tag: {0..N}; state: {OPEN,CLOSED} end

shared variables:
    gate: array[1..2] of GateNode init (0, CLOSED)
    active: array[1..2] of integer init 0

private variables:
    l_active: array[1..2] of integer /* private version of the active variable */
    l_gate: GateNode /* private version of the gate[mysession] variable */
    mysession: {1, 2} /* Session that p wants to attend */

 1: loop
 2:    NCS
 3:    ATOMIC_ADD(active[mysession],1);
 4:    l_gate := gate[mysession];
 5:    if l_gate ≠ (0, CLOSED) then
 6:       COMPARE_AND_SWAP(gate[mysession],l_gate,(0, CLOSED));
 7:    end if
 8:    COMPARE_AND_SWAP(gate[mysession],(0, OPEN),(0, CLOSED));
 9:    MUTEXDOORWAY();
10:    l_active[mysession] := FETCH_AND_ADD(active[mysession], N + 1);
11:    cobegin
12:         MUTEXWAITINGROOM();
13:         ||
14:         CONFLICT-FREE(); /* defined in Figure 6.2 */
15:    coend *when one coroutine terminates, go to the next line
16:    ATOMIC_ADD(active[mysession],−(N + 1));
17:    CS
18:    l_active[mysession] := FETCH_AND_ADD(active[mysession], −1);
19:    if l_active[mysession] mod (N + 1) = 1 ∧ l_active[mysession] ≥ (N + 1) then
20:       gate[mysession] := (p, CLOSED);
21:       if active[mysession] mod (N + 1) = 0 then
22:          COMPARE_AND_SWAP(gate[mysession],(p, CLOSED),(0, OPEN));
23:       end if
24:    end if
25:    if mutex qualified then
26:       MUTEXEXIT();
27:    else
28:       MUTEXABORT();
29:    end if
30: end loop
```

Figure 6.2 Method CONFLICT-FREE() for Algorithm in Figure 6.1

```
1: if l_active[mysession] mod (N + 1) > 0 then
2:    await gate[mysession] = (0, OPEN);
3: end if
```

6.1.1 Introduction

In this section we develop a 2-session GME algorithm. It will be used later (in Section 6.2) to construct the M-session GME algorithm. Our 2-session GME algorithm uses as a building block an arbitrary FCFS abortable mutual exclusion algorithm, $\mathbb{A}$. The 2-session algorithm uses only $O(1)$ RMRs in addition to those used by $\mathbb{A}$. Thus if $\mathbb{A}$ has RMR complexity $O(f(N))$, so does our 2-session GME algorithm.

The algorithm is presented in Figure 6.1. It satisfies mutual exclusion, lockout freedom, concurrent entering, bounded exit, and FCFS, under the condition that processes request only session 1 or 2. It does not satisfy either FIFE or strong concurrent entering.

We assume that each process sets *mysession* to either 1 or 2 before leaving the NCS.

We proceed as follows: we give some definitions and notation relative to the 2-session algorithm; we provide an informal description of how the algorithm works, followed by a line by line commentary of the algorithm; finally, we present a proof of correctness, and then analyze the RMR complexity of algorithm.

6.1.2 Definitions and Notation

GME Doorway: The doorway consists of lines 3..10.

GME Waiting Room: The waiting room consists of lines 11..16

GME Trying Protocol: The trying protocol consists of lines 3..16

GME Exit Protocol: The exit protocol consists of lines 18..29

As with previous algorithms we occasionally use definitions in which, to be strictly correct, we would need to refer to the invocations that the processes are executing. However, for simplicity, we assume that the invocations in question are the "current" (i.e., the latest) ones with respect to some point in a given run. This comment applies to the following definitions.

Mutex Qualified Process: Process p is **mutex qualified** iff $p@\{16..25\}$ and p reached

line 16 because it completed execution of the method MUTEXWAITINGROOM().

Conflict-free Qualified Process: Process p is **conflict-free qualified** iff $p@\{16..18\}$ and p reached line 16 because it completed execution of the method CONFLICT-FREE().

If $s \in \{1, 2\}$ is one of the sessions, $\overline{s}$ denotes the *other* session; more precisely, $\overline{s} = 3 - s$.[2]

6.1.3 Informal Description of the Algorithm

In this section we provide a very high-level overview of the algorithm. We start by describing the shared variables used in the algorithm, and then describe how they are used.

Associated with each session $s \in \{1, 2\}$ are two shared variables: a counter, $active[s]$, and a spin-lock, $gate[s]$. Each process increments by one the counter of the session it is requesting when it enters the trying protocol (at line 3), and atomically reads and decrements that counter by one when it leaves the CS (at line 18). In addition, each process atomically reads and increments by $N + 1$ the counter of the *other* session when it enters the GME waiting room (at line 10), and decrements that counter by the same amount when it leaves the GME waiting room (at line 16). Thus, if $active[s] = a(N + 1) + b$, where $0 \leq b \leq N$, then there are exactly b processes requesting s in lines 4..18, and exactly a processes requesting $\overline{s}$ in lines 11..16. We will later see in more detail how this variable is used, but it is clear that by reading this variable a process can get some idea of how many active processes request each of the sessions.

To enter the CS, a process executes the doorway of the underlying FCFS abortable ME algorithm (at line 9) and then executes as co-routines the GME waiting room of that algorithm and the CONFLICT-FREE() procedure (lines 11..15). The latter allows a process to enter the CS if it detects that there are no active processes requesting the

[2]This is analogous to the use of the notation $\overline{s}$ in previous chapters, except that there this notation was used for $s \in \{0, 1\}$ (rather than $s \in \{1, 2\}$) and so $\overline{s}$ was defined as $1 - s$ (rather than $3 - s$).

other session. We now discuss how CONFLICT-FREE() works. As alluded to earlier, a process requesting s can detect that there are no active processes currently requesting $\overline{s}$ by checking that $active[\overline{s}]$ **mod** $(N+1) = 0$. Unfortunately, a mechanism for detecting absence of conflicting requests that spins on the value of $active[\overline{s}]$ does not give us the desired complexity of only $O(1)$ RMR in the CC model.

For this reason we need a different mechanism for processes to detect the absence of conflicting requests. The shared variable $gate[s]$ provides this mechanism. At a high level, the idea behind this mechanism is that as soon as a process requesting $\overline{s}$ leaves the NCS, it "closes" $gate[s]$ by setting $gate[s] = (0, CLOSED)$ (lines 4..8), thereby preventing processes requesting s from entering the CS through the co-routine CONFLICT-FREE(). We refer to this as the "gate closing" phase of the trying protocol. The subsequent opening of $gate[s]$ is accomplished as follows: As each process requesting $\overline{s}$ leaves the CS, it checks whether (a) it is the last such process to do so, and (b) there are active processes requesting the other session that are in the GME waiting room. Process p, requesting session $\overline{s}$, does this by looking at the old value v of $active[\overline{s}]$ when it decremented it (at line 19): If v **mod** $(N+1) = 1$, then p is the last one requesting $\overline{s}$ to leave the CS; if $v \geq (N+1)$ then there are active processes requesting s that are in the GME waiting room. If both of these conditions are satisfied, p attempts to "open" $gate[s]$ (lines 20..23).

We now examine the reasons why the preceding conditions need to be satisfied before p opens $gate[s]$. The need for the first condition, that p is the last process that leaves the CS, is fairly obvious: if p weren't the last process requesting $\overline{s}$ to leave the CS, then opening $gate[s]$ is premature and could result in the violation of the mutual exclusion property. The second condition, that a process requesting session s is in the GME waiting room, has to do with the fact that we want our algorithm to make $O(1)$ RMRs outside of the FCFS abortable ME algorithm. (In fact, removing the check of the second condition does not break any of the correctness properties of the algorithm; it only affects the RMR complexity.) So as not to digress too much from our high-level description of the

algorithm, we will return to this point later.

Process p should not "open" $gate[s]$ by using a simple assignment statement to set $gate[s] = (0, OPEN)$. If p were to do this, then there is the possibility that between the time when p reads $active[\overline{s}]$ (at line 18) and the time when it writes $gate[s] = (0, OPEN)$, some other process p' requesting session $\overline{s}$ executes through the trying protocol and goes into the GME waiting room. Process p then sets $gate[s] = (0, OPEN)$, and returns to the NCS. We now have a situation where $gate[s]$ is open, and a process requesting session $\overline{s}$ (i.e., p') has gotten past the "gate closing" phase of the algorithm (lines 4..8) without ensuring that $gate[s]$ is closed. This is a dangerous situation; after p' enters the CS, since $gate[s]$ is open, processes requesting session s can also enter the CS via the CONFLICT-FREE() method, thus violating the mutual exclusion property.

