The Interdisciplinary Center, Herzliya

Efi Arazi School of Computer Science

# Constant RMR Group Mutual Exclusion for Arbitrarily Many Processes and Sessions

by

**Liat Maor**

M.Sc. dissertation, submitted in partial fulfillment of the requirements
for the Master's degree, School of Computer Science
The Interdisciplinary Center, Herzliya

September 8, 2021

This work was carried out under the supervision of Prof. Gadi Taubenfeld as part of the M.Sc. program of Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.

## Acknowledgments

I would like to thank my supervisor, Prof. Gadi Taubenfeld, for his invaluable advice, continuous support, and patience during my master study. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research.

My appreciation also goes out to my family and friends for their encouragement and support all through my studies.

## Thesis summary

In this work, a group mutual exclusion (GME) algorithm is presented. The algorithm consists of entry code and exit code. The entry code is executed before accessing the critical section, and the exit code is executed right after completing the critical section. Together, the entry code and the exit code assemble a GME lock, such that processes that request the same session may enter their critical sections concurrently while processes that request different sessions may not enter their critical sections simultaneously.

The algorithm that is presented in this work, is the first to achieve a constant Remote Memory Reference (RMR) complexity for both cache coherent (CC) and distributed shared memory (DSM) machines, and is the first that can be accessed by arbitrarily many dynamically allocated processes with arbitrarily many session names. The algorithm satisfies the following properties:

1. *Mutual exclusion*: Two processes can be in their CS at the same time, only if they request the same session.

2. *Starvation freedom*: If a process is trying to enter its CS, then this process must eventually enter its CS.

3. *Group concurrent entering (GCE)*: If a process p requests a session s while no process requesting a conflicting session, then (1) some process with session s can complete its entry section within a bounded number of its own steps, and (2) p eventually completes its entry section, even if other processes do not leave their CS. (The algorithm also satisfies strong-GCE)

4. *Group bounded exit (GBE)*: If a process p is in its exit section, then (1) some

process can complete its exit section within a bounded number of its own steps, and (2) p eventually completes its exit section.

5. *First-come-first-served (FCFS)*: If a process p completes its doorway before a process q enters its doorway and the two processes request different sessions, then q does not enter its CS before p enters its CS.

6. *First-in-first-enabled (FIFE)*: If a process p completes its doorway before a process q enters its doorway, the two processes request the same session, and q enters its CS before p, then p enters its CS in a bounded number of its own steps.

7. *Suitability for dynamic systems*: The algorithm can be accessed by an arbitrarily number of processes, that is, processes may appear or disappear intermittently; and the number and names of the sessions are not limited in any way.

8. $O(1)$ *RMR complexity*: An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache or memory and must transact over the multiprocessor's interconnection network in order to complete the operation. This algorithm achieve the ideal RMR complexity of $O(1)$ for both Cache Coherent (CC) and Distributed Shared Memory (DSM) machines.

9. $O(1)$ *space per process*: A small constant number of memory locations are allocated for each process.

10. Atomic instruction set: read, write, Fetch-And-Store (FAS), Compare-And-Swap (CAS)

The properties GCE (also strong-GCE) and GBE are first introduced in this work to circumvent the lower bound from [9].

# Abstract

Group mutual exclusion (GME), introduced by Joung in 1998, is a natural synchronization problem that generalizes the classical mutual exclusion and readers and writers problems. In GME a process requests a session before entering its critical section; processes are allowed to be in their critical sections simultaneously provided they have requested the same session.

We present a GME algorithm that (1) is the first to achieve a constant Remote Memory Reference (RMR) complexity for both cache coherent and distributed shared memory machines; and (2) is the first that can be accessed by arbitrarily many dynamically allocated processes and with arbitrarily many session names. Neither of the existing GME algorithms satisfies either of these two important properties. In addition, our algorithm has constant space complexity per process and satisfies the two strong fairness properties, first-come-first-served and first-in-first-enabled. Our algorithm uses an atomic instruction set supported by most modern processor architectures, namely: read, write, fetch-and-store and compare-and-swap.

# Contents

# 1 Introduction

## 1.1 Motivation and results

In the *group mutual exclusion* (GME) problem $n$ processes repeatedly attend $m$ sessions. Processes that have requested to attend the same session may do it concurrently. However, processes that have requested to attend different sessions may not attend their sessions simultaneously. The GME problem is a natural generalization of the classical mutual exclusion (ME) and readers/writers problems [7, 10]. To see this, observe that given a GME algorithm, ME can be solved by having each process use its unique identifier as a session number. Readers/writers can be solved by having each writer request a different session, and having all readers request the same special session. This allows readers to attend the session concurrently while ensuring that each writer attends in isolation. The GME problem has been studied extensively since it was introduced by Yuh-Jzer Joung in 1998 [19, 20].

A simple usage example has to do with the design of a concurrent queue or stack [5]. Using a GME algorithm, we can guarantee that no two users will ever simultaneously be in the *enqueue.session* or *dequeue.session*, so the enqueue and dequeue operations will never be interleaved. However, it will allow any number of users to be in either the enqueue or dequeue session simultaneously. Doing so simplifies the design of a concurrent queue as our only concern now is to implement concurrent enqueue operations and concurrent dequeue operations.

In this work, we present a GME algorithm that is the first to satisfy several desired properties (the first two properties are satisfied only by our algorithm).

1. *Suitability for dynamic systems*: All the existing GME algorithms are de-

signed with the assumption that either the number of processes or the number of sessions is a priori known. Our algorithm is the first that does not make such an assumption:

- it can be accessed by an arbitrary number of processes; that is, processes may appear or disappear intermittently, and
- the number and names of the sessions are not limited in any way.

2. $O(1)$ *RMR complexity*: An operation that a process performs on a memory location is considered a remote memory reference (RMR) if the process cannot perform the operation locally on its cache or memory and must transact over the multiprocessor's interconnection network in order to complete the operation. RMRs are undesirable because they take long to execute and increase the interconnection traffic. Our algorithm

   - achieves the optimal RMR complexity of $O(1)$ for Cache Coherent (CC) machines; and
   - is the first to achieve the optimal RMR complexity of $O(1)$ for Distributed Shared Memory (DSM) machines. (In Subsection 1.3, we explain why this result does not contradict the lower bound from [9].)

   This means that a process incurs only a constant number of RMRs to satisfy a request (i.e, to enter and exit the critical section once), regardless of how many other processes execute the algorithm concurrently.

3. $O(1)$ *space per process*: A small constant number of memory locations are allocated for each process. On DSM machines, these memory locations

reside in the process local memory; on CC machines, these locations reside in the shared memory.

4. *Strong fairness*: Requests are satisfied in the order of their arrival. That is, our algorithm satisfies the *first-come-first-served* and *first-in-first-enabled* properties, defined later.

5. *Hardware support*: Atomic instruction set that is supported by most modern processor architectures is used, namely: read, write, fetch-and-store and compare-and-swap.

We point out that when using a GME as a ME algorithm, the number of processes is the same as the number of sessions (each process uses its identifier as its session number). Thus, in GME algorithms, in which the number of sessions is a priori known also the number of processes must be known, at least when these GME algorithms are used as ME algorithms or readers and writers locks.

Our GME algorithm is inspired by J. M. Mellor-Crummey and M. L. Scott MCS queue-based ME algorithm [25]. The idea of our GME algorithm is to employ a queue, where processes insert their requests for attending a session. The condition when a process $p$ may attend its session depends on whether $p$'s session is the same as that of all its predecessors. Otherwise, $p$ waits until $p$ is notified (by one of its predecessors) that all its predecessors which have requested different sessions completed attending their sessions.

A drawback of the MCS ME algorithm is that releasing a lock requires spinning – a process $p$ releasing the lock may need to wait for a process that is trying to acquire the lock (and hence is behind $p$ in the queue) to take a step before $p$ can proceed. The ME algorithms in [11] overcome this drawback while preserving the simplicity, elegance, and properties of the MCS algorithm. We use a key idea inspired by [11] in our GME algorithm to ensure that a process releasing the GME lock will never have to wait for a process that has not attended its session yet.

Another key idea of our algorithm is to count down completed requests for attending a session by moving a pointer by one node (in the queue) for each such request and to ensure the integrity of this scheme by gating the processes that have completed attending a session (and are now trying to move the pointer) through a mutual exclusion lock.

## 1.2 The GME problem

Formally, the GME problem is defined as follows: it is assumed that each process executes a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections of code: the *remainder, entry, critical section (CS)*, and *exit*.

A process starts by executing its remainder section. At some point, it might need to attend some session, say $s$. To attend session $s$, a process has to go through an entry code that guarantees that while it is attending this session, no other process is allowed to attend another session. In addition, once a process completes attending a session, the process executes its exit section in which it notifies other processes that it is no longer attending the session. After executing its exit section, the process returns to its remainder.

The group mutual exclusion problem is to write the code for the *entry section* and the *exit section* so that the following requirements are satisfied.

- *Mutual exclusion*: Two processes can be in their CS at the same time, only if they request the same session.

- *Starvation-freedom*: If a process is trying to enter its CS, then this process must eventually enter its CS.

- *Group concurrent entering* (*GCE*): If a process $p$ requests a session $s$ while no process is requesting a conflicting session, then (1) some process with session $s$ can complete its entry section within a bounded number of its own steps, and (2) $p$ eventually completes its entry section, even if other processes do not leave their CS.

- *Group bounded exit* (*GBE*): If a process $p$ is in its exit section, then (1) some process can complete its exit section within a bounded number of its own steps, and (2) $p$ eventually completes its exit section.

GCE precludes using a given mutual exclusion algorithm as a solution for the GME problem since GCE enables processes to attend the same session concurrently.

Our algorithm also satisfies the following strong fairness requirements. To formalize this, we assume that the entry code starts with a bounded section of code (i.e., one that contains no unbounded loops), called the *doorway*; the rest of the entry code is called the waiting room. The fairness requirements, satisfied by our algorithm, can now be stated as follows:

- *First-come-first-served* (*FCFS*): If a process $p$ completes its doorway before a process $q$ enters its doorway and the two processes request different sessions, then $q$ does not enter its CS before $p$ enters its CS [14, 24].

- *First-in-first-enabled* (*FIFE*): If a process $p$ completes its doorway before a process $q$ enters its doorway, the two processes request the same session, and $q$ enters its CS before $p$, then $p$ enters its CS in a bounded number of its own steps [18].

We notice that FCFS and FIFE do not imply starvation-freedom or group concurrent entering.

## 1.3 Further explanations

To illustrate the various GME requirements, imagine the critical section as a lecture hall that different professors can share for their lectures. Furthermore, assume that the lecture hall has one entrance door and one exit door. When solving the GME problem, the property of mutual exclusion guarantees that two different lectures cannot be arranged in the lecture hall simultaneously, while starvation-freedom guarantees that the lecture hall will eventually be reserved for every scheduled lecture.

Assuming that only one lecture is scheduled, group concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall through the entrance door, possibly one after the other, and attend the lecture. Furthermore, at any given time, when there are students who want to attend the lecture, at least one of them can always enter the lecture hall without any delay. Similarly, group bounded exit ensures that all the students who want to leave a lecture can do so through the exit door, possibly one after the other. Furthermore, at any given time, at least one of them can exit the lecture hall without delay.

Group concurrent entering and group bounded exit are first introduced and formally defined in this work. They are slightly weakened versions of two known requirements (formally defined below) called concurrent entering and bounded exit. Using the lecture hall metaphor, assuming that only one lecture is scheduled, concurrent entering ensures that all the students who want to attend this lecture can enter the lecture hall *together*. Similarly, bounded exit ensures that all the students who want to leave a lecture can do so *together*. So, why have we not used these two stronger requirements?

**Danek and Hadzilacos lower bound.** Let $n$ denotes the total number of processes. In [9], it is proven that $\Omega(n)$ RMRs are required for any GME algorithm that satisfies mutual exclusion, starvation-freedom, concurrent entering, and bounded exit, in the DSM model, using basic primitives of any strength. This result holds even when the number of sessions is only two. (Concurrent entering and bounded exit are as defined below [14].) Since we are aiming at finding a solution that has $O(1)$ RMR complexity, we had to weaken either concurrent entering, bounded exit, or both. (GME would not be interesting if the mutual exclusion or starvation-freedom properties are weakened.)

**Group concurrent entering.** To avoid an inefficient solution to the GME problem using a traditional ME algorithm and forcing processes to be in their CS one-at-a-time, even if all processes are requesting the same session, Joung required that a GME algorithm satisfies the following property (which he called concurrent entering):

- If some processes request a session and no process requests a different session, then the processes can concurrently enter the CS [19].

The phrase "can concurrently enter," although suggestive, is not precise. In [22, 23], Keane and Moir were the first to give a precise definition that captures their interpretation of Joung's requirement (which they also called concurrent entering):

- *Concurrent occupancy*: If a process $p$ requests a session and no process requests a different session, then $p$ eventually enters its CS, even if other processes do not leave their CS. (The name "concurrent occupancy" is from [14].)

In [14], Hadzilacos gave the following interpretation, which is stronger than that of Keane and Moir.

- *Concurrent entering*: If some process, say $p$, is trying to attend a session $s$ while no process is requesting a conflicting session, then $p$ completes its entry section in a bounded number of its own steps.

To circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we looked for a slightly weaker version of concurrent entering that would still capture the property that Joung intended to specify. We believe that group concurrent entering, which is strictly stronger than concurrent occupancy, is such a property. We point out that our algorithm actually satisfies the following stronger version of group concurrent entering,

- *Strong group concurrent entering*: If a process $p$ requests a session $s$, and $p$ completes its doorway before any conflicting process starts its doorway, then (1) some process with session $s$ can complete its entry section within a bounded number of its own steps, and (2) $p$ eventually completes its entry section, even if other processes do not leave their CS.

Strong group concurrent entering (SGCE) is a slightly weakened version of a known property called strong concurrent entering [18].

**Group bounded exit.** Our group bounded exit property is replaced by the following two (weaker and stronger) properties in previously published papers.

- *Terminating exit*: If a process $p$ enters its exit section, then $p$ eventually completes it [22].
- *Bounded exit*: If a process $p$ enters its exit section, then $p$ eventually completes it within a bounded number of its own steps [14].

Again, to circumvent the Danek and Hadzilacos $\Omega(n)$ lower bound, we have defined group bounded exit, which is slightly weaker than bounded exit and is strictly stronger than terminating exit.

**Open question.** We have modified both concurrent entering and bounded exit. Is this necessary? With minor modifications to the Danek and Hadzilacos lower bound proof, it is possible to prove that their lower bound still holds when replacing only bounded exit with group bounded exit. Thus, to circumvent the lower bound, the weakening of concurrent entering is necessary. However, the question of whether it is possible to circumvent the lower bound by replacing only concurrent entering with group concurrent entering, and leaving bounded exit as is, is open.

## 1.4 Related work

Table 1 summarizes some of the (more relevant) GME algorithms mentioned below and their properties. The group mutual exclusion problem was first stated and solved by Yuh-Jzer Joung in [19, 20], using atomic read/write registers. The problem is a generalization of the mutual exclusion problem [10] and the readers and writers problem [7] and can be seen as a special case of the drinking philosophers problem [6].

Group mutual exclusion is similar to the room synchronization problem [5]. The room synchronization problem involves supporting a set of $m$ mutually exclusive "rooms" where any number of users can execute code simultaneously in any one of the rooms, but no two users can simultaneously execute code in separate rooms. In [5], room synchronization is defined using a set of properties that is different than that in [19], a solution is presented, and it is shown how it can be used to efficiently implement concurrent queues and stacks.

In [22, 23], a technique of converting any solution for the mutual exclusion problem to solve the group mutual exclusion problem was introduced. The algorithms from [22, 23] do not satisfy group concurrent entering and group bounded exit and have $O(n)$ RMR complexity, where $n$ is the number of processes. (By mistake, in some of the tables in [22, 23], smaller RMR complexity measures are mentioned.) In [14], a simple formulation of concurrent entering is proposed which is stronger than the one from [22], and an algorithm is presented that satisfies this property.

In [18], the first FCFS GME algorithm is presented that uses only $O(n)$ bounded shared registers, while satisfying concurrent entering and bounded exit. Also, it is demonstrated that the FCFS property does not fully capture the intuitive notion of fairness, and additional fairness property, called first-in-first-enabled (FIFE) was presented. Finally, the authors presented a reduction that transforms any *abortable* FCFS mutual exclusion algorithm, into a GME algorithm, and used it to obtained GME algorithm satisfying both FCFS and FIFE.

A GME algorithm is presented in [9] with $O(n)$ RMR complexity in the DSM model, and it is proved that this is asymptotically optimal. Another algorithm in [9] requires only $O(\log n)$ RMR complexity in the CC model, but can be used just for two sessions.

Our algorithm satisfies FCFS fairness. That is, if the requests in the queue are for sessions 1, 2, 1, 2, 1, 2 and so on, those requests would be granted in that order. Yet, for practical considerations, one may want to batch all requests for session 1 (and, separately, for session 2) and run them concurrently. Our algorithm does not support "batching" of pending requests for the same session, as FCFS fairness and "batching" of pending requests for the same session are contradicting (incompatible) requirements. This idea was explored in [4], where a GME algorithm is presented that satisfies two "batching" requirements called pulling and relaxed-FCFS, and requiring only $O(\log n)$ RMR complexity in the CC model. Reader-Writer Locks were studied in [**?** ], which trade fairness between readers and writers for higher concurrency among readers and better back-to-back batching of writers.

An algorithm is presented in [12] in which a process can enter its critical section within a constant number of its own steps in the absence of any other requests (which is typically referred to as contention-free step complexity). In the presence of contention, the RMR complexity of the algorithm is $O(min(k, n))$, where $k$ denotes the interval contention. The algorithm requires $O(n^2)$ space and does not satisfy fairness property like FCFS or FIFE.

In [1], a GME algorithm with a constant RMR complexity in the CC model is presented. This algorithm does not satisfy group concurrent entering (or even concurrent occupancy) and FCFS. However, it satisfies two other interesting properties (defined by the authors) called simultaneous acceptance and forum-FCFS.

In [15], the first GME algorithm with both linear RMR complexity (in the CC model) and linear space was presented, which satisfies concurrent entering and bounded exit, and uses only read/write registers. A combined problem of $\ell$-exclusion and group mutual exclusion, called the group $\ell$-exclusion problem, is considered in [26, 28].

Besides the algorithms mentioned above, for the shared-memory model, there are algorithms that solve the GME problem under the message-passing model. Several types of the network's structure were considered, for example, tree networks [3], ring networks [29], and fully connected networks [2]. In [2, 21, 27], quorum-based message-passing algorithms are suggested in which a process that is interested in entering its CS has to ask permission from a pre-defined quorum.

## 2 Preliminaries

### 2.1 Computational model

Our model of computation consists of an asynchronous collection of $n$ deterministic processes that communicate via shared registers (i.e., shared memory locations). Asynchrony means that there is no assumption on the relative speeds of the processes. Access to a register is done by applying operations to the register. Each operation is defined as a function that gets as arguments one or more values and registers names (shared and local), updates the value of the registers, and may return a value. Only one of the arguments may be a name of a *shared* register. The execution of the function is assumed to be atomic. Call by reference is used when passing registers as arguments. The operations used by our algorithm are:

| GME Algorithms | Group bounded exit BE/GBE | Group concurrent entering CE/GCE | Fairness FCFS/ FIFE | Unknown number of processes & sessions | Shared space for all processes | RMR in CC | RMR in DSM | Hardware used |
|---|---|---|---|---|---|---|---|---|
| Joung 1988 | BE | CE | ✗ | ✗ | $O(n)$ | $\infty$ | $\infty$ | read/write |
| Keane & Moir 1999 | ✗ | ✗ | ✗ | ✗ | $O(n)$ | $O(n)$ | $O(n)$ | read/write |
| Hadzilacos 2001 | BE | CE | FCFS | ✗ | $O(n^2)$ | $O(n^2)$ | $\infty$ | read/write |
| Jayanti et.al. 2003 | BE | CE | FCFS FIFE | ✗ | $O(n)$ | $O(n^2)$ | $\infty$ | read/write |
| Danek&Hadzilacos 2004 | BE | CE | FCFS FIFE | ✗ | $O(n^2)$ | $O(n)$ | $O(n)$ | CAS fetch&add |
| Bhatt & Huang 2010 | BE | CE | ✗ | ✗ | $O(mn)$ | $O(min(k, \log n))$ | $\infty$ | LL/SC |
| He et. al. 2018 | BE | CE | FCFS | ✗ | $O(n)$ | $O(n)$ | $\infty$ | read/write |
| Aravid&Hesselink 2019 | BE | ✗ | FIFE | ✗ | $O(L)$ | $O(1)$ | $\infty$ | fetch&inc |
| Gokhale & Mittal 2019 | BE | CE | ✗ | ✗ | $O(n^2)$ | $O(min(c, n))$ | $O(n)$ | CAS fetch&add |
| **Our algorithm** | GBE | GCE | FCFS FIFE | ✓ | $O(n)$ | $O(1)$ | $O(1)$ | CAS fetch&store |

| | | |
|---|---|---|
| ✓ - satisfies the property | $k$ - point contention | BE - bounded exit |
| ✗ - does not satisfy the property | $c$ - interval contention | GBE - group bounded exit |
| $n$ - number of processes | $L$ - a constant number | CE - concurrent entring |
| $m$ - number of sessions | $s.t.\ L > min(n, m)$ | GCE - group concurrent entring |

Table 1: Comparing the properties of our algorithm with those of several GME algorithms.

- *Read:* takes a shared register $r$ and simply returns its value.
- *Write:* takes a shared register $r$ and a value *val*. The value *val* is assigned to $r$.
- *Fetch-and-store* (FAS): takes a shared register $r$ and a local register $\ell$, and atomically assigns the value of $\ell$ to $r$ and returns the previous value of $r$. (The fetch-and-store operation is also called *swap* in the literature.)

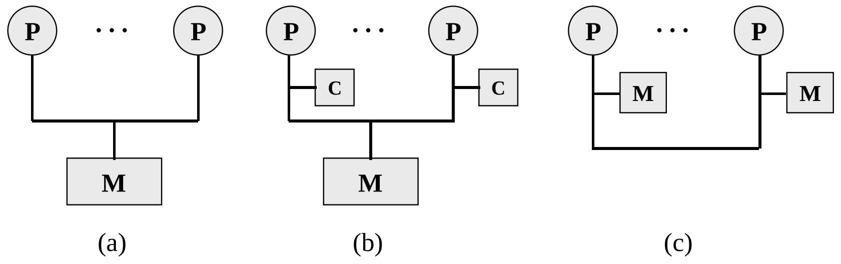

Figure 1: Shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory.

- *Compare-and-swap* (CAS): takes a shared register $r$, and two values: *new* and *old*. If the current value of the register $r$ is equal to *old*, then the value of $r$ is set to *new* and the value *true* is returned; otherwise, $r$ is left unchanged and the value *false* is returned.

Most modern processor architectures support the above operations.

## 2.2 The CC and DSM machine architectures

We consider two machine architecture models: (1) Cache coherent (CC) systems, where each process (or processor) has its own private cache. When a process accesses a shared memory location, a copy of it migrates to a local cache line and becomes locally accessible until some other process updates this shared memory location and the local copy is invalidated; (2) Distributed shared memory (DSM) systems, where instead of having the "shared memory" in one central location, each process "owns" part of the shared memory and keeps it in its own local memory. These different shared memory models are illustrated in Figure 1.

A shared memory location is locally accessible to some process if it is in the part of the shared memory that physically resides on that process' local memory.

Spinning on a remote memory location while its value does not change, is counted only as *one* remote operation that causes communication in the CC model, while it is counted as *many* operations that cause communication in the DSM model. An algorithm satisfies *local spinning* (in the CC or DSM models) if the only type of spinning required is local spinning.

## 2.3 RMR complexity: counting remote memory references

We define a *remote reference* by process $p$ as an attempt to reference (access) a memory location that does not physically reside in $p$'s local memory or cache. The remote memory location can either reside in a central shared memory or in some other process' memory.

Next, we define when remote reference causes *communication*. (1) In the DSM model, any remote reference causes communication; (2) in the CC model, a remote reference to register $r$ causes communication if (the value of) $r$ is not (the same as the value) in the cache. That is, communication is caused only by a remote write access that overwrites a different value or by the first remote read access by a process that detects a value written by a different process.

Finally, we define time complexity when counting only remote memory references. This complexity measure, called RMR complexity, is defined with respect to either the DSM model or the CC model, and whenever it is used, we will say explicitly which model is assumed.

- *The RMR complexity* in the CC model (resp. DSM model) is the maximum number of remote memory references which cause communication in the CC model (resp. DSM model) that a process, say $p$, may need to perform in its entry and exit sections in order to enter and exit its critical section since the last time $p$ started executing the code of its entry section.

## 3 The GME Algorithm

Our algorithm has the following properties: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes or the number of sessions is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS and FIFE fairness, (5) it satisfies the properties: mutual exclusion, starvation-freedom, SGCE, and GBE, (6) it uses an atomic instruction set supported by most modern processor architectures (i.e., read, write, FAS and CAS).

### 3.1 An informal description

The algorithm maintains a queue of nodes which is implemented as a linked list with two shared objects, *Head* and *Tail*, that point to the first and the last nodes, respectively. Each node represents a request of a process to attend a specific session. A node is an object with a pointer field called *next*, a boolean field called *go*, an integer field called *session*, and two status fields called $status$ and $active$. Each process $p$ has its own *two* nodes, called $Nodes_p[0]$ and $Nodes_p[1]$, which can be assumed to be stored in the process $p$'s local memory in a DSM machine, and in the shared memory in a CC machine. Each time $p$ wants to enter its CS section, $p$ uses alternately one of its two nodes. We say that a process $p$ is *enabled* if $p$ can enter its CS in a bounded number of its own steps.

In its doorway, process $p$ initializes the fields of its node as follows:

- *session* is set to the session $p$ wants to attend, letting other processes know the session $p$ is requesting (line 2).
- *next* is a pointer to the successor's node and is initially set to null. This field is being updated later by $p$'s successor (line 11).
- *go* is set to *false*. Later, if $p$ is not enabled, $p$ would spin on its $go$ bit until the value is changed to $true$. The *go* bit is the only memory location a process may spin on.
- *status* is set to *WAIT*. This field is being used to determine if a process is enabled. When a process becomes enabled, it sets this field to *ENABLED* (line 26). When process $p$ sees that its predecessor is not enabled (line 13), $p$ spins on its *go* bit (line 14). Otherwise, $p$ informs its predecessor that $p$ has seen that the predecessor is enabled (and hence $p$ does not need help), by setting its predecessor's *status* field to $NO\_HELP$. When a process $p$ sees that its *status* is *ENABLED* (line 30), $p$ tries to help its successor to become enabled and notifies the successor by setting $p$'s own *status* to $TRY\_HELP$.
- *active* is set to *YES*. This field is being used to determine whether $p$'s node is active or not. A node is active if there is a process $p$ that is currently using the node in an attempt to enter $p$'s critical section.

At the end of its doorway, process $p$ threads its node to the end of the queue (line 7). Afterward, $p$ checks what its state is. The state can be one of the following:

1. its node is the first in the queue,
2. its predecessor requests the same session, or

3. its predecessor requests a different session.

In the first case, $p$ can safely become enabled and enters its CS. In the second case, $p$ becomes enabled only if its predecessor is enabled. In the third case, $p$ eventually becomes enabled, once all the processes it follows completed their CSs. We observe that in the exit section, each process causes *Head* to be advanced by exactly one step. So, if $p$'s predecessor's node is inactive, it implies that all the processes that $p$ follows completed their CSs, and thus, $p$ can become enabled and enters its CS.

In the last two cases, once $p$ is enabled, $p$ checks whether it should help its predecessor advance *Head*, by checking if $p$'s predecessor's node is inactive. If the predecessor's node is inactive, then *Head* should point to the node after this inactive node, which is $p$'s node. Therefore, in such a case, $p$ advances *Head* to point to its node.

Once $p$ is enabled to enter its CS, $p$ notifies its successor by setting $p$'s *status* to *ENABLED*. Next, $p$ checks if it has a successor that requests the same session and needs help also to become enabled. If so, $p$ tries to help its successor to become enabled. Only then $p$ enters its CS. The processes that may enter their CS simultaneously are: the process, say $p$, that *Head* points to its node, and every process $q$ that (1) requests the same session as $p$, and (2) no conflicting process entered its node between $p$'s node and $q$'s node.

Most of the exit code is wrapped by a mutual exclusion lock. This ensures that each process can cause *Head* to be advanced by a single step every time a process completes its CS. A process $p$ that completes its CS and succeeds in acquiring the ME lock tries to advance *Head*. If $p$ succeeds in advancing *Head*, then *Head* value is either *null* or points to the next node in the queue. If *Head* is not *null*, $p$ changes the *go* bit to *true* in the node that *Head* points to. By doing so, $p$ lets the next process become enabled.

If $p$ fails to advance *Head*, this means that some other process either,

1. enters the queue after $p$ sets *Tail* to *null* (line 38),

2. enters the queue but has not notified its predecessor yet (line 11), or

3. has not entered the queue yet (line 7).

In the first case, the process, say $q$, in its entry section overrides *Head* to point to $q$'s node (line 9) because $q$'s predecessor is *null*, and so $q$ "advances" *Head* for $p$. In the latter cases, $q$ in its entry section overrides $Head$ to point to $q$'s own node because it sees $q$'s predecessor's node is inactive, and so $q$ "advances" *Head* for $p$. Afterward, $p$ releases the ME lock, changes the index of its current node (for the next attempt to enter $p$'s critical section), and completes its exit section.