The preceding could be avoided if p, when it leaves the CS, could somehow do the following atomically: read and decrement the variable $active[\overline{s}]$, check if the value read meets the appropriate conditions, and if it does, set $gate[s] = (0, OPEN)$. This would ensure that other processes requesting session $\overline{s}$ (e.g., p' as described above) could not race into the GME waiting room before p has the opportunity to set $gate[s] = (0, OPEN)$. However, there exist no realistic synchronization primitives that would allow p to perform all the required operations on $active[\overline{s}]$ and $gate[s]$ atomically, which leads us to explore whether there are other mechanisms that achieve an equivalent effect.

There is in fact such a mechanism. It is located in lines 20..23. The first thing that p does in this fragment of the algorithm is to set $gate[s] = (p, CLOSED)$. This essentially "tags" the $gate[s]$ variable with p's identifier. Using this tag, p, at some later point, can determine if anyone else has written to $gate[s]$ since p last wrote to it. Process p then repeats the check of whether it is the last active process to leave the CS (line 21). If the recheck fails, then p knows that it should not continue trying to open $gate[s]$ to processes requesting session s (i.e., p should not set $gate[s] = (0, OPEN)$). If, however, the check succeeds, then p is still responsible for opening $gate[s]$. This is what p does in line 22.

Again, however, p cannot use a simple assignment statement to set $gate[s] = (0, OPEN)$ because of the danger of some other process requesting session $\overline{s}$ racing out of the NCS and into the GME waiting room. Instead, p makes use of the fact that it can safely assign $(0, OPEN)$ to $gate[s]$ as long as no one wrote to $gate[s]$ since the time p performed the recheck. Specifically, by using COMPARE_AND_SWAP, p atomically checks that its tag is still in $gate[s]$ (which it set before the recheck) and, if it is, it sets $gate[s] = (0, OPEN)$.

We consider the initial part of the exit protocol, which is responsible for checking the necessary conditions on the counter and performing the opening of the gate, as the "gate opening" phase of the exit protocol (lines 18..24). After a process completes the "gate opening" phase, the process finishes the exit protocol by executing one of MUTEXEXIT() or MUTEXABORT(), whichever is appropriate based on whether it entered the CS by finishing the method MUTEXWAITINGROOM() or CONFLICT-FREE().

We now return to the question why, in order to open the gate, a process leaving the CS needs to ensure that there is a process requesting the other session in the GME waiting room (see the second clause in the conditional of line 19). This check is necessary to ensure the nice RMR complexity property of our algorithm. To see this, consider the following scenario illustrating how the algorithm would operate if the condition were not present: one at a time, a succession of processes all requesting session $\overline{s}$ leave the NCS, execute conflict-free through the CS, and end up at line 20 of the exit protocol. The next process in this succession does not leave the NCS until the preceding one has reached line 20. Since there is no bound on the number of processes that can do this, the algorithm effectively allows processes requesting session $\overline{s}$ to "accumulate" at line 20. After this, a process requesting session s leaves the NCS concurrently with some process requesting session $\overline{s}$, and both end up busy-waiting in the CONFLICT-FREE() method. The processes requesting session $\overline{s}$ that have accumulated at line 20 then execute that line one at a time, each time invalidating $gate[s]$, causing the process requesting session s that is stuck in the GME waiting room to make $\omega(1)$ RMRs. This scenario is prevented

by ensuring that there is a process requesting session s in the GME waiting room when a process requesting session $\overline{s}$ leaves the CS; essentially, a process requesting session s in the GME waiting room acts as a "barricade" to processes requesting session $\overline{s}$, preventing them from racing through the CS and accumulating in the exit protocol. (This has to do with the FCFS property of the algorithm, the details of which are handled in the formal proof.)

6.1.4 Line by Line Commentary of the Algorithm

Line 3: Process p, requesting session s, increments the counter $active[s]$.

Line 4..8: Process p, requesting session s, attempts to set $gate[\overline{s}] = (0, CLOSED)$. The purpose of this is to ensure that any processes requesting session $\overline{s}$ that are still in the early stages of executing the trying protocol don't end up prematurely conflict-free qualified.

Line 9: Process p executes MUTEXDOORWAY() of the underlying abortable FCFS ME algorithm.

Line 10: Process p, requesting session s, atomically reads the value of $active[\overline{s}]$ and adds a value of $(N+1)$ to it. This has two purposes: the read is intended to check whether there are any active processes requesting session $\overline{s}$ that have yet to exit the CS, so that p knows whether to block on $gate[s]$ in the method CONFLICT-FREE(); and the addition of $(N+1)$ to $active[\overline{s}]$ is intended to inform any processes requesting session $\overline{s}$ that there is now a process requesting session s in the GME waiting room.

Line 12: One coroutine that p executes is MUTEXWAITINGROOM(), which is the waiting room of the underlying abortable FCFS ME algorithm.

Line 14: The other coroutine that p executes is CONFLICT-FREE(). The purpose of CONFLICT-FREE() is to allow process p, requesting session s, to enter the CS concurrently with other processes requesting session s when there are no processes requesting session $\overline{s}$.

Line 16: Process p, requesting session s, atomically subtracts $(N+1)$ from $active[\overline{s}]$. The purpose of this is to inform processes requesting session $\overline{s}$ that there is one less process requesting session s in the GME waiting room.

Line 17: Process p enters the CS.

Line 18: Process p, requesting session s, atomically reads the value of $active[s]$ and subtracts a value of 1 from it. This has several purposes. The read is intended to check whether p is the only process requesting session s active in lines 4..18, and also to check whether there are any processes requesting session $\overline{s}$ in the GME waiting room. If both of these conditions are true, then p may be able to set $gate[\overline{s}] = (0, OPEN)$, so that processes requesting session $\overline{s}$ may proceed into the CS. Also, the subtraction of 1 from $active[s]$ is intended to indicate to any processes requesting session $\overline{s}$ that have yet to enter the GME waiting room that there is one less process requesting session s that is active in lines 4..18.

Line 19: Process p, requesting session s, checks whether the following conditions hold when p executes the read at line 18: p is the only process requesting session s active in lines 4..18, and there is a process requesting session $\overline{s}$ that is in the GME waiting room.

Line 20: Process p sets $gate[\overline{s}] = (p, CLOSED)$. If anyone overwrites $gate[\overline{s}]$ before p gets a chance to execute the COMPARE_AND_SWAP at line 22, p's COMPARE_AND_SWAP at line 22 will fail.

Line 21: Process p, requesting session s, ensures that no other process requesting session s has become active between the time p executes line 18 and the time p executes this line. If some process requesting session s is active in lines 4..18, then p must not attempt to set $gate[\overline{s}] = (0, OPEN)$.

Line 22: Process p, requesting session s, attempts to set $gate[\overline{s}] = (0, OPEN)$, and will succeed provided that no process has overwritten $gate[\overline{s}]$ since p set it to $(p, CLOSED)$ at line 20.

Line 25..29: Process p executes one of MUTEXEXIT() or MUTEXABORT(), depending

on whether p is mutex qualified or conflict-free qualified.

6.1.5 Proof of Correctness

We start by proving some elementary lemmas that will be useful in proving that the algorithm in Figure 6.1 satisfies the stated properties.

The first lemma formalizes the intuition about the counter $active[s]$ that we provided in the informal description. It says that $active[s] = a(N+1) + b$, where $0 \leq b \leq N$, if and only if there are a processes requesting $\overline{s}$ in the GME waiting room, and there are b processes requesting s that have executed at least the first line of the trying protocol but have yet to start executing the exit protocol.

Lemma 6.1: $active[s] = a(N+1) + b$, where $0 \leq b \leq N$, iff there are a processes p that request session $\overline{s}$ such that $p@\{11..16\}$ and there are b processes q that request session s such that $q@\{4..18\}$.

Proof: The proof is by induction on the number of steps in the execution of the algorithm. (Recall that in the formal model an execution is an alternating sequence of global states and process identifiers, where, intuitively, the process identifier indicates which process takes a step at that particular point in the execution. Moreover, recall that every time a process takes a step, it involves executing an operation on zero or one shared variables.) Since $active[s] = 0$ initially, the statement is obviously true in the base case where zero steps have taken place. Let k be any positive integer. For the induction hypothesis, assume that the statement is true for any number of steps less than k. We now prove the statement true after exactly k steps have taken place. Assume that $active[s] = a(N+1) + b$, where $0 \leq b \leq N$, after the k^{th} step of the algorithm. Further assume that after the k^{th} step there are a' processes p that request session $\overline{s}$ such that $p@\{11..16\}$ and there are b' processes q that request session s such that $q@\{4..18\}$. (Clearly, since there are at most N processes in the system, $0 \leq b' \leq N$.) We now show that $a = a'$ and $b = b'$. By the Division Theorem, it suffices to show that

$active[s] = a'(N+1) + b'$ after the k^{th} step.