To guarantee that our GME algorithm satisfies group bounded exit, the mutual exclusion used in the exit section (lines 36 and 49) must satisfy three properties, (1) starvation-freedom, (2) bounded exit, and (3) a property that we call *bounded entry*. Bounded entry is defined as follows: If a process $p$ is in its entry section, while no other process is in its critical section or exit section, some process can complete its entry section within a bounded number of its own steps.[1] While the important and highly influential MCS lock [25] does not satisfy bounded exit, there are variants of it, like the mutual exclusion algorithms from [8, 11, 17], that satisfy all the above three properties.

[1]It is interesting to notice that the bounded entry property cannot be satisfied by a ME algorithm that uses only read/write atomic registers [**?** ], [**?** ] (page 119).

We will use one of the mutual exclusion algorithms from [11, 17], since (in addition to satisfying the above three properties) each of these algorithms satisfies the following properties which match those of our GME algorithm: (1) it has constant RMR complexity in both the CC and the DSM models, (2) it does not require to assume that the number of participating processes is a priori known, (3) it uses constant space per process, (4) it satisfies FCFS, (5) it uses the same atomic instruction set as our algorithm, (6) it makes no assumptions on what and how memory is allocated (in [8] it is assumed that all allocated pointers must point to even addresses).

## 3.2 The algorithm

Two memory records (nodes) are allocated for each process. On DSM machines, these two records reside in the process local memory; on CC machines, these two records reside in the shared memory. In the algorithm, the following symbols are used:

**&** – this symbol is used to obtain an object's memory location address (and not the value in this address). For example, &$var$ is the memory location address of variable $var$.

$\rightarrow$ – this symbol is used to indicate a pointer to data of a field in a specific memory location. For example, assume $var$ is a variable that is a struct with a field called $number$. We now define another variable $loc :=$ &$var$ s.t. $loc$ points to $var$. Using $loc \rightarrow number$ we would get the value of $var.number$.

**Q** – the queue in the algorithm is denoted by Q. Q is only used for explanations and does not appear in the algorithm's code.

**Algorithm 1** The GME algorithm: Code for process $p$

```
Type:    $QNode$: { session: int, go: bool, next: QNode*,
                 active: $\in$ {YES, NO, HELP}
                 status: $\in$ {ENABLED, WAIT, TRY_HELP, NO_HELP} }
Shared:  $Head$: type QNode*, initially null                    ▷ pointer to the first node in Q
         $Tail$: type QNode*, initially null                    ▷ pointer to the last node in Q
         $Lock$: type ME lock                                   ▷ mutual exclusion lock
         $Nodes_p[0, 1]$: each of type QNode, initial value immaterial  ▷ nodes local to $p$
in DSM
Local:   $s$: int                                               ▷ the session of $p$
         $node_p$: type QNode*, initial value immaterial    ▷ pointer to $p$'s currently used
node
         $pred_p$: type QNode*, initial value immaterial ▷ pointer to $p$'s predecessor node
         $next_p$: type QNode*, initial value immaterial    ▷ pointer to $p$'s successor node
         $temp\_head_p$: type QNode*, initial value immaterial      ▷ temporarily save the
head
         $current_p$: $\in$ {0, 1}, initial value immaterial    ▷ the index for $p$'s current node

procedure Thread($s$: int)                          ▷ $s$ is the session $p$ wants to attend
                                                                        ▷ Begin Doorway
 1: $node_p := \&Nodes_p[current_p]$   ▷ pointer to current node for this attempt to enter
    $p$'s CS
 2: $node_p \rightarrow session := s$                                ▷ $p$'s current session
 3: $node_p \rightarrow go :=$ false                           ▷ may spin locally on it later
 4: $node_p \rightarrow next :=$ null                               ▷ pointer to successor
```

```
5: $node_p \to status$ := $WAIT$                                  ▷ $p$ isn't enabled
6: $node_p \to active$ := $YES$                                   ▷ $p$'s node is active

7: $pred_p$ := FAS(Tail, $node_p$)                ▷ $p$ enters its current node to Q
                                                                  ▷ End Doorway
8: if $pred_p = \text{null}$ then                ▷ was Q empty before $p$ entered?
9:     Head := $node_p$                                ▷ $node_p$ is the first in Q
10: else                                                          ▷ $p$ has pred
11:     $pred_p \to next$ := $node_p$                                ▷ notify pred
12:     if $pred_p \to session = s$ then          ▷ do we have the same session?
13:         if not CAS($pred_p \to status$, $ENABLED$, $NO\_HELP$) then
                                              ▷ should wait for help from pred?
14:             await $node_p \to go = \text{true}$   ▷ wait until released by pred with the same
    session
15:         else if not CAS($pred_p \to active$, $YES$, $HELP$) then        ▷ should help
    advance Head?
16:             Head := $node_p$                                ▷ help advance Head
17:         end if
18:     else                                             ▷ we have different sessions
19:         if CAS($pred_p \to active$, $YES$, $HELP$) then   ▷ pred's node is still active?
20:             await $node_p \to go = \text{true}$
                        ▷ wait until release by a process with a different session
21:         else                        ▷ pred's node is inactive in Q thus $p$ is enabled
22:             Head := $node_p$
23:         end if
24:     end if
25: end if
```

```
26: $node_p \rightarrow status := ENABLED$                                          ▷ can enter the CS
                                                                     ▷ *Try helping the successor*
27: $next_p := node_p \rightarrow next$                                  ▷ save next pointer locally
28: **if** $next_p \neq$ null **then**                                             ▷ has successor?
29:     **if** $next_p \rightarrow session = s$ **then**                       ▷ we have the same session
30:         **if** CAS($node_p \rightarrow status$, *ENABLED*, *TRY_HELP*) **then**
31:             $next_p \rightarrow go :=$ true                          ▷ make your successor enabled
32:         **end if**
33:     **end if**
34: **end if**

35: ***critical section***

36: $Acquire(Lock)$                                         ▷ *Mutual exclusion entry section*

37: $temp\_head_p :=$ Head                                              ▷ save current head locally
38: **if** CAS(Tail, $temp\_head_p$, null) **then**        ▷ remove node from tail if it is the only
    node in Q
39:     CAS(Head, $temp\_head_p$, null)                        ▷ try removing it from the head
40: **else if** $temp\_head_p \rightarrow next \neq$ null **then**                  ▷ head has successor
41:     $temp\_head_p := temp\_head_p \rightarrow next$                     ▷ advance the temp head
42:     Head := $temp\_head_p$                                                   ▷ advance the head
43:     $temp\_head_p \rightarrow go :=$ true                                   ▷ enable the new head
44: **else if not** CAS($temp\_head_p \rightarrow active$, *YES*, *NO*) **then**
                          ▷ someone in Tail but hasn't notify to its predecessor in time
45:     $temp\_head_p := temp\_head_p \rightarrow next$                     ▷ advance the temp head
```

```
46:     Head := $temp\_head_p$                          ▷ advance the head
47:     $temp\_head_p \rightarrow go$ := true          ▷ enable the new head
48: end if

49: $Release(Lock)$                                     ▷ Mutual exclusion exit section

50: $current_p := 1 - current_p$                        ▷ toggle for further use
end procedure
```

## 3.3 Further explanations

To better understand the algorithm, we explain below several delicate design issues which are crucial for the correctness of the algorithm.

1. *Why does each process $p$ need two nodes $Nodes_p[0]$ and $Nodes_p[1]$?* This is done to avoid a *deadlock*. Assume each process has a single node instead of two, and consider the following execution. Suppose $p$ is in its CS, and $q$ completed its doorway. $p$ resumes and executes its exit section. $p$ completes its exit section while $q$ is in the queue but has not notified $p$ that $q$ is $p$'s successor (line 11). $p$ leaves its $status$ field as *ENABLED* and changes its $active$ field to *NO* (line 44), so $q$ should be able to enter its CS, no matter what session $q$ requests. $p$ starts another attempt to enter its CS, before $q$ resumes and executes either line 13 or line 19 (depends on which session $p$ requests). $p$ uses its single node and sets $status$ to *WAIT* and $active$ to *YES* in its doorway (lines 5 and 6, respectively). Notice, in that execution, $q$'s predecessor's node is $p$'s node while $p$'s predecessor's node is $q$'s node, see figure 2.

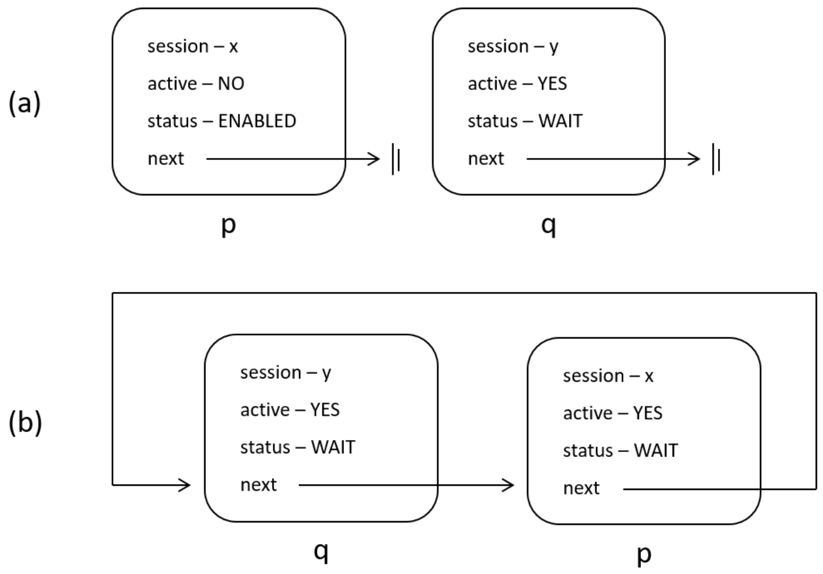

Figure 2: Deadlock scenario while using only one node for each process. (a) process $p$ completes its iteration while $q$ hasn't executed line 11 yet. (b) process $p$ starts the algorithm again with the same node.

Now, $q$ continues and (by executing either line 13 or line 19) sees that $p$ is not enabled and $p$'s node is active, so $q$ spins on its $go$ bit. Also, $p$ (by executing either line 13 or line 19) sees that $q$ is not enabled and its node is active, so $p$ also spins on its $go$ bit. No process will release $q$, and a deadlock occurs. This problem is resolved by having each process owns two nodes.

2. *Why do we need the CAS operations at lines 13 and 30?* The CAS operations at these lines prevent a potential race condition that may violate the *mutual exclusion* property.

   Assume we replace the CAS operations at lines 13 and 30, as follows: (see figure 3)

   - At line 13, $p$ checks if $pred_p \rightarrow status \neq ENABLED$. If so, $p$

```
...

13:  **if** $pred_p \to status \neq$ *ENABLED* **then**
14:      **await** $node_p \to go =$ true
     **else**
14.5:    $pred_p \to status :=$ *NO_HELP*
15:      **if not** CAS($pred_p \to active$, *YES*, *HELP*) **then**
16:          Head := $node_p$
17:      **end if**
18:  **end if**

...

27:  $next_p := node_p \to next$
28:  **if** $next_p \neq$ null **then**
29:      **if** $next_p \to session = s$ **then**
30:          **if** $node_p \to status =$ *ENABLED* **then**
30.5:            $node_p \to status :=$ *TRY_HELP*
31:              $next_p \to go :=$ true
32:          **end if**
33:      **end if**
34:  **end if**

...
```

Figure 3: The GME algorithm without CAS operation on $status$ field

waits at line 14. Otherwise, at line **14.5**, $p$ executes $pred_p \to status = NO\_HELP$.

- At line 30, $p$ checks if $node_p \to status = ENABLED$. If so, at line **30.5**, $p$ executes $node_p \to status = TRY\_HELP$ and then continues to line 31 and helps $p$'s successor.

Suppose $p$ is the predecessor of $q$, and they both request the same session $s$. $p$ executes line 30, sees that its own $status$ is still *ENABLED*, and continues

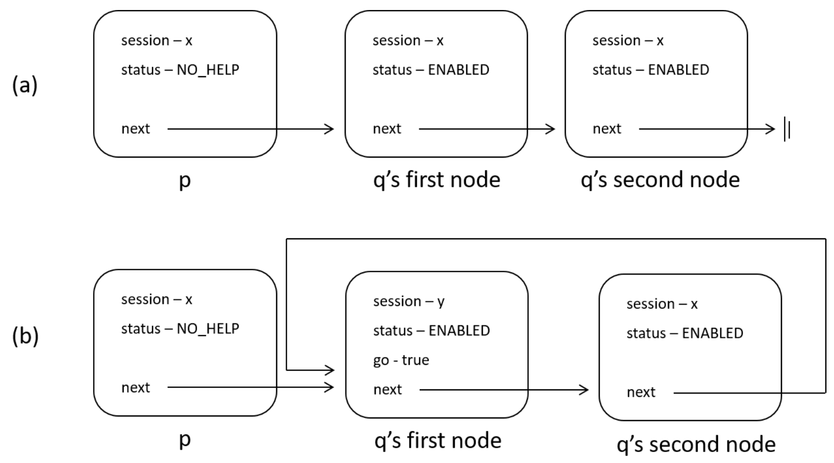

Figure 4: GME violation scenario while not using CAS operation on $status$. (a) process $q$ enters again with a different node. (b) process $q$ enters again with its first node.

to line 30.5 but does not execute this statement yet. Then, $q$ executes line 13, sees that $p$'s *status* is *ENABLED*, executes line 14.5, changes $p$'s *status* to $NO\_HELP$ and continues to $q$'s CS. $q$ completes its CS, executes $q$'s exit section, and starts the algorithm again using $q$'s second node. $q$ requests the same session as before, $s$, and continues to $q$'s CS since $q$'s predecessor is enabled. $q$ completes its exit code and enters the entry code again using $q$'s first node, but now $q$ requests a different session $s' \neq s$. Notice, $q$'s first node is the same node that $p$ has seen as its successor (see figure 4). $q$ continues to line 20 (because it does not request the same session as its predecessor). And so, $q$ waits until its *go* bit is set to *true*. Now, $p$ executes line 30.5 that changes $p$'s *status* to $TRY\_HELP$, continues to line 31 that sets $q$'s first node's *go* bit to *true* and enters its CS. $q$ sees that its *go* bit is *true* and also enters its CS. Therefore, both $p$ and $q$, which request different sessions, are in their CSs at the same time. This problem is resolved by using the

CAS operations, so a process atomically sees $status$ is $ENABLED$ and changes the value of $status$. That way, we guarantee that only one process could see that $status$ is $ENABLED$. If the process sees it at line 13, then its predecessor won't continue to line 31. If the process sees it at line 30, then its successor won't continue to the critical section before getting the help.

We use four different values for the $status$ field ($WAIT$, $ENABLED$, $TRY\_HELP$, $NO\_HELP$) to make the code more clear. Although, we only need two values and we can use a simple boolean variable. If we use a simple boolean variable, assume it is called $enabled$, then at line 26, the process sets its own $enabled$ to $true$, while at lines 13 and 30, the process may set this $enabled$ to $false$ while the process is still enabled.

3. *Why do we need the CAS operations at lines 15, 19, and 44?* The CAS operations at these lines are used to prevent a potential race condition that may cause a *deadlock*.

   Assume we replace the CAS operations are at lines 15, 19, and 44, as follows: (see figure 5)

   - At line 15, $p$ checks if $pred_p \to active \neq YES$. If so, $p$ sets *Head* to its node at line 16. Otherwise, at line **16.5**, $p$ executes $pred_p \to active = HELP$.
   - At line 19, $p$ checks if $pred_p \to active = YES$. If so, at line **19.5**, $p$ executes $pred_p \to active = HELP$.
   - At line 44, $p$ checks if $temp\_head_p \to active = YES$. If so, $p$ executes lines 45-47 and advances *Head*. Otherwise, $p$ continues to line

```
    ...

12:  if $pred_p \rightarrow session = s$ then
13:      if not CAS($pred_p \rightarrow status$, ENABLED, NO_HELP) then
14:          await $node_p \rightarrow go =$ true
15:      else if $pred_p \rightarrow active \neq$ YES then
16:          Head := $node_p$
         else
16.5:        $pred_p \rightarrow active :=$ HELP
17:      end if
18:  else
19:      if $pred_p \rightarrow active =$ YES then
19.5:        $pred_p \rightarrow active :=$ HELP
20:          await $node_p \rightarrow go =$ true
21:      else
22:          Head := $node_p$
23:      end if
24:  end if

    ...

44:  else if $temp\_head_p \rightarrow active =$ YES
45:      $temp\_head_p := temp\_head_p \rightarrow next$
46:      Head := $temp\_head_p$
47:      $temp\_head_p \rightarrow go :=$ true
     else
47.5:    $temp\_head_p \rightarrow active :=$ NO
48:  end if

    ...
```

Figure 5: The GME algorithm without CAS operation on $active$ field

**47.5** and executes $temp\_head_p \rightarrow active = NO$.

Suppose $p$ is at line 44 while its successor $q$ is at line 15. $q$ executes line 15 and sees its predecessor's node's *active* equals to *YES*. So $q$ continues to line 16.5 but does not execute it yet. Now, $p$ continues and sees that the *active* field of the first node in the queue is *YES*, so $p$ continues to line 47.5. Then, $p$ sets this node's *active* field to *NO*, while $q$ sets it to *HELP*. Next, $p$ completes its exit section and $q$ enters its CS. Since no process advanced *Head*, *Head* still points to the same node. Assume another process, $r$, wants to enter its CS and requests a different session than $q$. $r$ starts the algorithm and gets $q$'s node as its predecessor's node (at line 7). $r$ continues to line 19, as $r$ requests a different session than its predecessor $q$, and sees that its predecessor's node's *active* field is set to *YES*. Then, $r$ continues to line 19.5, notifying that it did not help to advance *Head*, and waits at line 20 for the $go$ bit to be set to *true*. $q$ completes its CS, advances *Head* at line 42, sets the new first node's *go* bit to *true* (line 43), and completes its exit code. But the new first node is $q$'s node, since no process advanced *Head* when $p$ completed its CS. All the new processes will wait until $r$ becomes enabled, but no process can help $r$ becoming enabled and a deadlock occurs. This problem is resolved by using the CAS operations at lines 15, 19, and 44, so a process atomically sees $active$ as $YES$ and changes its value. That way, we guarantee that only one process could see that $active$ is $YES$, while another process would see a different value and would advance $Head$.

We use three different values for the $active$ field ($YES$, $NO$, $HELP$) to make the code more clear. Although, we only need two values and we can use a simple boolean variable. If we use a simple boolean variable, assume it is called $active$, then at line 6, each process sets its $active$ field to $true$, and at lines 15, 19 and 44, we use the CAS operations to change $active$ from $true$ to $false$. While it's true that the node is not active when a process executes line 44, the node is still active when a process executes either line 15 or 19 but the $active$'s value is changed to $false$.

4. *Why don't we use a dummy node?* The head is being set for the first time at line 9 by the first process that executes the algorithm. The head can be set in line 9 only by one process, the first process, because of the use of the FAS operation at line 7. Only the first process returns null from this operation. The other times that a process may set the head at line 9 is when another process, say $q$, sets the tail to null in $q$'s exit section, and then $q$ should set the head to null and clear the queue. That means the algorithm is returned to its initial state.

5. *Why do we need line 39, although the algorithm is correct without line 39?* We can remove line 39, and the algorithm would still be correct, as we would override Head at line 9 with the next process that executes the algorithm. We have added this line for semantics reasons, as we do not want to get into a situation where Head points to a node that is no longer active while there are no processes that want to execute the algorithm. That is, when no processes are executing the algorithm, Head and Tail should be null.

6. *Is it essential to include lines 43 and 47 within the ME critical section?* We can move lines 43 and 47 outside the ME critical section (CS), and the algorithm would still be correct. At these lines, we use a local variable $temp\_head$, which no other process can change. We placed these lines inside the ME CS for better readability. If we move these lines outside the ME CS, we would need to check if we executed line 38, line 40, or line 44, and only if we executed lines 40 or 44, we then should set $go$.

7. *Who can set process $p$'s go bit to $true$ when $p$ waits at line 20?* By inspecting the code, we can see that $p$'s $go$ bit can be changed to *true* either in the entry section (line 31) or in the exit section (lines 43 and 47). Assume $p$ spins on its $go$ bit at line 20. $p$ would stop spinning when its $go$ bit changed to *true* by another process. Since $p$ is at line 20, $p$ has already tested the condition at line 12 and got *false*. This means that $p$ has requested a different session than its predecessor. Thus, $p$'s predecessor will not reach line 31 because the predecessor will see (line 29) that its successor requests a different session. Each process that acquires the ME lock causes *Head* to be advanced by exactly one step. Therefore, the process that will change $p$'s $go$ bit to *true* is the last process that acquires the ME lock and requests the same session as $p$'s predecessor.

8. *The algorithm might become simpler if one can obviate the use of Head. Is the use of Head necessary?* We have tried to simplify the algorithm by not using Head, as done for mutual exclusion in the implementation of the MCS lock [25]. Solving the GME problem is more complex than solving ME. There are more possible race conditions that should be avoided, and using Head helped us in the design of the algorithm. In particular, in the exit code, in lines 43 & 47 the new process at the head of the queue is enabled, by a process that is exiting. We do not see how to implement this in constant time without using Head.

## 4 Correctness proof

We prove that our algorithm satisfies the following properties: *mutual exclusion*, *starvation freedom*, *group bounded exit*, *strong group concurrent entering*, *FCFS* and *FIFE*. Also, we prove our algorithm uses only $O(n)$ shared memory locations (that is, constant space complexity per process) and its RMR complexity is $O(1)$ in both the CC and the DSM models. To prove all of these properties we will also show that the algorithm satisfies the following properties:

- *Deadlock freedom*: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section. (Notice that deadlock freedom is a weaker property than starvation freedom.)
- *Strong-FCFS*: If a process $p$ completes its doorway before a process $q$ (enters or) completes its doorway and the two processes request different sessions, then $q$ does not enter its critical section before $p$ does.

Consider an execution $e$. The notions, notations, definitions, lemmas and theorems in this proof are with respect to $e$. The following notions and notations are used in this proof:

1. **Doorway**: A process $p$ is considered to be in its *doorway* while running statements 1-7.
2. **The $i$-th iteration**: A process $p$ during its $i$-th iteration (i.e., its $i$-th attempt to enter its critical section) is denoted by $p^i$.
3. **Enabled**: A process $p^i$ is enabled if it can enters its critical section within a bounded number of its own steps.

4. **Follows**: A process $q^j$ *follows* a process $p^i$ if and only if $p^i$ completes its doorway before $q^j$.

5. **Predecessor**: A process $p^i$ is the *predecessor* of a process $q^j$ if and only if $q^j$ follows $p^i$, and each process that follows $p^i$ also follows $q^j$.

6. **Successor**: A process $q^j$ is the *successor* of a process $p^i$ if and only if $p^i$ is the predecessor of $q^j$.

7. **Passed over**: A pointer $x$ *passed over* $p^i$'s node if and only if $x$ points to $q^j$'s node and $q^j$ follows $p^i$.

8. **Q**: A $p^i$'s node is considered to be in Q if and only if $node_p$ passed over Tail and Head didn't pass over $p^i$'s node.

**Lemma 1.** *For every process* $p$ *at iteration* $i$ *and process* $q$ *at iteration* $j$*,*

- *Each adds exactly one node to Q.*
- *Each has at most one predecessor and at most one successor.*
- $p^i$ *and* $q^j$ *have different predecessors and different successors.*

*Proof.* The fact that a process in a specific iteration adds exactly one node to Q, may only have a single predecessor, and $p^i$ and $q^j$ have different predecessors follows from the fact that the last step of the doorway at line 7 is an atomic Fetch-And-Set (FAS) operation, which updates $pred$ with the last pointer that was in Tail and atomically updates Tail with the process' node, and so add it to Q. Each process has at most one predecessor initialized at line 7 into $pred$ variable. At line 11, each process updates its predecessor's successor with its own node. Then, each process has at most one successor. Assume on the contrary that $p^i$ and $q^j$ have the same successor, say $r^t$, that means either $r^t$ has two predecessors in contradiction to this lemma, or $r^t$ runs the algorithm twice, but each process runs an iteration only once. Therefore, $p^i$ and $q^j$ have different successors. $\square$

**Remark.** *Assume Head points to $p^i$'s node and $q^j$ follows $p^i$ and $x - 1$ processes that follow $p^i$. In the sequel, when we write "a process causes Head to be advanced by $x$ steps", we mean it either (1) sets Head to point to $q^j$'s node, or (2) lets another process know it should advance Head to $q^j$'s node. Also, when we write "Head's node" we refer to the node that Head points to.*

**Lemma 2.** *Each process $p^i$ that completes its exit section, causes Head to be advanced by exactly one step.*

*Proof.* Assume on the contrary that $p^i$ completes its exit section but does not cause Head to be advanced by exactly one step.

$p^i$ completed its CS and started its exit section. We assumed that $p^i$ has already completed its exit section, so it had to acquire the mutual exclusion (ME) lock $Lock$ at line 36, execute the ME CS (lines 37-48), and release $Lock$ at line 49. At the beginning of the ME CS, $p^i$ saved the current value of Head in its local variable $temp\_head$. The ME CS includes three lines that can change Head, lines 39, 42, and 46. Any process that enters its exit section can execute at most one of these lines as a result of the if-else statement. There are three options:

1. $p^i$ got true at line 40 or line 44. Then, it continued to line 41 or 45 respectively and saved in its local variable $temp\_head$ the successor of the node that Head pointed to. It continued to line 42 or 46 respectively and set Head to its $temp\_head$ which advanced Head by one step.

2. $p^i$ got true at line 38, so at that point, Tail was equal to Head, which means that there was only one node in Q. Then, no process changed Tail. That means no process has executed line 7 yet. Thus, both Tail and Head didn't have a successor at this moment. $p^i$ got true at line 38, so it continued to line 39. Here we have two more options:

   - $p^i$ successfully executed the CAS operation at line 39, so no other process changed Head at this point, and $p^i$ atomically changed Head to be null. Notice that the successor of Head was null, so $p^i$ advanced Head to be its successor, which means $p^i$ advanced Head by exactly one step.
   - $p^i$ executed line 39 and the CAS operation at this line failed, so another process changed Head after $p^i$ executed line 37 but before it executed line 39. These lines are being executed inside the ME CS along with the other lines that change Head in the exit code (lines 42 and 46). So while $p^i$ was about to execute line 39, no other process could change Head in its exit section. Therefore, there must be another process in its entry section that changed Head. $p^i$ reached line 39, so it successfully executed the CAS operation at line 38 and set Tail with null. Also, it means that while $p^i$ executed line 38, no other process changed Tail. Therefore, no other process executed line 7 before $p^i$ successfully executed line 38. Let $q^j$ be the process that executed line 7 immediately after $p^i$ successfully executed line 38 and before $p^i$ executed line 39. So $q^j$ got null to its $pred$ at line 7 and atomically set Tail to point to its node. If $q^j$ got null to its $pred$ at line 7 then it is the first process that completed its doorway after the process that Head points to its node

completed its doorway (by using the FAS operation). Thus, $q^j$ is the successor of the process that Head points to its node. $q^j$ continued to line 9 because it saw at line 8 that its $pred$ is null. At line 9, $q^j$ set Head to its node and so it advanced the previous Head to its successor. $p^i$ failed to execute the CAS operation at line 39.

3. $p^i$ got false in all the conditions (lines 38, 40, and 44). So at the time $p^i$ tried to execute the CAS operation at line 38, Tail and Head pointed to different nodes, which means Head's node had a successor other than null. At the time $p^i$ tried to execute line 40, the Head's node's successor hasn't executed line 11 yet to set Head's node's $next$ to point to its successor's node and let the other processes know it is its successor. And if $p^i$ got false at line 44, then the CAS operation at this line succeeded and changed the $active$ of the Head's node to NO. At the time $p^i$ executed line 44, no other process executed lines 15 and 19 that could have changed the $active$ of the Head's node. $p^i$ completed its exit section without changing Head, but the successor of the Head's node would get to line 15 or 19, depends if it requests the same session as its predecessor or not, and the CAS operation at these lines would fail because $p^i$ has already changed the $active$ of the Head's node to NO. If it would get to line 15, then it would succeed with the condition at this line and continue to line 16. If it would get to line 19, then it would fail with the condition at this line and continue to line 22. At these lines (16 and 22), it would set Head to point to its own node which is the Head's node's successor. Therefore, it would advance Head by exactly one step for $p^i$ as a result of $p^i$ successfully executed the CAS operation at line 44.

In all the cases above, $p^i$ completes its exit section and either advances Head or lets another process know it should advance Head by exactly one step. We proved that there is no valid scenario that $p^i$ completed its exit section without causing Head to be advanced by exactly one step. $\square$

**Lemma 3.** *Assume there are only two processes, $p$ and $q$. $p^i$ is the predecessor of $q^j$, and they request different sessions. Then, $p^i$ causes Head to be advanced before $q^j$ is enabled.*

*Proof.* Assume on the contrary that $q^j$ becomes enabled before $p^i$ causes Head to be advanced. $p^i$ is the predecessor of $q^j$, so $q^j$ gets $p^i$'s node as its $pred$ at line 7 and it must continue to line 12. $p^i$ and $q^j$ request different sessions, then $q^j$ must execute line 19. There are two cases:

1. $q^j$ successfully executes the CAS operation at line 19. Therefore, it continues to line 20 and spins on its $go$ until another process sets true to $q^j$'s $go$. But we assumed there are only two processes, $p$ and $q$, so $p$ must change $q^j$'s $go$ to true to let $q^j$ becomes enabled. $p^i$ and $q^j$ request different sessions which means $p^i$ cannot change $q^j$'s $go$ to true at line 31 (because of line 29), so it must change $q^j$'s $go$ to true in its exit section. But to be able to change $q^j$'s $go$ to true, Head must point to $q^j$'s node. According to lemma 2 and assuming we have only two processes, Head must point to $p^i$'s node. $p^i$ must change $q^j$'s $go$ to true in the exit section, so $p^i$ must advance Head by one step to let $q^j$ know it can be enabled, which contradicts the assumption that $q^j$ becomes enabled before $p^i$ advances Head.