Either the k^{th} step involved writing $active[s]$ or it did not. Consider the latter case first. The only ways $active[s]$ can be written to is if a process requesting session s executes line 3 or line 18, or a process requesting session $\overline{s}$ executes line 10 or line 16. This, and our assumption that the k^{th} step did not involve writing $active[s]$, imply that the k^{th} step did not involve a process requesting session s executing line 3 or line 18, and it did not involve a process requesting session $\overline{s}$ executing line 10 or line 16. It follows that after the $k-1^{th}$ step (just before the k^{th} step), there are a' processes p that request session $\overline{s}$ such that $p@\{11..16\}$ and there are b' processes q that request session s such that $q@\{4..18\}$. By the induction hypothesis, $active[s] = a'(N+1) + b'$ after the $k-1^{th}$ step. Furthermore, by the assumption that $active[s]$ is not written in the k^{th} step, after the k^{th} step, $active[s] = a'(N+1) + b'$, as wanted.

Now consider the case where the k^{th} step did involve writing $active[s]$. The following are the possible operations that could have been executed during the k^{th} step: a process requesting session s incremented (decremented) $active[s]$ by one when it executed line 3 (line 18); or a process requesting session $\overline{s}$ incremented (decremented) $active[s]$ by $N+1$ when it executed line 10 (line 16). Consider each of these possibilities in turn:

Case 1: A process requesting session s incremented (decremented) $active[s]$ by one when it executed line 3 (line 18) during the k^{th} step.

The assumption that a process requesting session s executed line 3 (line 18) during the k^{th} step implies that just before the k^{th} step there was one less (additional) process requesting session s in lines 4..18. That is, after the $k-1^{th}$ step, there are a' processes p that request session $\overline{s}$ such that $p@\{11..16\}$, and there are $b'-1$ $(b'+1)$ processes q that request session s such that $q@\{4..18\}$. By the induction hypothesis, $active[s] = a'(N+1) + (b'-1)$ $(active[s] = a'(N+1) + (b'+1))$ after the $k-1^{th}$ step. Furthermore, by the assumption that $active[s]$ is incremented (decremented) by one in the k^{th} step, $active[s] = a'(N+1) + (b'-1) + 1 = a'(N+1) + b'$ $(active[s] = a'(N+1) + (b'+1) - 1 =$

$a'(N+1)+b'$) after the k^{th} step, as wanted.

Case 2: A process requesting session $\overline{s}$ incremented (decremented) $active[s]$ by $N+1$ when it executed line 10 (line 16) during the k^{th} step.

The assumption that a process requesting session $\overline{s}$ executed line 10 (line 16) during the k^{th} step implies that just before the k^{th} step there was one less (additional) process requesting session $\overline{s}$ in lines 11..16. That is, after the $k-1^{th}$ step, there are $a'-1$ ($a'+1$) processes p that request session $\overline{s}$ such that $p@\{11..16\}$, and there are b' processes q that request session s such that $q@\{4..18\}$. By the induction hypothesis, $active[s] = (a'-1)(N+1)+b'$ ($active[s] = (a'+1)(N+1)+b'$) after the $k-1^{th}$ step. Furthermore, by the assumption that $active[s]$ is incremented (decremented) by $N+1$ in the k^{th} step, $active[s] = (a'-1)(N+1)+b'+(N+1) = a'(N+1)+b'$ ($active[s] = (a'+1)(N+1)+b'-(N+1) = a'(N+1)+b'$) after the k^{th} step, as wanted. □

The next corollary says that a process p requesting session s will attempt to busy-wait in the method CONFLICT-FREE() if there is a process requesting session $\overline{s}$ that has executed at least one line of the trying protocol but not yet started the exit protocol.

Corollary 6.2: A process p requesting session s evaluates the condition at the start of the method CONFLICT-FREE() to be true iff at the time when p executes line 10 there is a process q requesting session $\overline{s}$ such that $q@\{4..18\}$.

Proof: Suppose a process p, requesting some session s, evaluates the condition at the start of the method CONFLICT-FREE() to be true. Hence $l\_active[\overline{s}]$ **mod** $(N+1) > 0$. Now, $l\_active[\overline{s}]$ is the value of $active[\overline{s}]$ p reads when it executes line 10. Hence $active[\overline{s}]$ **mod** $(N+1) > 0$ when p executes line 10. It then follows from Lemma 6.1 that there must be a process q, requesting session $\overline{s}$, such that $q@\{4..18\}$, when p executes line 10.

Conversely, suppose that when p executes line 10 there is a process q requesting session $\overline{s}$ such that $q@\{4..18\}$. So, by Lemma 6.1, $active[\overline{s}]$ **mod** $(N+1) > 0$ when p executes line 10. So p reads into $l\_active[\overline{s}]$ a value such that $l\_active[\overline{s}]$ **mod** $(N+1) > 0$. This

means that p will evaluate the condition at the start of the method CONFLICT-FREE() to be true. □

Intuitively, the corollary below states that a process p requesting session s will attempt to open $gate[\overline{s}]$ upon leaving the CS if and only if when p leaves the CS it is the last process requesting s to do so, and there is a process requesting session $\overline{s}$ that is in the GME waiting room.

Corollary 6.3: A process p requesting session s evaluates the condition at line 19 to be true iff at the time when p executes line 18 there is no process $q \neq p$ requesting session s such that $q@\{4..18\}$ and there is a process r requesting session $\overline{s}$ such that $r@\{11..16\}$.

Proof: Assume that a process p requesting session s evaluates the condition at line 19 to be true. Then $l\_active[s]$ **mod** $(N+1) = 1 \wedge l\_active[s] \geq (N+1)$. Now, $l\_active[s]$ is the value of $active[s]$ p reads when it executes line 18. So $active[s]$ **mod** $(N+1) = 1 \wedge active[s] \geq (N+1)$ is true when p starts executing line 18. So, by Lemma 6.1, when p executes line 18, there is a process r requesting session $\overline{s}$ such that $r@\{11..16\}$. Now, suppose, by way of contradiction, that there is a process $q \neq p$ requesting session s such that $q@\{4..18\}$ when p executes line 18. Then just before p executes line 18, there are at least two processes u such that $u@\{4..18\}$, namely p and q. Therefore, by Lemma 6.1, just before p executes line 18, $active[s]$ **mod** $(N+1) \geq 2$. Therefore the value of $l\_active[s]$ immediately after p executes line 18 is such that $l\_active[s]$ **mod** $(N+1) \geq 2$. Therefore p evaluates the condition at line 19 to be false, contradicting the assumption that p evaluates the condition at that line to be true.

Conversely, assume that when p executes line 18 there is no process $q \neq p$ requesting session s such that $q@\{4..18\}$ and there is a process r requesting session $\overline{s}$ such that $r@\{11..16\}$. Now, we consider $p@18$ to be true when p starts executing line 18, and so by Lemma 6.1, at the time p starts executing line 18, $active[s]$ **mod** $(N+1) = 1 \wedge active[s] \geq (N+1)$. Hence, p will read into $l\_active[s]$ at line 18 a value such that $l\_active[s]$ **mod** $(N+1) = 1 \wedge l\_active[s] \geq (N+1)$, and thus evaluate the condition at

line 19 to be true. □

The next lemma says that while a process requesting session $\overline{s}$ is enabled to execute lines 9..18, $gate[s]$ is not open, and so, as we will prove in a corollary to this lemma (Corollary 6.5), processes requesting session s cannot enter the CS conflict-free qualified.

Lemma 6.4: If there exists a process p requesting session $\overline{s}$, such that $p@\{9..18\}$ is invariant in some time interval $[t_a, t_b]$, then $gate[s].state = CLOSED$ is invariant in $[t_a, t_b]$.

Proof: Assume there exists a process p requesting session $\overline{s}$, such that $p@\{9..18\}$ is invariant in the time interval $[t_a, t_b]$. Suppose, by way of contradiction, that $gate[s].state = CLOSED$ is not invariant in $[t_a, t_b]$. Then there exists a time $t_4 \in [t_a, t_b]$ when $gate[s] = (0, OPEN)$. But $gate[s] = (0, CLOSED)$ initially. Hence it must be the case there exists a process r, requesting session $\overline{s}$, that sets $gate[s] = (0, OPEN)$ at some time $t_3 \leq t_4$ at line 22. Due to the atomicity of the COMPARE_AND_SWAP at line 22, if there is more than one process that sets $gate[s] = (0, OPEN)$ prior to t_4, then the times when such processes execute line 22 can be totally ordered. So, without loss of generality, assume t_3 is the last time prior to t_4 that some process sets $gate[s] = (0, OPEN)$.

Let t_2 be the last time when r executes line 21 prior to t_3.

Claim 6.4.1: Process p executes line 3 after time t_2.