2. $q^j$ executes line 19 but the CAS operation at this line fails. Therefore, $q^j$ continues to line 22 and becomes enabled. But if the CAS operation at line 19 fails, then $p^i$'s $active$ is not YES at this time. $p^i$ sets its $active$ to YES

in its doorway. This can be changed either by its successor $q^j$ at line 19 (or 15 if they request the same session) if the CAS operation succeeds, but $q^j$ fails with the CAS operation at this line, or by a process in its exit section at line 44. According to lemma 2 and the assumption that there are only two processes, $p^i$ must reach the exit section and change Head's node's $active$, that is $p^i$'s node's $active$ in this case, to be NO at line 44. But by doing so, $p^i$ lets another process ($q^j$ in this case) know that it should advance Head by one step for $p^i$, in contradiction to the assumption that $q^j$ becomes enabled before $p^i$ causes Head to be advanced.

We see that under the lemma's assumptions, there is no case that $q^j$ can become enabled before $p^i$ causes Head to be advanced. $\square$

**Convention.** *For simplicity, for the rest of this proof, we would consider the same process* $p$ *that appears in two different iterations,* $p^i$ *and* $p^j$ *where* $i \neq j$*, as two different processes. Such that whenever we write "process" we mean a process in a specific iteration.*

The following notions are used in the following definition:

- A group is closed if it is the maximal group s.t. it includes exactly one process that does not have a predecessor in this group. (Thus, every other process has a predecessor in this group.)
- $S_x$ is the maximal set of groups s.t. each group is a closed group of processes that request session x.
- Let $G_1$ and $G_2$ be disjoint groups of processes. $G_2$ follows $G_1$ if a process in $G_2$ follows a process in $G1$.

**Definition 1.** *$S_x^k$ is a group in $S_x$ that follows $k-1$ other groups in $S_x$. A process is the **first** in $S_x^k$ if and only if its predecessor requests session $y \neq x$. A process is the **last** in $S_x^k$ if and only if its successor requests session $z \neq x$. Then:*

1. *$h$**-th process in group**: A process $p^i$ is the $h$-th process in a group $S_x^k$ if and only if $p^i$ follows $h-1$ processes that are in $S_x^k$. For simplicity, we would consider $p^i$'s node as the $h$-th node in $S_x^k$.*

2. ***Group follows**: A group $S_y^t$ follows a group $S_x^k$ if and only if a process $q^j \in S_y^t$ follows a process $p^i \in S_x^k$.*

3. ***Group predecessor**: A group $S_x^k$ is the predecessor of a group $S_y^t$ (s.t. $x \neq y$) if and only if the last process $p^i \in S_x^k$ is the predecessor of the first process $q^j \in S_y^t$.*

4. ***Group successor**: A group $S_y^t$ is the successor of a group $S_x^k$ (s.t. $x \neq y$) if and only if the first process $q^j \in S_y^t$ is the successor of the last process $p^i \in S_x^k$.*

5. ***Group enters CS**: A group $S_x^k$ enters its critical section if and only if at least one of its processes is enabled.*

6. ***Group waits**: A group $S_x^k$ waits if and only if all its processes are not enabled.*

7. ***Group completes CS**: A group $S_x^k$ completes its critical section if and only if all its processes complete their critical sections.*

**Lemma 4.** *Assume $p^i \in S_x^k$ and $q^j \in S_y^t$. If $S_x^k$ is the predecessor of $S_y^t$, then $p^i$ and $q^j$ request different sessions.*

*Proof.* Assume by contradiction that $S_x^k$ is the predecessor of $S_y^t$ but $p^i$ and $q^j$ request the same session, $s$. Let $a$ be the last process in $S_x^k$ and $b$ be the first process in $S_y^t$. Then, by definition, $a$ is the predecessor of $b$. $a$ is in the same group as $p^i$ so it requests the same session as $p^i$, which is $s$. $b$ is in the same group as $q^j$ so it requests the same session as $q^j$, which is also $s$. That means $a$ and $b$ request the same session - $s$. According to definition 1, $a$ and $b$ are in the same group, which means that $S_x^k = S_y^t$. Therefore, $S_x^k$ is not the predecessor of $S_y^t$ in contradiction to the assumption. □

**Lemma 5.** *Assume Head points to $q^j$'s node. Then, if $p^i$ causes Head to be advanced, $p^i$ and $q^j$ are in the same group.*

*Proof.* We prove it by induction on the number of processes that run the algorithm. We look at two base cases - $n = 1$ and $n = 2$. In the base case that $n = 1$ there is only one process, so $p^i = q^j$, that means $q^j$'s node is $p^i$'s node and so they are in the same group. In the other base case, we assume $p^i \neq q^j$. Assume on the contrary that $p^i$ and $q^j$ are not in the same group, which means that they request different sessions. There are two options:

1. $p^i$ is the predecessor of $q^j$. According to lemma 3, $p^i$ causes Head to be advanced before $q^j$ enters its CS. By lemma 2, we know that $p^i$ can cause Head to be advanced by exactly one step, so $p^i$ can only cause Head to be advanced from $p^i$'s node and can't cause Head to be advanced from $q^j$'s node, which contradicts the lemma assumption.

2. $p^i$ is the successor of $q^j$. According to lemma 3, $q^j$ causes Head to be advanced before $p^i$ enters its CS. By lemma 2, we know that $q^j$ can cause Head to be advanced by exactly one step, so $q^j$ causes Head to be advanced from

$q^j$'s node. Therefore, $p^i$ can't cause Head to be advanced from $q^j$'s node, which contradicts the lemma assumption.

Next, we assume that the lemma holds for $n$ processes and prove that it also holds for $n+1$ processes. Let $p_1, p_2, ..., p_n$ be the processes that the lemma holds for, which means that if those processes cause Head to be advanced from another process' node, then they request the same session and are in the same group. Let $p_{n+1}$ be the $(n+1)$-th process that we want to prove the lemma also holds for.

Note, in this lemma's proof, from now on, for readability, any time we say a process advances Head, it means that the process causes Head to be advanced.

We define $S$ as a function that gets a process as an input and returns the group that this process belongs to, s.t. $S(p^i)$ is the group that contains process $p^i$.

Assume $p_{n+1}$ advanced Head from $p_k$'s node, $1 \leq k \leq n+1$. Then, we want to show that $p_{n+1}$ is in the same group as $p_k$. If $k = n+1$ then it immediately follows that the lemma holds, so we assume $k \neq n+1 \Rightarrow 1 \leq k \leq n$. We have two options:

1. $p_k$ advances Head from a node of a process that $p_k$ follows, say without loss of generality that it is $p_1$. So by the induction assumption $p_k$ and $p_1$ are in the same group, $S(p_k) = S(p_1)$. Now, again without loss of generality, we assume $p_1$ advances Head from $p_2$'s node, $p_2$ advances Head from $p_3$'s node and so on until $p_{k-2}$ advances Head from $p_{k-1}$'s node. By the induction assumption, all of these processes are in the same group, $S(p_1) = S(p_2) = S(p_3) = ... = S(p_{k-2}) = S(p_{k-1})$. Then, $p_{k-1}$ must advance some process that follows $p_k$. Assume, without loss of generality, $p_{k-1}$ advances Head from $p_{k+1}$'s node and $p_{k+1}$ advances Head from $p_{k+2}$'s node, $p_{k+2}$ advances Head from $p_{k+3}$'s node and so on until $p_{n-1}$ advances Head from $p_n$'s node

and we have only left $p_n$ that advances Head from $p_{n+1}$'s node. By the induction assumption, all of these processes are in the same group, $S(p_{k-1}) = S(p_{k+1}) = S(p_{k+2}) = S(p_{k+3}) = ... = S(p_{n-1}) = S(p_n) = S(p_{n+1})$. Therefore, we get $S(p_k) = S(p_1) = S(p_2) = S(p_3) = ... = S(p_{k-2}) = S(p_{k-1}) = S(p_{k+1}) = S(p_{k+2}) = S(p_{k+3}) = ... = S(p_{n-1}) = S(p_n) = S(p_{n+1}) \Rightarrow S(p_k) = S(p_{n+1})$ and $p_{n+1}$ and $p_k$ are in the same group.

2. $p_k$ advances Head from a process' node that follows $p_k$, say $p_i$, s.t. $p_i$ follows $p_k$ and $p_{n+1}$ follows $p_i$. So by the induction assumption $p_k$ and $p_i$ are in the same group, $S(p_k) = S(p_i)$. Now, without loss of generality, we assume $p_i$ advances Head from $p_1$'s node, $p_1$ advances Head from $p_2$'s node, $p_2$ advances Head from $p_3$'s node and so on until $p_{k-2}$ advances Head from $p_{k-1}$'s node. By the induction assumption, all of these processes are in the same group, $S(p_i) = S(p_1) = S(p_2) = S(p_3) = ... = S(p_{k-2}) = S(p_{k-1})$. Then, $p_{k-1}$ must advance other process that follows $p_k$. Assume, without loss of generality, $p_{k-1}$ advances Head from $p_{k+1}$'s node and $p_{k+1}$ advances Head from $p_{k+2}$'s node, $p_{k+2}$ advances Head from $p_{k+3}$'s node and so on until $p_{i-2}$ advances Head from $p_{i-1}$'s node. By the induction assumption, all of these processes are in the same group, $S(p_{k-1}) = S(p_{k+1}) = S(p_{k+2}) = S(p_{k+3}) = ... = S(p_{i-2}) = S(p_{i-1})$. Then, $p_{i-1}$ must advance other process that follows $p_i$. Again, assume without loss of generality that $p_{i-1}$ advances Head from $p_{i+1}$'s node and $p_{i+1}$ advances Head from $p_{i+2}$'s node, $p_{i+2}$ advances Head from $p_{i+3}$'s node and so on until $p_{n-1}$ advances Head from $p_n$'s node, and we only have $p_n$ left that advances Head from $p_{n+1}$'s node. By the induction assumption, all of these processes are in the same group, $S(p_{i-1}) = S(p_{i+1}) = S(p_{i+2}) = S(p_{i+3}) = ... = S(p_{n-1}) = S(p_n) =$

$S(p_{n+1})$. Therefore, we get $S(p_k) = S(p_i) = S(p_1) = S(p_2) = S(p_3) = ... = S(p_{k-2}) = S(p_{k-1}) = S(p_{k+1}) = S(p_{k+2}) = S(p_{k+3}) = ... = S(p_{i-2}) = S(p_{i-1}) = S(p_{i+1}) = S(p_{i+2}) = S(p_{i+3}) = ... = S(p_{n-1}) = S(p_n) = S(p_{n+1}) \Rightarrow S(p_k) = S(p_{n+1})$ and $p_{n+1}$ and $p_k$ are in the same group.

□

**Lemma 6.** *For every process that executes line 9, either its node is the first node that has been added to Q, or there is another process that has already successfully executed line 38.*

*Proof.* Let $p^i$ be the process that executes line 9, so $p^i$ gets true in the condition at line 8. That means $pred_p$ is null. $pred_p$ is being set at line 7 by atomically Fetch-And-Set (FAS) operation that sets $pred_p$ with the last value of Tail. If $p^i$'s node is the first node that has been added to Q, then Tail is null (Tail is initialized to null at the very beginning of the algorithm) and the lemma holds.

Now, assume $p^i$'s node is not the first node that has been added to Q. Therefore, some process changed Tail and set its value with null. Tail can be set at line 7 in the entry code and at line 38 in the exit code. Each process that starts the algorithm executes line 7 in its doorway, but at this line, the process sets Tail with its node which can't be null. So the only line Tail could be changed back to null is line 38 in the exit code if the CAS operation at this line succeeds. That means there must be another process in its exit section to execute line 38 successfully and the lemma holds. □

**Lemma 7.** *For every process that executes line 16 or line 22, there is another process that has already successfully executed the CAS operation at line 44.*

*Proof.* Assume on the contrary that process $p^i$ executes line 16 or line 22 but no process completes its CS and could execute line 44 in its exit section. If $p^i$ executes line 16 or line 22, then it also executes line 15 or line 19 respectively. But to get to lines 16 or 22, $p^i$ must fail with the CAS operation at lines 15 or 19 respectively. That means $p^i$'s pred's $active$ is not YES. Each process sets its $active$ to YES in its doorway and does not change it in its entry section anymore. This $active$ can be changed by its successor at lines 15 and 19 if the CAS operation succeeds or by some process in its exit section at line 44 if the CAS operation succeeds. $p^i$ fails to execute the CAS operation at line 15 or 19, so there is another process that changed $p^i$'s predecessor's $active$ at line 44 in its exit section. contradiction. □

**Lemma 8.** *Let $p^i$'s node be the $h$-th node in group $S_x^k$. If Head passes over $p^i$'s node, then at least $h$ processes in $S_x^k$ completed their critical sections.*

*Proof.* Assume Head passed over $p^i$'s node, so Head should be advanced by at least $h$ steps. Let $g$ be the number of processes in $S_x^k$ that have already caused Head to be advanced. According to lemma 2, those $g$ processes caused Head to be advanced in a total of $g$ steps. A process causes Head to be advanced by either advance Head in its exit section or let another process know it should advance Head in its entry section. Let $g_1$ be the number of processes from the $g$ processes that advanced Head in their exit sections, and $g_2$ be the number of processes from the $g$ processes that let other processes know that they should advance Head in their entry sections, s.t. $g = g_1 + g_2$.

Head was advanced by $g_1$ steps in the exit code. Head should be advanced by additional $h - g_1$ steps to pass over $p^i$'s node. We denoted $g_1$ as the number of processes in $S_x^k$ that completed their CSs and advanced Head in their exit sections, so no other process from $S_x^k$ can advance Head in its exit section other than those

$g_1$ processes. According to lemma 5, no other process from any group can cause Head to be advanced other than the $g$ processes from $S_x^k$. That means Head must be advanced by $h - g_1$ steps in the entry code.

Head can be changed in the entry code at lines 9, 16 and 22. Let $f = f_1 + f_2$ be the number of processes that execute lines 9, 16 and 22, s.t. $f_1$ is the number of processes that execute line 9 and $f_2$ is the number of processes that execute either line 16 or line 22.

- Line 9 - If a process $q^j$ changes Head at line 9, then by lemma 6, $q^j$'s node is either the first node that has been added to Q or there is another process that successfully executed line 38, which means it also completed its CS. If $q^j$'s node is the first node that has been added to Q, then it didn't advance Head because Head has not pointed to any node yet. So assume all nodes of all the $f_1$ processes that executed this line are not the first nodes that have been added to Q. By lemma 1, each process has a different predecessor. Therefore, for each process that executes line 38 and sets Tail with null, there is at most one process that can see Tail as null, and according to lemma 6, there must be $f_1$ processes that completed their CS and executed line 38.

- Lines 16 and 22 - If Head was changed at line 16 or line 22 by $f_2$ processes, then by lemmas 7 and 1, there must be $f_2$ processes that successfully executed the CAS operation at line 44 and so they also completed their CS.

We get that Head was advanced by exactly $f$ steps in the entry code, as a result of exactly $f$ processes that completed their CSs and have already executed the line that asks another process to advance Head in its entry section. Therefore, $f = g_2 = h - g_1 \Leftrightarrow f + g_1 = g_2 + g_1 = h \Rightarrow h = g_1 + g_2 = g \Rightarrow h = g$. $g$ is the number of processes in $S_x^k$ that caused Head to be advanced. Therefore, it also indicates that at least $g$ processes completed their CSs, which means that at least $h$ processes in $S_x^k$ completed their CSs if Head passed over $p^i$'s node and it is the $h$-th node in $S_x^k$. □

**Lemma 9.** *If Head passes over the last node in $S_x^k$, then all the processes in $S_x^k$ completed their critical sections.*

*Proof.* Let $h$ be the number of processes in $S_x^k$. Then, the node of the last process in $S_x^k$, say $p^i$, is the $h$-th node in $S_x^k$. According to lemma 8, if Head passes over $p^i$'s node, then $h$ processes in $S_x^k$ completed their CSs. Thus, all the processes in $S_x^k$ completed their CSs. □

**Lemma 10.** *Assume $S_x^k$ is the predecessor of $S_y^t$. Then, if $S_y^t$ completed its critical section, $S_x^k$ also completed its critical section.*

*Proof.* Suppose $S_y^t$ completed its CS. $S_x^k$ is the predecessor of $S_y^t$. Then, the last process in $S_x^k$, say $p^i$, is the predecessor of the first process in $S_y^t$, say $q^j$. Assume by contradiction that $S_x^k$ hasn't completed its CS yet. That means there is at least one process in $S_x^k$ that hasn't completed its CS yet. According to lemma 9, Head hasn't passed over the last node in $S_x^k$ yet, which is $p^i$'s node.

By inspecting the algorithm's code, $p^i$ sets its node's $active$ to YES in its doorway and doesn't change its $active$ in its entry section anymore. We know Head hasn't passed over $p^i$'s node yet, so $p^i$'s node's $active$ couldn't be changed to NO (line 44).

$S_y^t$ completed its CS so all the processes in this group completed their CSs, including $q^j$. $q^j$ has already entered its CS. Head hasn't passed over $p^i$'s node yet, so $q^j$'s $pred$, which is $p^i$'s node, is not null. Thus, $q^j$ executed line 12 and continued to line 19, because $p^i$ and $q^j$ request different sessions by corollary 4. We have two options:

- $q^j$ gets true for the condition at line 19. Then, it continues to line 20 and waits on its $go$, which can be changed only in the exit code (lines 43 and 47) when Head points to $q^j$'s node. But if Head points to $q^j$'s node then Head passed over $q^j$'s predecessor's node, which is $p^i$'s node, but we already proved that Head hasn't passed over $p^i$'s node.

- $q^j$ gets false for the condition at line 19. Then, it continues to line 22, sets Head to its node, and continues to its CS. But to get false at line 19, the $active$ of $q^j$'s predecessor's node, $p^i$'s node, had to be different than YES. We've already proved that $p^i$'s node's $active$ couldn't be changed in the exit code, so it should have changed in the entry code. By lemma 1, each process has a different predecessor, which means $p^i$ is the predecessor only for $q^j$. So the only process that can change $p^i$'s node's $active$ in its entry section is $q^j$. In the entry code, the $active$ of the predecessor's node can be changed at line 15, if they request the same session, or at line 19 if they request different sessions. Therefore, the $active$ of $p^i$'s node couldn't change at all.

Then, in both cases, $q^j$ waits at line 20 at least until all the processes in $S_x^k$ complete their CSs according to lemma 9. Therefore, $q^j$ can't complete its CS as long as $S_x^k$ hasn't completed its CS yet. Therefore, $S_y^t$ hasn't completed its CS yet and we get a contradiction. □

**Lemma 11.** *Assume $S_y^t$ follows $S_x^k$. If $S_y^t$ completed its critical section, then $S_x^k$ also completed its critical section.*

*Proof.* Let $S$ be a set of all the groups that $S^t_y$ follows. We prove this lemma by induction on the number of groups that $S^t_y$ follows, say $|S| = A$. If $|S| = A = 0$, then $S^t_y$ doesn't follow any group and the lemma is vacuously true. In the base case $A = 1$, $S^t_y$ follows only one group, $S^k_x$. That means $S^k_x$ is the predecessor of $S^t_y$ and according to lemma 10, $S^k_x$ completed its CS too, and so the lemma holds.

Next, assume the lemma holds for $A$ groups and we show that it also holds for $|S| = A + 1$ groups. Let's assume on the contrary that it doesn't hold for $A + 1$ groups, which means there is a group in $S$ that hasn't completed its CS yet, say $S^k_x$. This group must have a successor group from $S$ or be the predecessor group of $S^t_y$. If it has a successor group from $S$, then its successor group has already completed its CS by the induction assumption and according to lemma 10, $S^k_x$ also completed its CS. If $S^k_x$ is the predecessor of $S^t_y$ then again according to lemma 10, $S^k_x$ also completed the CS. contradiction. □

**Lemma 12.** *Assume $q^j$ follows $p^i$ and $q^j$ executes line 19. Then, if the CAS operation fails, it must be that $p^i$ has already completed its critical section.*

*Proof.* $q^j$ executes line 19. If it gets to line 19, then it requests a different session than its predecessor. Assume by contradiction that $q^j$ gets false at line 19 while $p^i$ hasn't completed its CS yet. If $q^j$ gets false at line 19, then it fails to execute the CAS operation at this line, so the $active$ for its predecessor's node is not YES. In its doorway, each process sets its $active$ to YES and doesn't change it in its entry section (by code inspection). By lemma 1 and inspecting the algorithm's code, the only process that can change the $active$ of $q^j$'s predecessor in its entry section is its successor, $q^j$. $q^j$ can change its predecessor's $active$ at lines 15 and 19 if the CAS operation succeeds, but it can succeed only if $active$ is not YES. So another process has to change $q^j$'s predecessor's $active$ in its exit section at line 44. It can be changed at line 44 if the CAS operation at this line succeeds. If a process executes line 44 on $q^j$'s predecessor's node, then it also makes Head pass over this node. $q^j$ requests a different session than its predecessor, which means that its node is the first node in its group and its predecessor's node is the last node in a different group, say $S_x^k$. According to lemma 9, all the processes in $S_x^k$ completed their CSs. Therefore, by lemma 11, all the groups that $S_x^k$ follows also completed their CSs. We get that all the groups that $q^j$ follows completed their CSs, so all the processes that $q^j$ follows completed their CSs, including $p^i$, in contradiction to the assumption. □

**Lemma 13.** *If $p^i$ is at line 26, then it is enabled.*

*Proof.* By definition, a process $p^i$ is enabled if it can enter its CS within a bounded number of its own steps. If $p^i$ is at line 26, then it changes its $status$ to ENABLED and continues to line 27. At this line, $p^i$ sets a local variable $next_p$ with the value in $node_p \rightarrow next$ and continues to line 28 to check if its next is null or it already has a successor. If $next_p$ is null, then $p^i$ continues to its CS. Otherwise, $next_p$ is not null and $p^i$ has a successor. $p^i$ continues to line 29 and checks if it requests the same session as its successor. If $p^i$ and its successor request different sessions, then $p^i$ continues to its CS. If they request the same session, $p^i$ continues to line 30 and checks using the CAS operation if its successor needs help to become enabled. $p^i$ tries to execute the CAS operation and change its $status$ to TRY_HELP to let its successor know that it is trying to help. If $p^i$ fails to execute the CAS operation, then it does not need to help its successor and so $p^i$ continues to its CS. And if $p^i$ succeeds to execute the CAS operation, then it continues to line 31, sets its successor's $go$ to true, and continues to its CS. We can see that no matter how $p^i$

continues from line 26, no other process can prevent $p^i$ from entering its CS and $p^i$ can enter its CS within a bounded number of its own steps. $\square$

**Lemma 14.** *Assume $q^j$ follows $p^i$. If $p^i$ is not enabled then $q^j$ is not enabled.*

*Proof.* Let $P$ be a set of all the processes that follow $p^i$. We prove the lemma by induction on the number of processes that follow $p^i$, say $|P| = k$. If $k = 0$, then no process follows $p^i$ and the lemma is vacuously true, so we assume $k > 0$. In the base case, $k = 1$, only one process follows $p^i$, say $q^j$. $p^i$ is not enabled, then by lemma 13, it can't reach line 26 and set its $status$ to ENABLED. Also, $p^i$ sets its $active$ to YES in its doorway and doesn't change it in its entry section anymore. $q^j$ reaches line 12 as its $pred$ is not null ($p^i$'s node). If $q^j$ requests the same session as $p^i$, then it continues to line 13, sees its predecessor is not enabled, and spins on its $go$ at line 14. Otherwise, $q^j$ continues to line 19, sees that its predecessor's node's $active$ is still YES, and spins on its $go$ at line 20. $q^j$ spins on its $go$ until another process changes its value to true. According to the base case, $p^i$ and $q^j$ are the only active processes. Therefore, $p^i$ is the only process that can release $q^j$ and the lemma holds.

We assume the lemma holds for $k$ processes and prove that it also holds for $|P| = k + 1$ processes. Assume by contradiction that the lemma does not hold for $|P| = k + 1$ processes, then there is a process in $P$, say $q^j$, that is enabled to enter its CS while $p^i$ is not enabled.

Suppose $p^i$'s node is the $h$-th node in its group and $q^j$ has not entered its CS yet. According to lemma 8, at least $h$ processes completed their CSs. But if $p^i$'s node is the $h$-th node in its group, then $p^i$ is the $h$-th process in its group. That means, $p^i$ follows $h - 1$ processes from its group. Even if all these processes complete their exit sections, according to lemma 2, they cause Head to be advance by exactly

$h-1$ steps, which leads Head to point to $p^i$'s node. By the induction assumption, $p^i$ and all the processes that follow $p^i$ (except for $q^j$ which has not entered its CS yet) are not enabled, so they can't complete their exit sections. Therefore, Head can't pass over $p^i$'s node at this moment.

By inspecting the algorithm, there are four options:

1. $q^j$'s $pred$ is null. This case is invalid because $q^j \in P$ which means it follows $p^i$, then it must have a predecessor.

2. $q^j$'s predecessor's $status$ is ENABLED. $q^j$'s predecessor is either $p^i$ or another process from $P$, which means $q^j$'s predecessor is not enabled (by the induction assumption). Each process sets its $status$ to WAIT in its doorway (line 5) and to ENABLED later in its entry section at line 26. According to lemma 13, $q^j$'s predecessor can't set its $status$ to ENABLED and $q^j$ must continue to line 14 and spin on its $go$.

3. $q^j$'s $go$ is true. $q^j$ sets its $go$ value to false in its doorway at line 3. A process can set true in $q^j$'s $go$ at line 31 if it is $q^j$'s predecessor or at lines 43 or 47 in the exit code. $q^j$'s predecessor is not enabled by the induction assumption, so it can't reach line 31. Therefore, another process must be in its exit section and execute either line 43 or line 47 on $q^j$'s node, which means Head points to $q^j$'s node at this moment. But we proved that Head can't pass over $p^i$'s node, then it can't reach $q^j$'s node. Therefore, no process can set $q^j$'s $go$ to true.

4. $q^j$'s predecessor's $active$ is not YES. Each process sets its $active$ to YES in its doorway (line 6) and does not change this value in its entry section anymore. The lines that can change a process' $active$ from YES are lines

15 and 19 in the entry code that are being executed by its successor, or line 44 in the exit code. We assume $q^j$'s predecessor's $active$ is not YES, then the CAS operations at lines 15 and 19 fail. Then, there is another process that changes $q^j$'s predecessor's $active$ in its exit section at line 44. To do so, Head must point to $q^j$'s predecessor's node. $q^j$'s predecessor is either $p^i$ or another process in $P$ that follows $p^i$. Either way, we proved Head can't pass over $p^i$'s node so it also can't pass over $q^j$'s predecessor's node. Therefore, no process can change $q^j$'s predecessor's $active$ from YES.

We get that as long as $p^i$ and $t$ processes that follow $p^i$ are not enabled, $q^j$ that also follows $p^i$ is not enabled in contradiction to the assumption that $q^j$ is enabled. □

**Theorem 1** (Strong-FCFS). *If process $p^i$ completes its doorway before process $q^j$ (enters or) completes its doorway and the two processes request different sessions, then $q^j$ does not enter its critical section before $p^i$ does.*

*Proof.* Suppose $p^i \in S^k_x$. $p^i$ completes its doorway before $q^j$ completes its doorway, which means $q^j$ follows $p^i$ and so $q^j$'s group follows $S^k_x$. If $p^i$ is not enabled then by lemma 14 $q^j$ is not enabled too. Therefore, we assume $p^i$ is enabled but has not entered its CS yet.

Denote the successor of $S^k_x$ as $S^t_y$ s.t. $x \neq y$ (by lemma 4) but $y$ may be equal to $q^j$'s session, which means $q^j$'s group may follow $S^t_y$ or be equal to this group. Let $r^\alpha_1$ be the last process in $S^k_x$ and $r^\beta_2$ be the first process in $S^t_y$, then $r^\alpha_1$ is the predecessor of $r^\beta_2$. If $r^\alpha_1$ is not enabled then by lemma 14 $q^j$ is not enabled too. So suppose $r^\alpha_1$ is enabled.

According to lemma 4, $r^\alpha_1$ and $r^\beta_2$ request different sessions, then $r^\beta_2$ cannot reach line 13 (as a result of line 12), see that its predecessor's $status$ is ENABLED and become enabled. Also, $r_2^\beta$'s predecessor, which is $r_1^\alpha$, can't reach line 31 (as a result of line 29), set $r_2^\beta$'s $go$ to true and help $r_2^\beta$ become enabled. Then, by inspecting the algorithm's code, $r_2^\beta$'s $go$ can be set to true only by another process in its exit section (lines 43 and 47). Therefore, $r_2^\beta$ can become enabled in two options:

1. $r_2^\beta$ sees its $go$ is true at line 20.

2. $r_2^\beta$ sees its predecessor's node is not active.