Proof: Since r executes line 22 at time t_3, r must have evaluated the condition at line 21 to be true at time t_2. Thus Lemma 6.1 implies that either p executes line 3 after time t_2, or p executes line 18 before t_2. The second alternative is not possible:

By assumption $p@\{9..18\}$ is invariant during $[t_a, t_b]$. So, if p executes line 18 before t_2, we would have $t_b < t_2$. By definition, $t_2 < t_3 \leq t_4$. Thus $t_b < t_4$, which contradicts the fact that $t_4 \in [t_a, t_b]$. Thus the first alternative, that p executes line 3 after time t_2, must hold. □ (Claim 6.4.1)

Let t_1 be the last time when r executes line 20 prior to t_2.

Claim 6.4.2: No process requesting session $\overline{s}$ executes line 20 in the interval (t_1, t_3).

Proof: Suppose, by way of contradiction, that the claim is false. It follows that

when r executes the COMPARE_AND_SWAP at line 22 at time t_3, it will fail. This contradicts the assumption that r's execution of COMPARE_AND_SWAP at time t_3 succeeds. □ (Claim 6.4.2)

Claim 6.4.3: No process requesting session $\overline{s}$ that executes line 3 after time t_2 will execute past line 5 in the interval (t_2, t_3).

Proof: Suppose, by way of contradiction, that the claim is false. Let S be the set of processes requesting session $\overline{s}$ that execute line 3 after t_2 and execute at least one line after line 5 in (t_2, t_3). By the fact that r sets $gate[s] = (r, CLOSED)$ at time t_1, and Claim 6.4.2, it must be the case that at line 4 each process in S reads into its copy of l_gate a value such that $l\_gate = (r, CLOSED)$. Thus all processes in S will evaluate the condition at line 5 to be true. Since the COMPARE_AND_SWAP at line 6 is atomic, we can totally order the processes in S by the time when they execute line 6. Hence there must be some process $u \in S$ that is the first to execute line 6 at some time $t_u \in (t_2, t_3)$. By Claim 6.4.2, and the fact that u is the first process in S to execute line 6, $gate[s] = (r, CLOSED)$ is still true at time t_u. This, and the fact that $u.l\_gate = (r, CLOSED)$, imply that u's COMPARE_AND_SWAP at line 6 at time t_u will be successful. But this implies that r's COMPARE_AND_SWAP at t_3 will fail, contradicting the assumption that it succeeds. □ (Claim 6.4.3)

By Claim 6.4.1, p executes line 3 after t_2. It then follows from Claim 6.4.3 that p cannot execute past line 5 in the interval (t_2, t_3). Hence p will execute line 8 at some time $t \in (t_3, t_a)$. The only reason the COMPARE_AND_SWAP at this line can fail is if some other process sets $gate[s] \neq (0, OPEN)$ at some time $t' \in (t_3, t)$. By inspection of the algorithm, every place where a process can set $gate[s] \neq (0, OPEN)$ has the side effect of setting $gate[s].state = CLOSED$. Hence, it must be the case that either some process other than p sets $gate[s].state = CLOSED$ at $t' \in (t_3, t)$, or p's execution of COMPARE_AND_SWAP at line 8 succeeds and p sets $gate[s] = (0, CLOSED)$ at time $t \in (t_3, t_a)$. But, by assumption, $gate[s] = (0, OPEN)$ at time $t_4 \in [t_a, t_b]$. Hence some process must set

$gate[s] = (0, \mathit{OPEN})$ at some time after t_3, but before t_4. This contradicts the fact that t_3 is the last time prior to t_4 that a process sets $gate[s] = (0, \mathit{OPEN})$. □

Corollary 6.5: Suppose a process q requests session s and executes line 10 at time t_a. Let t_b be a time such that $t_b \geq t_a$. If there exists a process p, requesting session $\overline{s}$, such that $p@\{9..18\}$ is invariant in the interval $[t_a, t_b]$, then q cannot be conflict-free qualified in $[t_a, t_b]$.

Proof: Assume process p requests session $\overline{s}$ and $p@\{9..18\}$ is invariant in the interval $[t_a, t_b]$. Now suppose, by way of contradiction, that q is conflict-free qualified in the interval $[t_a, t_b]$. Thus q finishes executing the method CONFLICT-FREE() by time t_b. Since, by assumption, $p@\{9..18\}$ is invariant in $[t_a, t_b]$, Corollary 6.2 implies that q evaluates the condition at the start of the method CONFLICT-FREE() to be true. Hence q busy-waits in CONFLICT-FREE() until $gate[s] = (0, \mathit{OPEN})$. By Lemma 6.4, however, $gate[s].state = \mathit{CLOSED}$ is invariant in $[t_a, t_b]$, and so q cannot finish executing CONFLICT-FREE() by time t_b, which contradicts the fact that it does. □

Mutual Exclusion

Theorem 6.6: The algorithm in Figure 6.1 satisfies the mutual exclusion property.

Proof: Let p be a process that requests session $\overline{s}$ and q be a process that requests session s. Suppose, by way of contradiction, that p and q are in the CS simultaneously at time t. Since at most one process in the CS is mutex qualified, without loss of generality, assume q is conflict-free qualified.

Let time t_q be the time when q executes line 10.

Claim 6.6.1: Process p executes line 8 of its current invocation after time t_q.

Proof: Suppose, by way of contradiction, that the claim is false. Thus $p@\{9..18\}$ is invariant in the interval $[t_q, t]$. From this and Corollary 6.5 it follows that q cannot be conflict-free qualified in $[t_q, t]$, which contradicts the fact that q is in the CS conflict-free qualified at time t. □ (Claim 6.6.1)

Either p is mutex qualified or conflict-free qualified at time t. We consider each of these cases in turn:

Case 1: Process p is mutex qualified.

By Claim 6.6.1, p executes line 8 after q executes line 10, and so p starts executing the method MUTEXDOORWAY() at line 9 after q finishes executing the method MUTEXDOORWAY() at line 9. Hence, due to the FCFS property of $\mathbb{A}$, p cannot be mutex qualified until q executes the method MUTEXABORT() at line 28, contradicting the assumption that p and q are in the CS simultaneously at time t.

Case 2: Process p is conflict-free qualified.

By Claim 6.6.1, p executes line 8 after q executes line 10, and so p executes line 10 after q does. Since p and q are both conflict-free qualified, symmetrically, q executes line 10 after p does. This is a contradiction.

Both cases lead to a contradiction, and hence mutual exclusion holds. □

FCFS

Theorem 6.7: The algorithm in Figure 6.1 satisfies the FCFS property.

Proof: Assume p requests session $\overline{s}$, q requests session s, and p doorway-precedes q. We must prove that q does not enter the CS before p enters the CS. Suppose, by way of contradiction, that it does. There are two cases, depending on how q enters the CS.

Case 1: Process q enters the CS mutex qualified before p.

Since p doorway-precedes q, p finishes executing the method MUTEXDOORWAY() at line 9 before q begins executing the method MUTEXDOORWAY() at line 9. Hence, by the FCFS property of $\mathbb{A}$, q cannot become mutex qualified until p executes one of MUTEXEXIT() or MUTEXABORT(). This contradicts the assumption that q enters the CS mutex qualified before p.

Case 2: Process q enters the CS conflict-free qualified before p.

Let t_q be the time when q executes line 10, and let t_p be the time when p enters

the CS. Since p doorway-precedes q, $p@\{9..18\}$ is invariant in the interval $[t_q, t_p]$. By Corollary 6.5, q cannot be conflict-free qualified in the interval $[t_q, t_p]$. This implies that q cannot enter the CS conflict-free qualified before p, which contradicts the assumption that it does. □

Lockout Freedom

Theorem 6.8: The algorithm in Figure 6.1 satisfies the lockout freedom property.

Proof: Suppose, by way of contradiction, that the algorithm does not satisfy the lockout freedom property. This means that there is an execution in which some process gets stuck forever in the GME waiting room when it tries to enter the CS. The following claim proves this is impossible.

Claim 6.8.1: No process can be stuck forever in the method MUTEXWAITINGROOM().

Proof: $\mathbb{A}$ is an abortable mutual exclusion algorithm that satisfies the lockout freedom property. Hence the only way for a process to get stuck forever in MUTEXWAITINGROOM() is if some other process becomes mutex qualified and then does not execute the method MUTEXEXIT(); or, if some process starts executing MUTEXWAITINGROOM(), aborts execution of it, but then does not execute the method MUTEXABORT(). If a process becomes mutex qualified, then we see by examining the algorithm it will eventually execute the method MUTEXEXIT(). Furthermore, the only way a process will abort execution of MUTEXWAITINGROOM() once started is if the process becomes conflict-free qualified. In this case, again by examining the algorithm, we see that such a process will eventually execute the method MUTEXABORT(). Hence no process can be stuck forever in the method MUTEXWAITINGROOM(). □ (Claim 6.8.1)

Claim 6.8.1 contradicts the fact that there is an execution in which some process is stuck forever in the GME waiting room. □

Concurrent Entering

Theorem 6.9: The algorithm in Figure 6.1 satisfies the concurrent entering property.