$r_1^\alpha$'s node is the last node in $S_x^k$ and $p^i$ has not entered its CS yet, then according to lemma 9, Head cannot pass over $r_1^\alpha$'s node. Therefore, $r_1^\alpha$'s $active$ can't be changed to $NO$ and Head can't reach $r_2^\beta$'s node, so $r_2^\beta$'s $go$ can't be changed to true. Then, the only option for $r_2^\beta$ to become enabled is to see its predecessor's $active$, $r_1^\alpha$'s $active$, is not YES (at line 19). But we've already proved that $r_1^\alpha$'s $active$ can't be changed in the exit code to NO as long as $p^i$ hasn't entered its CS. Each process sets its $active$ to YES in its doorway (line 6) and doesn't change it in its doorway anymore. Therefore, $r_1^\alpha$'s $active$ can be changed only by its successor, $r_2^\beta$, at lines 15 and 19. But $r_2^\beta$ requests a different session than its predecessor so it can't reach line 15 (by line 12). At line 19, $r_2^\beta$ can set its predecessor's $active$ to $YES$ if the CAS operation at this line succeeds, but then it must continue to line 20 and spin on its $go$. We get that if $p^i$ has not entered its CS yet, $r_2^\beta$ can't become enabled. According to lemma 14, $q^j$, that follows (or equals to) $r_2^\beta$, can't become enabled too. Therefore, $q^j$ cannot enter its CS before $p^i$ does. □

**Corollary 1.1** (FCFS). *The algorithm satisfies FCFS.*

*Proof.* This follows immediately from the fact that the algorithm satisfies strong-FCFS according to theorem 1. □

**Theorem 2** (FIFE)**.** *If process $p^i$ completes its doorway before process $q^j$ enters its doorway, the two processes request the same session, and $q^j$ enters its critical section before $p^i$ does, then $p^i$ enters its critical section within a bounded number of its own steps.*

*Proof.* Assume by contradiction that $q^j$ is in its CS but $p^i$ is not enabled. $p^i$ completes its doorway before $q^j$ enters its doorway, so $q^j$ follows $p^i$. According to lemma 14, if $p^i$ is not enabled then $q^j$ is not enabled which contradicts the assumption that $q^j$ is in its CS. □

**Theorem 3** (Mutual exclusion)**.** *If two processes are in their critical sections at the same time, then they request the same session.*

*Proof.* Let $p^i$ and $q^j$ request different sessions and assume by contradiction that they are in their CSs at the same time. Assume, without loss of generality, that $p^i$ enters its CS before $q^j$ does. According to theorem 1, the algorithm satisfies Strong-FCFS, then $p^i$ completes its doorway before $q^j$ does, so $p^i$ executes line 7 before $q^j$ does. There are two cases:

1. $q^j$ is the successor of $p^i$.

2. $q^j$ follows $p^i$ but it is not $p^i$'s successor.

According to lemma 14, once we prove that $q^j$ must wait until $p^i$ completes its CS in case 1, it would immediately follow that $q^j$ must wait until $p^i$ completes its CS in case 2. So let's prove case 1. We assume that $q^j$ is in its CS while $p^i$ is in its CS and both request different sessions. Therefore, $q^j$ must execute line 8. According to lemma 9, as long as $p^i$ is in its CS, Head can't pass over $p^i$'s node, which means Head can't reach $q^j$'s node. Then, $q^j$ gets $p^i$'s node at line 7 as its

$pred$ which is not null and continues from line 8 all the way to line 12. We assume that $q^j$ and $p^i$ request different sessions, so $q^j$ continues to line 19. $q^j$ follows $p^i$ and $p^i$ is still in its CS, so by lemma 12, $q^j$ gets true at line 19 and continues to line 20. Then, it waits at line 20 until another process changes its $go$ from false to true. The successor of $q^j$ is $p^i$. $p^i$ couldn't change $q^j$'s $go$ at line 31, because they request different sessions and so it gets false at line 29. The other lines that can change $q^j$'s $go$ are lines 43 and 47 in the exit code. These lines are being executed on Head, so to execute these lines on $q^j$'s node, Head must reach $q^j$'s node. But as we proved before, Head can't reach $q^j$'s node as long as $p^i$ is in its CS. Thus, $q^j$ must wait as long as $p^i$ is in its CS. Therefore, it also must wait in case 2. We get that as long as $p^i$ is in its CS, $q^j$ must wait in contradiction to the assumption that it is in its CS while $p^i$ is in its CS. $\square$

**Definition 2** (Bounded entry)**.** *If a process $p^i$ is in its entry section, while no other process is in its critical section or exit section, then some process can complete its entry section within a bounded number of its own steps.*

**Theorem 4** (Group bounded exit)**.** *If a process $p^i$ is in its exit section, then (1) some process can complete its exit section within a bounded number of its own steps, and (2) $p^i$ eventually completes its exit section.*

*Proof.* Let $P$ be the set of all the active processes that are in their exit sections, then

1. some process in $P$ can complete its exit section within a bounded number of its own steps, and

2. all the other processes in $P$ eventually complete their exit sections.

Notice that the exit code contains a *mutual exclusion* (ME) algorithm, other than that it contains one more line (line 50) that each process can execute without depending on another process. The lines within the ME CS are wait-free, without any loops or await operations, and can be executed by the process without depending on another process. Then, to guarantee that our GME algorithm satisfies bounded exit, the mutual exclusion used in the exit code (lines 36 and 49) must satisfy three properties, (1) starvation-freedom, (2) bounded exit, and (3) bounded entry. While the famous MCS lock [25] does not satisfy the bounded exit property, there are variants of it, like the mutual exclusion algorithms in [11, 17], that satisfy all the above three properties.

Let's use, for example, one of the algorithms that are described in [11, 17]. Let $P_A$ be the set of all the active processes in this ME algorithm, s.t. $P_A \subseteq P$. Then,

1. Suppose a process $p^i \in P_A$ is in its ME exit section, then according to the property bounded exit which this ME algorithm satisfies, $p^i$ can complete the ME algorithm within a bounded number of its own steps.

2. Assume all the processes in $P_A$ are in their entry sections, then according to the bounded entry property, which this ME algorithm satisfies, some process in $P_A$, say $p^i$, can complete its ME entry section within a bounded number of its own steps and enters its ME CS. We have already proved at the beginning of this theorem's proof, that a process can complete the ME CS within a bounded number of its own steps, so $p^i$ can complete its ME CS and continues to its ME exit section. Then, according to the previous section, $p^i$ can complete the ME algorithm within a bounded number of its own steps.

Therefore, there is always a process that can complete the ME algorithm within a bounded number of its own steps. And according to the starvation freedom property which this ME algorithm also satisfies, all the other processes in $P_A$ eventually enter their ME CSs and complete the ME algorithm and so the theorem is correct. □

**Theorem 5** (Deadlock freedom)**.** *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

*Proof.* Assume by contradiction that some set of processes $P$ are in their entry sections and none of them can ever access its CS, which means all of them are not enabled and never will be. Let $q^j$ be the first process in $P$ that completed its doorway. Thus, all the other processes in $P$ follow $q^j$. The fact that $q^j$ is not enabled and never will be, means that $q^j$ must spin either at line 14 or line 20 since all the other lines in the entry code do not contain any loop, are wait-free, and can be executed in a constant number of $q^j$'s steps. Any other execution path would lead $q^j$ to its CS and contradicts the assumption. From here it follows that Tail was not null when $q^j$ completed its doorway. Therefore, there exists another process, say $p^i$, s.t. $p^i \notin P$ and $p^i$ is the predecessor of $q^j$. Let $P'$ be a set of all the processes that $q^j$ follows, including $p^i$. That means all the processes in $P'$ completed their doorway sections before $q^j$ did. We denoted $q^j$ as the first process in $P$ that completed its doorway, s.t. $q^j$ is the first process that completed its doorway and would never enter its CS, so all the processes in $P'$ eventually are enabled and enter their CSs. Now, the processes in $P'$ complete their CSs and enter their exit sections. According to theorem 4, the algorithm satisfies *group bounded exit*. Therefore, each process in $P'$ eventually completes its exit section, and by lemma 2, causes Head to be advanced by exactly one step. Suppose one process in $P'$ has not yet completed its exit section and caused Head to be advanced, say $p^i$ without loss of generality. Then, all the other processes in $P'$ caused Head to be advanced by exactly $|P'| - 1$ steps, which caused Head to point to $p^i$'s node. Therefore, when $p^i$ completes its exit section, it causes Head to be advanced from $p^i$'s node to $q^j$'s node. Here we have three options:

1. $p^i$ executes line 38 before $q^j$ executes line 7. Therefore, $p^i$ successfully executes line 38 and sets Tail to null. Then, $q^j$ executes line 7 and sets its $pred$ to null, so it gets true at line 8 and becomes enabled.

2. $p^i$ executes line 38 after $q^j$ executes line 7, but also $p^i$ executes line 40 after $q^j$ executes line 11. Therefore, $p^i$ continues from line 38 to line 40 without changing Tail. At line 40, $p^i$ sees that the Head's $next$ is not null, so $p^i$ advances Head to its $next$, which is $q^j$'s node, and sets $q^j$'s node's $go$ to true. Therefore, $q^j$ can't spin on its $go$ and becomes enabled.

3. $p^i$ executes line 38 after $q^j$ executes line 7, but also $p^i$ executes line 40 before $q^j$ executes line 11. Therefore, $p^i$ continues from line 38 to line 40 without changing Tail. At line 40, $p^i$ sees that the Head's node's $next$ has not been set yet, so it continues to line 44. If $q^j$ executes line 19 after $p^i$ executes line 44, then $q^j$ becomes enabled because the CAS operation at line 44 would end successfully and $q^j$'s node is not active anymore. So we assume that $q^j$ executes line 19 before $p^i$ executes line 44, then $q^j$ waits on its $go$ at line 20, and $p^i$ fails to executes the CAS operation at line 44. Thus, $p^i$ gets true in the condition at line 44 and advances Head to its $next$, which is $q^j$'s node, and sets $q^j$'s node's $go$ to true. Therefore, $q^j$ stops spinning on its $go$ and becomes enabled.

We get that there is no case in which $q^j$ never becomes enabled to enter its CS which contradicts the assumption. □

**Corollary 5.1** (Starvation freedom)**.** *The algorithm satisfies starvation freedom.*

*Proof.* This follows from the fact that the algorithm satisfies both FCFS (corollary 1.1) and deadlock freedom (theorem 5). □

**Lemma 15.** *When no process is active, then Q is empty.*

*Proof.* Assume by contradiction that Q is not empty s.t. Q includes a node, say $a$, when no process is active. That means, either Tail points to $a$ or Head points to $a$ or both of them. At the very beginning of the algorithm, Tail and Head are initialized with null. Therefore, at least one process has already executed the algorithm and completed the algorithm because there is no active process. Let $P$ be the set of all the processes that completed their iterations. All the processes in $P$ completed their exit sections, and according to lemma 2, each of them caused Head to be advanced by exactly one step. Let's check the execution path of the last process in $P$ that acquired the ME lock, say $p^i$, in its exit section. At this time, all the processes in $P$ except for $p^i$ have already caused Head to be advanced by exactly $|P| - 1$ steps. Therefore, at this time, Head pointed to $a$. $a$ is the last node that has been entered to Q at line 7. Therefore, at the time $p^i$ acquired the ME lock, Tail pointed to $a$. $p^i$ acquired the lock at line 36 and continued to line 37 in which it saved Head's current value in its own local variable, which was a pointer to $a$. $p^i$ continued to line 38 and using the CAS operation, it atomically saw both Tail and its local variable pointed to $a$ and set Tail with null. Then, $p^i$ continued to line 39, as the CAS operation at line 38 succeeded, and using another CAS operation, it atomically saw both Head and its local variable pointed to $a$ and set Head with null.

$p^i$ released the ME lock and completed its iteration. Then, after all the processes completed their iterations and while no more active processes, Head and Tail are null. contradiction. $\square$

**Theorem 6** (Strong group concurrent entering)**.** *If a process $p^i$ requests a session $x$, and $p^i$ completes its doorway before any conflicting process starts its doorway, then (1) some process with session $x$ can complete its entry section within a bounded number of its own steps, and (2) $p^i$ eventually completes its entry section, even if other processes do not leave their critical sections.*

*Proof.* According to theorem 1, the algorithm satisfies strong-FCFS, so $p^i$ enters its CS before any conflicting process does. Assume $p^i \in S^k_x$. Let $P$ be a set of all the active processes in $S^k_x$, s.t. all the processes in $P$ request session $x$. Denote $p_1, p_2, p_3, ..., p_t \in P$ s.t. $p_1$ is the predecessor of $p_2$, $p_2$ is the predecessor of $p_3$, and so on. Note, $p^i$ is also active and in $S^k_x$ then $p^i$ is one of these processes.

We prove the theorem by induction on the number of processes in $P$, $|P| = t$. In the base case, $t = 1$: There is only one active process $p_1 = p^i \in P$. if $p^i$ executes line 7 and gets null as its $pred$, then $p^i$ is immediately enabled and the theorem holds. Otherwise, there is a node that is still in Q. Then, this node was in Q before $p^i$ completes its doorway. By the theorem assumption, $p^i$ completes its doorway before any conflicting process enters its doorway, so the node was in Q while there were no active processes, and by lemma 15, it cannot be that a node is in Q before $p^i$ has added its node.

Next, we assume the theorem holds for $t$ processes and shows it also holds for $|P| = t + 1$ processes. We have two cases:

1. The theorem holds for these $t$ processes $p_2, p_3, ..., p_{t+1} \in P$ and we prove

it also holds for $p_1$. $p_1$ is the first active process because $p^i$ is in $P$ and no conflicting process enters its doorway before $p^i$ completes its doorway. So according to lemma 15, $p_1$ gets null to its $pred$ at line 7. Then, it continues to line 9 as it sees its $pred$ is null (line 8), sets Head to its own node and becomes enabled, and the theorem holds for all the $t + 1$ processes in $P$.

2. The theorem holds for $t$ processes in $P$ including $p_1$. Assume, without loss of generality, that the theorem holds for $p_1, p_2, p_3, ..., p_t$ and we prove it also holds for $p_{t+1} \in P$. According to the induction assumption, $p_t$ is either enabled or eventually be enabled. Here we also have two options:

    - $p_t$ is enabled, then if $p_{t+1}$ executes line 13 after $p_t$ executes line 26, $p_{t+1}$ sees $p_t$ is enabled and $p_{t+1}$ becomes enabled too and the theorem holds. Otherwise, $p_{t+1}$ spins on its $go$ at line 14 until $p_t$ sets it to true. $p_t$ is enabled so it reaches line 31 as its successor has already executed line 11, they both requested the same session, and $p_{t+1}$ couldn't change $p_t$'s status at line 13. Therefore, $p_t$ sets $p_{t+1}$'s $go$ to true at line 31 and $p_{t+1}$ becomes enabled and the lemma holds.
    - $p_t$ is not enabled but by the induction assumption it eventually becomes enabled. Then, $p_{t+1}$ sees $p_t$ is not enabled and spins on its $go$ at line 14 until its successor, $p_t$, becomes enabled. $p_t$ eventually becomes enabled and sets $p_{t+1}$'s $go$ to true at line 31 and so $p_{t+1}$ eventually becomes enabled too.