Proof: Let p be a process requesting session s that is in the GME waiting room at time t_a, and let t_b be the time when p enters the CS. Assume there are no active processes requesting session $\overline{s}$ in $[t_a, t_b]$. We want to show that p enters the CS in a bounded number of its own steps in $[t_a, t_b]$.

Now, either p evaluates the condition at the beginning of CONFLICT-FREE() to be true prior to t_b, or it does not. In the latter case, it is clear that p will complete the method CONFLICT-FREE() and will enter the CS in a bounded number of its own steps.

So assume that p evaluates the condition at the beginning of CONFLICT-FREE() to be true at some time $t' < t_b$. Thus p will busy-wait at line 2 of the method CONFLICT-FREE(). For p to enter the CS in a bounded number of its own steps in $[t_a, t_b]$, it suffices that $gate[s] = (0, OPEN)$ is invariant in $[t_a, t_b]$. The rest of the proof shows this.

From the fact that p evaluates the condition at the beginning of CONFLICT-FREE() to be true, and Corollary 6.2, it follows there is a process q requesting session $\overline{s}$ such that $q@\{4..18\}$ when p executes line 10 at some time $t < t'$.

Let $S = \{r |$ process r requests session $\overline{s} \wedge r@\{4..18\}$ at time t $\}$. Clearly $q \in S$, and so S is non-empty. Since no process in S can be active in $[t_a, t_b]$, and the fact that processes execute line 18 atomically, it follows that some process $v \in S$ must be the last to execute line 18 at some time $t_v \in (t, t_a)$. Clearly $p@\{11..16\}$ at time t_v.

Claim 6.9.1: There is no process u requesting session $\overline{s}$ such that $u@\{4..18\}$ at any point in the interval $(t_v, t_b]$.

Proof: Suppose, by way of contradiction, that there is a process u requesting session $\overline{s}$ such that $u@\{4..18\}$ at some time in $(t_v, t_b]$. Now, v is the last process in S to execute line 18 (at time t_v), and so $u \notin S$. Hence u must execute line 3 at some point in the interval $(t, t_b]$. This implies that p doorway-precedes u, which, in turn, implies, by the

FCFS property of the 2-session algorithm, that p must enter the CS before u. Thus u must be active during $[t_a, t_b]$, which contradicts that no process requesting session $\overline{s}$ is active during $[t_a, t_b]$. □ (Claim 6.9.1)

Claim 6.9.2: There is no process $u \neq v$ requesting session $\overline{s}$ such that $u@\{4..18\}$ at time t_v.

Proof: Suppose, by way of contradiction, that there is a process $u \neq v$ requesting session $\overline{s}$ such that $u@\{4..18\}$ at time t_v. Since at time t_v process v executes an atomic operation at line 18, it must be the case that immediately after time t_v, $u@\{4..18\}$ is still true. Hence $u@\{4..18\}$ at a point in the interval $(t_v, t_b]$, which contradicts Claim 6.9.1. □ (Claim 6.9.2)

As noted before Claim 6.9.1, $p@\{11..16\}$ at time t_v. This, Claim 6.9.2, and Corollary 6.3, imply that v will evaluate the condition at line 19 to be true. In other words, v will attempt to open $gate[s]$ before it returns to the NCS.

Let $t'_v > t_v$ be the time when v executes line 20. By the atomicity of the write that takes place at line 20, some process z requesting session $\overline{s}$ (possibly $z = v$) is the last to execute line 20 in $[t'_v, t_a)$. It immediately follows from Claim 6.9.1 that z will evaluate the condition at line 21 to be true, and will successfully execute the COMPARE_AND_SWAP at line 22. Hence $gate[s] = (0, \mathit{OPEN})$ at some time $t_z \in (t'_v, t_a)$. The only way $gate[s] \neq (0, \mathit{OPEN})$ at any point in $[t_a, t_b]$ is if some process u requesting session $\overline{s}$ updates $gate[s]$ by executing line 6, line 8, or line 20 in the interval (t_z, t_a). However, by assumption, z is the last process requesting session $\overline{s}$ to execute line 20 prior to t_a, and Claim 6.9.1 implies that $\neg u@6$ and $\neg u@8$ in (t_z, t_a). Hence u cannot execute line 6, line 8, or line 20 in (t_z, t_a). It follows that $gate[s] = (0, \mathit{OPEN})$ in $[t_a, t_b]$, as required. □

Bounded Exit

Theorem 6.10: The algorithm in Figure 6.1 satisfies the bounded exit property.

Proof: Since the underlying FCFS abortable ME algorithm satisfies the bounded exit property, it is evident by inspection of the algorithm that there are no unbounded loops in the exit protocol. Hence the 2-session algorithm satisfies the bounded exit property. □

6.1.6 RMR Complexity

We first broadly sketch the proof that the resulting 2-session GME algorithm has RMR complexity $O(f(N))$, and then proceed with the details.

Lemma 6.12 shows that at most two processes requesting session $\overline{s}$ can update $gate[s]$ in lines 20..22 while a process requesting session s is blocked on $gate[s]$ in the GME waiting room. From this it follows (Lemma 6.13) that only a constant number of successful COMPARE_AND_SWAPs can take place at line 6 or line 8 by processes requesting session $\overline{s}$ while a process is blocked on $gate[s]$ in the GME waiting room. Together these two lemmas imply that while a process is blocked on $gate[s]$ its local copy of that variable can be invalidated only a constant number of times. From this and the fact that there are no other loops in the reduction, it follows that the reduction has $O(1)$ RMR complexity, and so the resulting 2-session GME algorithm has $O(f(N))$ RMR complexity, under the CC model.

The first Lemma we prove here is rather technical and is required in the proof of Lemma 6.12. This lemma essentially says that if a process q requesting session $\overline{s}$ finds that it is responsible for opening $gate[s]$, then processes requesting session s that are currently in the GME waiting room cannot leave it until q has finished the "gate opening" phase of the exit protocol.

Lemma 6.11: If a process q requesting a session $\overline{s}$ executes line 18 at some time t and subsequently evaluates the condition at line 19 to be true, then any process requesting session s that is in the GME waiting room at time t cannot leave the GME waiting room before q executes past line 22.

Proof: Suppose, by way of contradiction, that the lemma is false. Thus, there is a run α where the precondition of the lemma holds but the consequent is violated.

By the atomicity of operations, there exists a total ordering of the times in α at which some process requesting a session $\overline{s}$ executes line 18 and subsequently evaluates the condition at line 19 to be true. Let $t_1 < t_2 < t_3 < \ldots$ be this ordering, and let k be the smallest positive integer such that there is a process requesting a session s in the GME waiting room at time t_k that leaves the GME waiting room before the process requesting session $\overline{s}$ that executes line 18 at time t_k executes past line 22. Integer k is well-defined because of the assumption that the lemma is false. Let q be the process requesting session $\overline{s}$ that executes line 18 at time t_k, and let p be a process requesting session s that is in the GME waiting room at time t_k and leaves the GME waiting room before q executes past line 22.

For p to leave the GME waiting room it must be mutex qualified or conflict-free qualified. Likewise, when q executes through the CS, it must do so as a mutex qualified or conflict-free qualified process.

We prove some claims and then proceed by considering the different ways q and p execute through the CS.

Claim 6.11.1: If q executes through the CS mutex qualified, then q starts execution of MUTEXDOORWAY() at line 9 before p finishes execution of MUTEXDOORWAY() at line 9.

Proof: Assume that q executes through the CS mutex qualified, and suppose, by way of contradiction, that q starts execution of MUTEXDOORWAY() at line 9 after p finishes execution of MUTEXDOORWAY() at line 9. Then, by the FCFS property of $\mathbb{A}$, q cannot be mutex qualified before p executes one of MUTEXEXIT() or MUTEXABORT(). This contradicts that q executes line 18 while p is in the GME waiting room. □ (Claim 6.11.1)

Claim 6.11.2: If q executes through the CS conflict-free qualified, then q executes line 10 before p executes line 8.

Proof: Assume q executes through the CS conflict-free qualified, and let time t'_k be the time when q executes line 10. Now, suppose by way of contradiction that q executes line 10 after p executes line 8. From this and the fact p is in the GME waiting room at time t_k, it follows that $p@\{9..16\}$ is invariant during $[t'_k, t_k]$. By Corollary 6.5, q cannot be conflict-free qualified in the interval $[t'_k, t_k]$, which implies that q cannot execute through the CS in the interval $[t'_k, t_k]$. This, in turn, implies that q cannot execute line 18 at time t_k, which contradicts the assumption that it does. □ (Claim 6.11.2)

Claim 6.11.3: Process q executes line 8 before p executes line 10.

Proof: Follows immediately from Claim 6.11.1 and Claim 6.11.2. □ (Claim 6.11.3)

We now consider the different ways in which q and p execute through the CS, in each case reaching a contradiction.

Case 1: Both p and q execute through the CS mutex qualified.

By assumption, p is in the GME waiting room when q is mutex qualified and executes line 18. Hence, by the mutual exclusion property of $\mathbb{A}$, q must execute MUTEXEXIT() before p can leave the GME waiting room mutex qualified. From this and the fact that q must execute past line 22 before it can execute MUTEXEXIT() it follows that q must execute past line 22 before p can leave the GME waiting room. This contradicts the assumption that p leaves the GME waiting room mutex qualified before q executes past line 22.

Case 2: Process p executes through the CS mutex qualified and q executes through the CS conflict-free qualified.

Claim 6.11.2 implies that q finishes execution of MUTEXDOORWAY() at line 9 before p begins execution of MUTEXDOORWAY() at line 9. Hence, by the FCFS property of $\mathbb{A}$, p is mutex qualified after q executes MUTEXABORT(). From this and the fact that q must execute past line 22 before it executes MUTEXABORT(), q must execute past line 22 before p is mutex qualified. This contradicts the assumption that p leaves the GME waiting room mutex qualified before q executes past line 22.

Case 3: Process p executes through the CS conflict-free qualified.

Claim 6.11.3 and the assumption that p is in the GME waiting room when q executes line 18 imply $q@\{9..18\}$ when p executes line 10. From this and Corollary 6.2, it follows p will evaluate the condition at the start of CONFLICT-FREE() to be true and will hence block on $gate[s]$ until it equals $(0, \mathit{OPEN})$.

Since $gate[s] = (0, \mathit{CLOSED})$ initially, there must be some process r requesting session $\overline{s}$ that sets $gate[s] = (0, \mathit{OPEN})$ at line 22 before p leaves the GME waiting room conflict-free qualified. Due to the atomicity of the operation at line 22, a total order can be placed on the times when processes requesting session $\overline{s}$ execute this line. So, without loss of generality, we assume that r is the process that last executes line 22 successfully prior to p leaving the GME waiting room. Now, r must evaluate the condition at line 19 to be true prior to executing line 22. Let t_r be the time when r executes line 18. By Corollary 6.3, at time t_r there is no process $z \neq r$ requesting session $\overline{s}$ such that $z@\{4..18\}$ and there is a process u requesting session s such that $u@\{11..16\}$ (i.e., u is in the GME waiting room).

Thus either q executes line 3 after t_r or q executes line 18 before t_r.

Case 3.1: Process q executes line 18 before t_r.

Process r executes line 18 after q executes line 18, and so, again by Corollary 6.3, r executes line 3 after q executes line 18. (Recall that q, like r, evaluates the condition at line 19 to be true.) By assumption, p is in the GME waiting room when q executes line 18. Hence p is in the GME waiting room when r executes line 3. This implies that p doorway-precedes r. Thus by the FCFS property of the 2-session algorithm, p must enter the CS before r, and so r executes line 22 after p has left the GME waiting room. This contradicts the fact that r sets $gate[s] = (0, \mathit{OPEN})$ at line 22 before p leaves the GME waiting room.

Case 3.2: Process q executes line 3 after t_r.

Recall that we defined k as the smallest positive integer such that there is a process

requesting a session s in the GME waiting room at time t_k that leaves the GME waiting room before the process requesting session $\overline{s}$ that executes line 18 at time t_k executes past line 22.

Either $k = 1$ or $k > 1$.

Consider the case where $k = 1$. By assumption q executes line 3 after r executes line 18, and so q executes line 18 after r executes line 18. This implies that $t_k > t_r$. However, also by the assumption that $k = 1$, t_k is the first time in α some process requesting session $\overline{s}$ executes line 18 and subsequently evaluates the condition at line 19 to be true. This implies that $t_k < t_r$, which contradicts that $t_k > t_r$.

Now consider the case where $k > 1$.

Claim 6.11.4: Process r must execute line 22 before q executes line 18.

Proof: Recall that there is a process u requesting session s such that $u@\{11..16\}$ (i.e., u is in the GME waiting room) at time t_r. Since q executes line 3 after t_r, by assumption, it follows that u doorway-precedes q, and so by the FCFS property of the 2-session algorithm, u must enter the CS before q. Also, because q executes line 3 after t_r, q must execute line 18 after t_r. That is, $t_k > t_r$, and so, by the minimality of k, r must execute line 22 before u leaves the GME waiting room. This implies that r must execute line 22 before q enters the CS, which, in turn, implies that r must execute line 22 before q executes line 18. □ (Claim 6.11.4)

Claim 6.11.5: Process r must execute line 22 at the same time or after q executes line 18.

Let t''_k be the time when q executes line 8. Clearly $q@\{9..18\}$ in the interval (t''_k, t_k), and so by Lemma 6.4, $gate[s].state = CLOSED$ in (t''_k, t_k). Now, it follows from Claim 6.11.3 that p enters the GME waiting room after t''_k. From this and the fact that p must read $gate[s] = (0, OPEN)$ in order to leave the GME waiting room, p must read $gate[s] = (0, OPEN)$ at some time after t''_k. Since $gate[s].state = CLOSED$ is invariant in (t''_k, t_k), the earliest p can read $gate[s] = (0, OPEN)$ is at time t_k. This means that there

is a process that sets $gate[s] = (0, OPEN)$ at time t_k or after. Since, by assumption, r is the last process to set $gate[s] = (0, OPEN)$ prior to p leaving the GME waiting room, r must execute line 22 at time t_k or after. That is, r must execute line 22 at the same time or after q executes line 18. □ (Claim 6.11.5)

Claim 6.11.4 and Claim 6.11.5 contradict each other. It follows that the Lemma holds. □

The next lemma that we prove is at the heart of the RMR complexity result. It says that while a process requesting session s is in the GME waiting room, at most two processes requesting session $\overline{s}$ can attempt to open $gate[s]$.

Lemma 6.12: If p is a process requesting session s and it enters the GME waiting room at some time t_a and leaves at some time t_b, then there are at most two processes requesting session $\overline{s}$ that are ever enabled to execute lines 20..22 during the interval (t_a, t_b).

Proof: Assume p is a process requesting session s and enters the GME waiting room at a time t_a, and leaves at a time t_b. Now, suppose by way of contradiction that there are three processes, q_1, q_2, and q_3, requesting session $\overline{s}$ enabled to execute lines 20..22 at some times in the interval (t_a, t_b). The atomicity of the operation at line 18 allows us to totally order the times when q_1, q_2, and q_3 execute line 18. So assume without loss of generality that q_1 executes line 18 first, q_2 executes line 18 second, and q_3 executes line 18 third, at some times t_1, t_2, and t_3, respectively. Since q_1, q_2, and q_3 are enabled to execute lines 20..22 in (t_a, t_b), it must be the case that q_1, q_2, and q_3 each evaluate the condition at line 19 to be true prior to t_b. From this and Corollary 6.3 it follows that when each $q_i \in \{q_1, q_2, q_3\}$ executes line 18, there is no process $q \neq q_i$ requesting session $\overline{s}$ such that $q@\{4..18\}$. This, and the ordering we placed on q_1, q_2, and q_3, imply that q_2 executes line 3 after t_1, and q_3 executes line 3 after t_2.

We proceed by considering whether p is in the GME waiting room at time t_2, reaching a contradiction in each case.

Case 1: Process p is in the GME waiting room at time t_2.

By the hypothesis of this case and the fact that q_3 executes line 3 after t_2, p doorway-precedes q_3. Hence, by the FCFS property of the 2-session algorithm, p must enter the CS before q_3. This implies that p leaves the GME waiting room before q_3 is enabled to execute lines 20..22. But this contradicts the assumption p is in the GME waiting room when q_3 is enabled to execute lines 20..22.

Case 2: Process p is not in the GME waiting room at time t_2.

Claim 6.12.1: Process p enters the GME waiting room after q_2 executes line 18.

Proof: Suppose, by way of contradiction, that p enters the GME waiting room before q_2 executes line 18 (i.e., before time t_2). By the hypothesis of this case, p must also leave the GME waiting room before q_2 executes line 18 at time t_2. This implies that p will not be in the GME waiting room when q_2 is enabled to execute lines 20..22, which contradicts the assumption p is in the GME waiting room when q_2 is enabled to execute lines 20..22. □ (Claim 6.12.1)

By Corollary 6.3 there is some process r requesting session s that is in the GME waiting room at time t_1. Since q_2 executes line 3 after t_1, it follows that r doorway-precedes q_2. Now, by Lemma 6.11, we know that q_1 must execute past line 22 before r enters the CS; and by the FCFS property of the algorithm, r must enter the CS before q_2. Hence q_1 must execute past line 22 before q_2 enters the CS, and so q_1 must execute past line 22 before q_2 executes line 18. From this and Claim 6.12.1, it follows that p enters the GME waiting room after q_1 executes past line 22. This contradicts the assumption p is in the GME waiting room when q_1 is enabled to execute lines 20..22.