$\square$

**Theorem 7** (Constant RMR complexity)**.** *The RMR complexity of the algorithm is $O(1)$ in both the CC and the DSM models.*

*Proof.* By inspecting the algorithm, it is easy to count steps and see that except the busy-waiting loops at lines 14 and 20 in the entry code and the use of a Mutual Exclusion (ME) lock in the exit code, it takes a constant number of steps for a process to enter its CS and complete its exit section. By using an ME algorithm that also has a constant RMR complexity in both the CC and the DSM models, such as in [11, 17], the RMR complexity of the algorithm's exit code is $O(1)$. Therefore, it is sufficient to prove that for every process $p^i$, $p^i$ performs $O(1)$ RMRs at both lines 14 and 20, because these lines are the only busy-waiting loops in the algorithm. $p^i$ checks at line 12 if it requests the same session as its predecessor. If so, it can only spin at line 14. Otherwise, it can only spin at line 20. $p^i$ spins on $node_p \rightarrow go$ no matter if it spins at line 14 or line 20. We will prove that while the process is executing line 14 or line 20, it performs only a constant number of RMRs in both models:

- **DSM model**: $p^i$ spins on $node_p \rightarrow go$. $node_p$ points to either $Nodes_p[0]$ or $Nodes_p[1]$ as follows from line 1 in the algorithm. Both of them were initialized as local to process $p^i$'s memory. Thus, the algorithm performs $O(1)$ RMRs in the DSM model.

- **CC model**: We prove that in one iteration of a process, there is at most one cache invalidation. Before $p^i$ spins on $node_p \rightarrow go$ either at line 14 or line 20, its value migrates to $p^i$'s local cache, since $p^i$ sets the value to false at line 3. Except for line 3, the value can be updated at lines 31, 43, and 47. All these lines update the value to true. Then, no matter which line is being executed first or being executed at all, the value of $p^i$'s $node_p \rightarrow go$ can be changed from false to true only once. Therefore, when a process executes one of the lines above, $p^i$ that spins at line 14 or line 20, would have a cache invalidation and perform one RMR to read the new value of $node_p \rightarrow go$. Since the new value is necessarily equal to true, $p^i$ stops spinning on $node_p \rightarrow go$ and proceeds to its CS. Therefore, there is only one RMR during the loop execution, and the algorithm has $O(1)$ RMR complexity in the CC model.

□

**Theorem 8.** *The algorithm uses constant space per process and total $O(n)$ shared memory locations.*

*Proof.* By inspecting the algorithm's code, the algorithm uses two shared memory locations Head and Tail, and each process that runs the algorithm uses two nodes. The algorithm also uses an ME lock, so when we use an ME algorithm that also uses constant space per process and total of $O(n)$ shared memory locations such as in [11, 17], our algorithm uses $O(2+n+n) = O(n)$ shared memory locations. □

**Theorem 9.** *The algorithm satisfies (1) mutual exclusion, (2) starvation freedom, (3) group concurrent entering, (4) group bounded exit, (5) FCFS (even strong-FCFS), and (6) FIFE. Furthermore, the algorithm has constant RMR complexity in both the CC and the DSM models, it uses constant space per process and a total of $O(n)$ shared memory locations, and it does not require to assume that the number of processes or the number of sessions are prior known.*

*Proof.* The properties mutual exclusion, starvation freedom, group concurrent entering, group bounded exit, FCFS (strong-FCFS), and FIFE, follow from theorem 3, corollary 5.1, theorem 6, theorem 4, corollary 1.1 (theorem 1), and theorem 2 respectively. According to theorem 7, the algorithm has constant RMR complexity in both the CC and the DSM models, and according to theorem 8, it uses constant space per process and a total of $O(n)$ shared memory locations. The other properties are easily verified by inspecting the code of the algorithm. $\square$

# 5 Discussion

With the wide availability of multi-core systems, synchronization algorithms like GME are becoming more important for programming such systems. In concurrent programming, processes (or threads) are often sharing data structures and databases. The GME problem deals with coordinating access to such shared data structures and shared databases.

We have presented a new GME algorithm that is the first to satisfy several desired properties. Based on our algorithm, it would be interesting to design other GME algorithms, such as abortable GME [16] and recoverable GME [13], which will preserve the properties of our algorithm.

# References

[1] A. Aravind and W.H. Hesselink. Group mutual exclusion by fetch-and-increment. *ACM Trans. Parallel Comput.*, 5(4), 2019.

[2] R. Atreya, N. Mittal, and S. Peri. A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set. *IEEE Transactions on Parallel and Distributed Systems*, 18(10), 2007.

[3] J. Beauquier, S. Cantarell, A. K. Datta, and F. Petit. Group mutual exclusion in tree networks. In *Proc. of the 9th International Conference on Parallel and Distributed Systems*, pages 111–116, 2002.

[4] V. Bhatt and C.C. Huang. Group mutual exclusion in $O(log\ n)$ RMR. In *Proc. 29th ACM Symp. on Principles of Distributed Computing*, pages 45–54, 2010.

[5] G. E. Blelloch, P. Cheng, and P. B. Gibbons. Room synchronization. In *Proc. of the 13th Annual Symposium on Parallel Algorithms and Architectures*, pages 122–133, 2001.

[6] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6:632–646, 1984.

[7] P.L. Courtois, F. Heyman, and D.L Parnas. Concurrent control with Readers and Writers. *Communications of the ACM*, 14(10):667–668, 1971.

[8] T.S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR-93-02-02, Dept. of Computer Science, Univ. of Washington, February 1993.

[9] R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *18th international symposium on distributed computing*, October 2004. *LNCS 3274* Springer Verlag 2004, 71–85.

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[11] R. Dvir and G. Taubenfeld. Mutual exclusion algorithms with constant rmr complexity and wait-free exit code. In *Proc. of the 21st international conference on principles of distributed systems (OPODIS 2017)*, October 2017.

[12] S. Gokhale and N. Mittal. Fast and scalable group mutual exclusion, 2019. `arXiv:1805.04819`.

[13] W. Golab and A. Ramaraju. Recoverable mutual exclusion. In *Proc. 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.

[14] V. Hadzilacos. A note on group mutual exclusion. In *Proc. 20th symp. on Principles of distributed computing*, pages 100–106, 2001.

[15] Y. He, K. Gopalakrishnan, and E. Gafni. Group mutual exclusion in linear time and space. *Theoretical Computer Science*, 709:31–47, 2018.

[16] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, July 2003.

[17] P. Jayanti, S. Jayanti, and S. Jayanti. Towards an ideal queue lock. In *Proc.*

*21st International Conference on Distributed Computing and Networking*, ICDCN 2020, pages 1–10, 2020.

[18] P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *Proc. 22th ACM Symp. on Principles of Distributed Computing*, pages 275–284, July 2003.

[19] Yuh-Jzer Joung. Asynchronous group mutual exclusion. In *Proc. 17th ACM Symp. on Principles of Distributed Computing*, pages 51–60, August 1998.

[20] Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[21] H. Kakugawa, S. Kamei, and T. Masuzawa. A token-based distributed group mutual exclusion algorithm with quorums. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1153–1166, 2008.

[22] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proc. 18th ACM Symp. on Principles of Distributed Computing*, pages 23–32, 1999.

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

[24] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[25] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[26] M. Takamura, T. Altman, and Y. Igarashi. Speedup of Vidyasankar's algorithm for the group k-exclusion problem. *Inf. Process. Lett.*, 91(2):85–91, 2004.

[27] M. Toyomura, S. Kamei, and H. Kakugawa. A quorum-based distributed algorithm for group mutual exclusion. In *Proc. of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 742–746, 2003.

[28] K. Vidyasankar. A simple group mutual $\ell$-exclusion algorithm. *Inf. Process. Lett.*, 85(2):79–85, 2003.

[29] Kuen-Pin Wu and Yuh-Jzer Joung. Asynchronous group mutual exclusion in ring networks. In *Proc. 13th Inter. Parallel Processing Symposium and 10th Symp. on Parallel and Distributed Processing*, pages 539–543, 1999.

המרכז הבינתחומי בהרצליה

בית ספר אפי ארזי למדעי המחשב

התכנית לתואר שני (M.Sc.) - מסלול מחקרי

# אלגוריתם מניעה הדדית לקבוצות עם סיבוכיות קבועה של גישות לזיכרון מרוחק עבור כל מספר של תהליכים וקבוצות

מאת

**ליאת מאור**

עבודת תזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך

במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

יולי 2021

עבודה זו בוצעה בהדרכתו של פרופ' גדי טאובנפלד מבי"ס אפי ארזי למדעי המחשב, המרכז הבינתחומי, הרצליה.

# סיכום התזה

בעבודה זו מוצג אלגוריתם לפתרון בעיית המניעה ההדדית לקבוצות. האלגוריתם מורכב מקוד כניסה, המבוצע לפני הגישה לקטע הקריטי, וקוד יציאה, המבוצע לאחר הגישה לקטע הקריטי. יחדיו, קוד הכניסה וקוד היציאה, מרכיבים "מנעול קבוצתי", כך שכל התהליכים השייכים לאותה הקבוצה רשאים לגשת לקטע הקריטי שלהם במקביל ואילו תהליכים השייכים לקבוצות שונות לא רשאים לגשת לקטע הקריטי שלהם יחד באותו הזמן.

האלגוריתם המוצג בעבודה זו הוא האלגוריתם הראשון לפתרון בעיית המניעה ההדדית לקבוצות בעל סיבוכיות קבועה של גישות לזיכרון רחוק גם במודל זיכרון הכולל מטמון וגם במודל זיכרון משותף מבוזר. בנוסף, הוא גם האלגוריתם הראשון שמאפשר ריצה של כמות תהליכים שרירותית ודינמית עם כמות שרירותית של קבוצות. האלגוריתם מספק את התכונות הבאות:

1. מניעה הדדית: שני תהליכים יכולים להימצא בקטע הקריטי שלהם במקביל רק אם שניהם מבקשים את אותו הסשן.

2. מניעת הרעבה: אם תהליך מבקש לגשת לקטע הקריטי שלו, אז הוא בהכרח יוכל לגשת אליו בסופו של דבר.

3. כניסה מקבילית של קבוצה: אם תהליך $p$ מבקש סשן $s$ ואין תהליך שמבקש סשן אחר במקביל, אז (1) תהליך כלשהו המבקש סשן $s$ יכול להשלים את קוד הכניסה שלו במספר חסום של פעולות שלו, ו-(2) תהליך $p$ בסופו של דבר ישלים את קוד הכניסה שלו, גם אם תהליכים אחרים עדיין נמצאים בקטע הקריטי שלהם. (האלגוריתם מספק גם תכונה חזקה יותר של כניסה מקבילית של קבוצה)

4. יציאה חסומה של קבוצה: אם תהליך $p$ נמצא בקוד היציאה שלו, אז (1) תהליך כלשהו יכול להשלים את קוד היציאה שלו במספר חסום של פעולות שלו, ו-(2)

תהליך $p$ בסופו של דבר ישלים את קוד היציאה שלו.

5. "הראשון שבא הוא הראשון שנכנס": אם תהליך $p$ משלים את ה-doorway שלו לפני שתהליך אחר $q$ נכנס ל-doorway שלו ושני התהליכים מבקשים ססשנים שונים זה מזה, אז תהליך $q$ לא יכול להיכנס לקטע הקריטי שלו לפני שתהליך $p$ נכנס לקטע הקריטי שלו.

6. "הראשון שבא הוא הראשון שרשאי להיכנס": אם תהליך $p$ משלים את ה-doorway שלו לפני שתהליך $q$ נכנס ל-doorway שלו, שני התהליכים מבקשים את אותו הסשן, ותהליך $q$ נכנס לקטע הקריטי שלו לפני תהליך $p$, אז תהליך $p$ נכנס לקטע הקריטי שלו במספר חסום של פעולות שלו.

7. מתאים למערכות דינמיות: האלגוריתם מאפשר ריצה של כמות תהליכים שרירותית ודינמית עם כמות שרירותית של ססשנים, כלומר, תהליכים יכולים להופיע ולהיעלם באופן שרירותי וכמות הססשנים אינה חסומה.

8. סיבוכיות קבועה של גישות לזיכרון מרוחק: פעולה שתהליך מבצע על מקום בזיכרון נחשבת גישה לזיכרון מרוחק (RMR) אם התהליך לא יכול לבצע את הפעולה לוקאלית במטמון שלו או בזיכרון שלו. האלגוריתם בעבודה זו משיג סיבוכיות קבועה של גישות לזיכרון מרוחק עבור מודל זיכרון הכולל מטמון ועבור מודל זיכרון משותף מבוזר.

9. סיבוכיות מקום קבועה עבור תהליך בודד: עבור כל תהליך מוקצים מעט מקומות בזיכרון.

10. פעולות אטומיות: האלגוריתם עושה שימוש בפעולות אטומיות הנתמכות היום ברוב המעבדים המודרנים, והן: קריאה, כתיבה, קריאה-כתיבה, השוואה-והחלפה.

התכונות "כניסה מקבילית של קבוצה" (גם התכונה החזקה יותר של "כניסה מקבילית של קבוצה") ו"יציאה חסומה של קבוצה" מוצגים לראשונה בעבודה זו וזאת על מנת

להתגבר על החסם התחתון לסיבוכיות של גישות לזיכרון מרוחק במודל זיכרון משותף מבוזר המוכח ב-[9].

## תקציר

בעיית מניעה הדדית לקבוצות, שהוצגה לראשונה על ידי יאנג בשנת 1998, היא בעיית סנכרון המכלילה את בעיית המניעה ההדדית הקלאסית ואת בעיית הקוראים והכותבים. בבעיית מניעה הדדית לקבוצות, תהליך מבקש ססן לפני שהוא נכנס לקטע הקריטי; תהליכים רשאים להימצא בקטע הקריטי שלהם באותו הזמן בתנאי שהם מבקשים את אותו הססן.

בעבודה זו מוצג אלגוריתם לפתרון בעיית המניעה ההדדית לקבוצות אשר (1) הוא הראשון בעל סיבוכיות קבועה של גישות לזיכרון מרוחק עבור מודל זיכרון הכולל מטמון (CC) וגם עבור מודל זיכרון משותף מבוזר (DSM) ו-(2) הוא הראשון שמאפשר ריצה של כמות תהליכים שרירותית ודינמית עם כמות שרירותית של ססנים. כל האלגוריתמים הקיימים היום לפתרון בעיית המניעה ההדדית לקבוצות לא מקיימים אף תכונה משתי התכונות החשובות הללו. בנוסף, סיבוכיות המקום עבור תהליך בודד של האלגוריתם המוצג בעבודה זו היא קבועה. יתר על כן, האלגוריתם מספק שתי תכונות חזקות של הגינות, הקבוצה הראשון שמבקשת להיכנס לקטע הקריטי היא הקבוצה הראשונה שתיכנס לקטע הקריטי והתהליך הראשון שמבקש להיכנס לקטע הקריטי בקבוצה מסוימת הוא התהליך הראשון מהקבוצה שיהיה רשאי להיכנס לקטע הקריטי.

האלגוריתם משתמש ברגיסטרים אטומיים הנתמכים היום ברוב המעבדים המודרניים והם: קריאה, כתיבה, קריאה-וכתיבה, השוואה-והחלפה.