Each case leads to a contradiction, so the Lemma holds. □

Lemma 6.13: At most five successful COMPARE_AND_SWAPs in lines 6 or 8 are executed by processes requesting session $\overline{s}$ while a process requesting session s is in the GME waiting room.

Proof: Suppose, by way of contradiction, that there are six successful COMPARE_AND_SWAPs in line 6 or 8 by processes requesting session $\overline{s}$ while a process

p requesting session s is in the GME waiting room. Since $gate[s] = (0, CLOSED)$ after a successful COMPARE_AND_SWAP at line 6 or line 8, and for a COMPARE_AND_SWAP at line 6 or 8 to be successful it must be that $gate[s] \neq (0, CLOSED)$, it follows that $gate[s]$ is assigned a value not equal to $(0, CLOSED)$ in-between any two of the successful COMPARE_AND_SWAPs in line 6 or 8. The only places where such a value can be assigned in the algorithm are at lines 20 and 22. Therefore between the first and last of the six successful COMPARE_AND_SWAPs in line 6 or 8, there are at least five executions of lines 20 and 22. Thus, while p is in the GME waiting room, there are five executions of lines 20 and 22. However, by Lemma 6.12, while p is in the GME waiting room there are at most two processes requesting session $\overline{s}$ enabled to execute lines 20..22 and thus at most four executions of lines 20 and 22. Thus we have reached a contradiction. □

Theorem 6.14: The reduction presented in Figure 6.1 has RMR complexity $O(1)$ (i.e., the number of remote memory references outside of $\mathbb{A}$ is constant).

Proof: The only loop in the reduction is the busy-wait loop in the method CONFLICT-FREE(). Outside of this method, it is clear that there are only a constant number of remote memory references. Now, in the CC model, a process busy-waiting on some variable v makes a remote memory reference only when some other process invalidates v by writing a new value to it. Hence, to show that the reduction has $O(1)$ RMR complexity, it suffices to show that $gate[s]$ can be invalidated only a constant number of times while some process p requesting s is blocked on $gate[s]$ in CONFLICT-FREE() (i.e., while p is in the waiting room).

The only places in the algorithm where $gate[s]$ can be written to is at lines 6, 8, 20, and 22 (by a process requesting session $\overline{s}$). By Lemma 6.13, there are at most 5 successful COMPARE_AND_SWAPs in lines 6 and 8 that are executed by processes requesting session $\overline{s}$ while a process requesting s is in the GME waiting room. (We assume failed COMPARE_AND_SWAPs do not invalidate variables, as we described at the beginning of this chapter.) Furthermore, by Lemma 6.12 there are at most 2 processes requesting

session $\overline{s}$ that are ever enabled to execute lines 20..22 while a process requesting session s is in the GME waiting room. This implies that there are most 4 times $gate[s]$ can be invalidated at lines 20 and 22 while a process requesting session s is in the GME waiting room. Hence, in total, there are at most 9 times that $gate[s]$ can be invalidated while some process requesting session s is in the waiting room. □

Theorem 6.15: The 2-session GME algorithm in Figure 6.1 has RMR complexity $O(f(N))$, where $O(f(N))$ is the RMR complexity of $\mathbb{A}$.

Proof: Follows immediately from Theorem 6.14. □

6.2 M-Session GME With Low RMR Complexity

Figure 6.3 M-Session GME Algorithm; Process $p \in \{1..N\}$

```
private variables:
    mysession: integer /* Session that p wants to attend; set before p leaves the NCS */
    mysession_2S: {1, 2} /* Version of mysession for the 2-session GME algorithm */
    M: integer /* The number of sessions this GME algorithm can handle */
 1: loop
 2:   NCS
 3:   if M > 1 then
 4:     if mysession ≤ ⌊M/2⌋ then
 5:       TRYING_LEFT(mysession);
 6:       mysession_2S := 1;
 7:     else
 8:       TRYING_RIGHT(mysession − ⌊M/2⌋);
 9:       mysession_2S := 2;
10:     end if
11:     TRYING_2S(mysession_2S);
12:   end if
13:   CS
14:   if M > 1 then
15:     EXIT_2S(mysession_2S);
16:     if mysession_2S = 1 then
17:       EXIT_LEFT(mysession);
18:     else
19:       EXIT_RIGHT(mysession − ⌊M/2⌋);
20:     end if
21:   end if
22: end loop
```

6.2.1 Introduction

In this section we present an M-session, N-process GME algorithm with RMR complexity $O\big(\log M \cdot f(N)\big)$ under the CC model. The algorithm is constructed recursively on M. It uses as subroutines three N-process GME algorithms: an $\lfloor M/2 \rfloor$-session, which we call the "left" algorithm and denote $\mathbb{A}_L$; an $\lceil M/2 \rceil$-session, which we call the "right" algorithm and denote $\mathbb{A}_R$; and a 2-session algorithm, which we will denote $\mathbb{A}_2$. $\mathbb{A}_L$ and $\mathbb{A}_R$ are obtained recursively; $\mathbb{A}_2$ is the algorithm we presented in Section 6.1.

The base case of the recursion, when $M = 1$, is trivial: Since there is only one session (and therefore, no possibility of conflict) each process can enter the CS directly, without any coordination: That is, the trying and exit protocols are the empty programs! For $M > 1$, the M-session algorithm works as follows. A process p requesting session $s \in [1..M]$ invokes the trying protocol of $\mathbb{A}_L$ requesting session s, if s is in the first ("left") half of the range $[1..M]$, i.e., if $s \in [1..\lfloor M/2 \rfloor]$; and it invokes the trying protocol of $\mathbb{A}_R$ requesting session $s - \lfloor M/2 \rfloor$, if s is in the second ("right") half of $[1..M]$, i.e., if $s \in [\lceil M/2 \rceil..M]$. Because of the mutual exclusion property of $\mathbb{A}_L$ and $\mathbb{A}_R$, all processes that have completed the trying protocol of $\mathbb{A}_L$ must be requesting the same session, say s_L; and all processes that have completed the trying protocol of $\mathbb{A}_R$ must be requesting the same session, say s_R. To ensure mutual exclusion of the M-session algorithm, we must now allow only the group of processes requesting s_L or only the group of processes requesting s_R to gain access to the CS. To achieve this, these two groups of processes compete for access to the CS by executing the trying protocol of the 2-session GME algorithm $\mathbb{A}_2$. Processes requesting s_L — i.e., those that completed the trying protocol of $\mathbb{A}_L$ — try to enter the CS by first executing the trying protocol of $\mathbb{A}_2$, requesting session 1. Similarly, processes requesting s_R try to enter the CS by first executing the trying protocol of $\mathbb{A}_2$, requesting session 2. After completing the CS, a process executes the exit protocols of the algorithms it used in its entry protocol, in reverse order: First the exit protocol of $\mathbb{A}_2$ and then the exit protocol of $\mathbb{A}_L$ or $\mathbb{A}_R$ (whichever of the two it

used in its entry protocol).

The algorithm is shown in pseudocode in Figure 6.3, where we use the following notational conventions. The trying protocol of $\mathbb{A}_L$, $\mathbb{A}_R$ and $\mathbb{A}_2$ are denoted TRYING_LEFT, TRYING_RIGHT and TRYING_2S, respectively. Similarly for the exit protocols. In our GME algorithms we have always assumed that before the process leaves the NCS, a private variable *mysession* is set to the session that the process is requesting in its attempt to enter the CS. In the algorithm informally described above, when we use each of $\mathbb{A}_L$, $\mathbb{A}_R$ and $\mathbb{A}_2$ we want to be able to set the private *mysession* variable used by *that* algorithm appropriately. We indicate the desired value of *mysession* by "passing" it as a parameter to the trying or exit protocol of the relevant algorithm. For example, the notation TRYING_2S(2) means: execute the trying protocol of $\mathbb{A}_2$, having set the private *mysession* variable of that algorithm to 2.

We proceed as follows: we give some definitions relative to the algorithm, present a proof of correctness, and then analyze the RMR complexity of the algorithm.

6.2.2 Definitions

GME Trying Protocol: The trying protocol consists of lines 3..12

GME Waiting Room: The waiting room is the same as the trying protocol.

GME Exit Protocol: The exit protocol consists of lines 14..21

6.2.3 Proof of Correctness

Let $\mathbb{A}_M$ be the M-session GME algorithm depicted in Figure 6.3, and let $\mathbb{A}_2$ be the 2-session algorithm in Figure 6.1 (i.e., the algorithm whose trying and exit protocols are defined by TRYING_2S and EXIT_2S).

We now prove that $\mathbb{A}_M$ satisfies each of the required GME properties.

Theorem 6.16: $\mathbb{A}_M$ satisfies the following properties: mutual exclusion, lockout freedom, concurrent entering, and bounded exit.

Proof: The proof is by induction on M, the number of sessions the algorithm handles. In the base case ($M = 1$) processes are immediately admitted to the CS and immediately return to the NCS after leaving the CS. Thus the algorithm satisfies lockout freedom, concurrent entering, and bounded exit in this case. Moreover, since $M = 1$, there is no possibility of processes requesting conflicting sessions, and hence the algorithm also satisfies mutual exclusion in this case. Let k be an integer such that $k \geq 2$. For the induction step, assume that the theorem holds for all values of M such that $1 \leq M < k$.

We now prove the theorem true for $M = k$. Let $\mathbb{A}_L$ be the $\lfloor k/2 \rfloor$-session algorithm whose trying and exit protocol is defined by TRYING_LEFT and EXIT_LEFT, respectively. Furthermore, let $\mathbb{A}_R$ be the $\lceil k/2 \rceil$-session algorithm whose trying and exit protocol is defined by TRYING_RIGHT and EXIT_RIGHT, respectively. By the induction hypothesis, $\mathbb{A}_L$ and $\mathbb{A}_R$ both satisfy mutual exclusion, lockout freedom, concurrent entering, and bounded exit.

We start by proving that mutual exclusion holds when $M = k$.

Let p be a process that requests session s_p and q be a process that requests session s_q, where $s_p \neq s_q$. Suppose, by way of contradiction, that p and q are in the CS simultaneously at time t. Either $p.mysession\_2S$ and $q.mysession\_2S$ have the same value, or different values.

Case 1: $p.mysession\_2S = q.mysession\_2S$.

In this case, both p and q must execute TRYING_LEFT, or they both must execute TRYING_RIGHT. Assume p and q both execute TRYING_LEFT. The mutual exclusion property of $\mathbb{A}_L$ implies that at most one of p or q can be enabled to execute lines 11..15 at a given time. This contradicts the assumption p and q are in the CS simultaneously at time t. Now assume p and q both execute TRYING_RIGHT. Then the values p and q pass to the call TRYING_RIGHT are in the range $[1, \lceil M/2 \rceil]$, and since $s_p \neq s_q$, the values passed to TRYING_RIGHT are different. Thus, by the mutual exclusion property of $\mathbb{A}_R$, at most one of p or q can be enabled to execute lines 11..15 at a given time. This contradicts

the assumption p and q are in the CS simultaneously at time t.

Case 2: $p.mysession\_2S \neq q.mysession\_2S$.

In this case, the mutual exclusion property of $\mathbb{A}_2$ prevents p and q from being in the CS simultaneously at time t, contradicting the assumption p and q are both in the CS at time t.

This completes our proof that mutual exclusion holds when $M = k$. We now prove that lockout freedom holds when $M = k$.

Suppose, by way of contradiction, that a process p is stuck forever in the trying protocol. Process p must be stuck in either TRYING_LEFT, TRYING_RIGHT, or TRYING_2S.

Claim 6.16.1: Process p cannot be stuck in TRYING_2S or EXIT_2S.

Proof: The bounded exit property of $\mathbb{A}_2$ immediately implies that p cannot be stuck in EXIT_2S. Suppose, by way of contradiction, that p is stuck in TRYING_2S. Then the lockout freedom property of $\mathbb{A}_2$ implies that there is some process q that executed TRYING_2S but then did not execute EXIT_2S. The only way for this to happen is if q stays in the CS forever; this is impossible. Hence q must execute EXIT_2S, contradicting the fact that q does not execute EXIT_2S. □ (Claim 6.16.1)

Claim 6.16.2: Process p cannot be stuck in TRYING_LEFT or TRYING_RIGHT.

Proof: Suppose, by way of contradiction, that p is stuck in TRYING_LEFT; the argument for TRYING_RIGHT is analogous. By the lockout freedom property of $\mathbb{A}_L$, the assumption that p is stuck in TRYING_LEFT implies that some other process q executed TRYING_LEFT but then did not execute EXIT_LEFT. The only way for this to happen is if q is stuck in one of TRYING_2S, EXIT_2S, or the CS. The first two alternatives contradict Claim 6.16.1, and the third is impossible. □ (Claim 6.16.2)

Claim 6.16.1 and Claim 6.16.2 contradict the fact that p is stuck in either TRYING_LEFT, TRYING_RIGHT, or TRYING_2S.

This completes our proof that lockout freedom holds when $M = k$. We now prove that concurrent entering holds when $M = k$.

Let p be a process requesting session s_p that is in its waiting room at time t_1, and t_2 be the time when p enters the CS. We show that if there is no active process that is requesting a different session in the interval $[t_1, t_2]$, then p enters the CS in a bounded number of its own steps. There are three possible places in the waiting room where p can be waiting at time t_1: TRYING_LEFT, TRYING_RIGHT, or TRYING_2S. If p is in TRYING_LEFT or TRYING_RIGHT, then the concurrent entering property of $\mathbb{A}_L$ and $\mathbb{A}_R$ implies that p can finish executing TRYING_LEFT and TRYING_RIGHT in a bounded number of its own steps. If p is in TRYING_2S, then the concurrent entering property of $\mathbb{A}_2$ implies that p will finish executing TRYING_2S in a bounded number of its own steps.

This completes our proof that concurrent entering holds when $M = k$. We now prove that bounded exit holds when $M = k$.

By the bounded exit property of $\mathbb{A}_2$, a process p can finish EXIT_2S in a bounded number of its own steps, and by the bounded exit property of $\mathbb{A}_L$ and $\mathbb{A}_R$, p can finish EXIT_LEFT and EXIT_RIGHT in a bounded number of its own steps. □

6.2.4 RMR Complexity

Let $T(M)$ be the worst-case RMR complexity of the M-session GME algorithm in Figure 6.3, and let $f(N)$ be the worst-case RMR complexity of the 2-session N-process GME algorithm in Figure 6.1.

It is clear from the algorithm that $T(M)$ satisfies the following recurrence:

$T(1) = 0$

$T(2) \leq f(N)$

$T(3) \leq 2f(N)$

$T(M) \leq T(\lceil M/2 \rceil) + f(N)$, if $M \geq 4$

A solution to this recurrence yields $T(M) = O(\log M \cdot f(N))$. Hence the algorithm in Figure 6.3 has RMR complexity $O(\log M \cdot f(N))$.

Chapter 7

Conclusion

The major contribution of this thesis is the presentation of a number of local-spin group mutual exclusion algorithms — each in the form of a reduction to a FCFS abortable mutual exclusion algorithm — that satisfy the concurrent entering property defined by Hadzilacos [8]. To our knowledge, there are no other such algorithms currently in existence. In addition to this, we have proven that the RMR lower bound under the DSM model for any 2-session group mutual exclusion algorithm (and hence for any GME algorithm) is $\Omega(N)$. This lower bound holds irrespective of the synchronization primitives allowed.

Of the GME algorithms we have presented, several are intended to run under the DSM model, and one is intended to run under the CC model. The algorithms intended to run under the DSM model have RMR complexity $O(N)$, and thus, given the lower bound we proved, are optimal (within a constant factor) with respect to RMR complexity. The algorithm intended to run under the CC model has RMR complexity $O(\log M \cdot f(N))$, where M is size of the set from which sessions are requested, and $O(f(N))$ is the RMR complexity of the FCFS abortable mutual exclusion algorithm used in the reduction. It is based on our 2-session GME algorithm that has RMR complexity $O(f(N))$ under the CC model. This latter result, together with the $O(\min(k, \log N))$ FCFS abortable

ME algorithm of Jayanti [9], shows that our lower bound proof necessarily requires the assumption that processes run under the DSM model.

We conclude by mentioning several open problems. First, the RMR lower bound under the CC model is unknown. Moreover, the amortized RMR lower bound under either the DSM model or the CC model is unknown, although we conjecture that the amortized RMR lower bound under the DSM model is still $\Omega(N)$. Finally, our fair local-spin group mutual exclusion algorithm that satisfies FIFE, strong concurrent entering, and FCFS, uses an integer that grows without bound. This is not necessarily a practical limitation given the size of modern-day registers and the fact that, in this algorithm, the number in the unbounded register grows only linearly with the number of invocations. However, it is theoretically undesirable, and it is an open problem whether there exists a GME algorithm that satisfies all the fairness properties and uses only bounded registers.

Bibliography

[1] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2002.

[2] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, September 2003.

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

[4] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.

[5] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[6] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, pages 234–254, 1979.

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

[8] V. Hadzilacos. A note on group mutual exclusion. In *Proceedings of the 20th Annual Symposium on Principles of Distributed Computing*, pages 100–106, 2001.

[9] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

[10] P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

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

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

[13] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, May 1966.

[14] L. Lamport. A new solution to Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

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

[16] M. L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, July 2002.

[17] M. L. Scott and W.N. Scherer III. Non-blocking timeout in scalable queue-based spin locks. In *Proceedings of the 8th ACM Symposium on Principles and Practice of Parallel Programming*, June 2001.

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