Fast and Scalable Group Mutual Exclusion

Shreyas Gokhale
The University of Texas at Dallas
Richardson, TX 75080, USA
shreyas.gokhale@utdallas.edu
https://orcid.org/0000-0002-7589-6927

Neeraj Mittal
The University of Texas at Dallas
Richardson, TX 75080, USA
neerajm@utdallas.edu
https://orcid.org/0000-0002-8734-1400

Abstract

The group mutual exclusion (GME) problem is a generalization of the classical mutual exclusion problem in which every critical section is associated with a type or session. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The well-known read-write mutual exclusion problem is a special case of the group mutual exclusion problem.

In this work, we present a novel GME algorithm for an asynchronous shared-memory system that, in addition to satisfying lockout freedom, bounded exit and concurrent entering properties, has $O(1)$ step-complexity when the system contains no conflicting requests as well as $O(1)$ space-complexity per GME object when the system contains sufficient number of GME objects. To the best of our knowledge, no existing GME algorithm has $O(1)$ step-complexity for concurrent entering. Moreover, most existing GME algorithms have $\Omega(n)$ space complexity per GME object, where $n$ denotes the number of processes in the system. We also show that our GME algorithm can be easily modified to use bounded space variables.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies → Concurrent algorithms

Keywords and phrases Group Mutual Exclusion, Multiple Locks, Space Complexity, Step Complexity, Contention-Free Complexity, Remote Memory Reference Complexity

Digital Object Identifier 10.4230/LIPIcs.CVIT.2018.23

Category Regular Paper (Eligible for Student Best Paper Award)

1 Introduction

The group mutual exclusion (GME) problem is a generalization of the classical mutual exclusion (ME) problem in which every critical section is associated with a type or session [12]. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. The GME problem models situations in which a resource may be accessed at the same time by processes of the same group, but not by processes of different groups. As an example, suppose data is stored on multiple discs in a shared CD-jukebox. When a disc is loaded into the player, users that need data on that disc can access the disc concurrently, whereas users that need data on a different disc have to wait until the current disc is unloaded [12]. Another example includes a meeting room for philosophers interested in different forums or topics [13] [21].
The well-known readers/writers problem is a special case of the group mutual exclusion problem in which all read critical sections belong to the same session but every write critical section belongs to a separate session.

Note that any algorithm that solves the mutual exclusion problem also solves the group mutual exclusion problem. However, the solution is inefficient since critical sections are executed in a serial manner and thus the solution does not permit any concurrency. To rule out such inefficient solutions, a group mutual exclusion algorithm needs to satisfy concurrent entering property. Roughly speaking, the concurrent entering property states that if all processes are requesting the same session, then they must be able to execute their critical sections concurrently.

The GME problem has been defined for both message-passing and shared-memory systems. The focus of this work is to develop an efficient GME algorithm for shared-memory systems.

1.1 Related Work

Since the GME problem was first introduced by Joung around two decades ago [12], several algorithms have been proposed to solve the problem for shared-memory systems [12, 14, 5, 21, 11, 3, 1, 6]. These algorithms provide different trade-off between fairness, concurrency and performance. A detailed description of the related work is given in section 5. To the best of our knowledge, all of the prior work suffers from at least one and possibly both of the following drawbacks.

Drawback 1 (high space complexity with multiple GME objects): All the existing work in this area has (implicitly) focused on a single GME object. However, many systems use fine-grained locking to achieve increased scalability in multi-core/multi-processor systems. For example, each node in a concurrent data structure may be protected by a separate lock [9]. Recently, GME-based locks have been used to improve the performance of skip lists using the notion of unrolling by storing multiple key-value pairs in a single node [20].

All the existing GME algorithms that guarantee starvation freedom have a space-complexity of at least $\Theta(n)$ per GME object. Note that this is expected because mutual exclusion is a special case of group mutual exclusion and any starvation-free mutual exclusion algorithm requires $\Omega(n)$ space even when powerful atomic instructions such as compare-and-swap are used [2]. Some of these GME algorithms (e.g., [14, 3, 6]) can be modified relatively easily to share the bulk of this space among all GME objects and, as a result, the additional space usage for each new GME object is only $O(1)$. However, it is not clear how the other GME algorithms (e.g., [12, 5, 21, 11, 3, 1, 6]) can be modified to achieve the same space savings. For these GME algorithms, the additional space usage for each new GME object is at least $\Theta(n)$. We refer to the former set of GME algorithms as space-efficient and the latter set of GME algorithms as space-inefficient.

Consider the example of a concurrent data structure using GME-based locks to improve performance. If $n$ is relatively large, then the size of a node equipped with a lock based on a GME algorithm that is space-inefficient may be several factors more than its size otherwise. This will significantly increase the memory footprint of the concurrent data structure, which, in turn, will adversely affect its performance and may even negate the benefit of increased concurrency resulting from using a GME-based lock.

Drawback 2 (high step complexity in uncontended situations): In a system using fine-gained locking, most of the lock acquisitions are uncontended in practice (i.e., at most one process is trying to acquire the same lock). However, in almost all of the existing GME algorithms (e.g., [12, 14, 5, 21, 11, 3, 6]) except for one [1], a process needs to execute at least
S. Gokhale and N. Mittal

$\omega(1)$ steps (and $\Theta(n)$ in most cases) in order to lock a GME object even though no other process wants to lock the same object simultaneously. Clearly, this is inefficient and highly undesirable if $n$ is large. Note that two requests to lock the same GME object conflict only if they involve different sessions. Therefore, it is actually desirable to design GME algorithms in which a process is able to lock a GME object within a small (preferably, constant) number of its own steps in the worst-case even if all processes are trying to lock the same GME object simultaneously, provided all requests involve the same session. Even the GME algorithm in [1] has $\omega(1)$ step complexity (specifically, $\Theta(\log n)$ step complexity) in this situation.

1.2 Our Contributions

In this work, we present a novel GME algorithm that, in addition to satisfying the usual group mutual exclusion, lockout freedom, bounded exit and concurrent entering properties, has the following desirable features. First, a process can enter its critical section within a constant number of its own steps as long as no other process is trying to enter a critical section belonging to a different session. Note that, as a corollary, a process can enter its critical section within a constant number of its own steps in the absence of any other request (typically referred to as contention-free step complexity). Second, it uses only $O(m + n)$ space for managing $m$ GME objects, where $O(n)$ space is shared among all $m$ GME objects. In addition, each process needs only $O(\ell)$ space, where $\ell$ denotes the maximum number of GME objects (or locks) a process needs to hold at the same time, which is space-optimal. To the best of our knowledge, no existing GME algorithm has both the aforementioned features. We further show that our algorithm can be easily modified to use bounded space variables.

Two other important performance measures of a GME algorithm are context switch complexity and remote memory reference (RMR) complexity. Roughly speaking, context switch complexity measures the worst-case number of different sessions that can be established while a process is waiting to enter a session, and RMR complexity measures the worst-case number of remote (non-local) memory references required by a process in order to enter and then exit its critical section. Both the context switch complexity and the RMR complexity (under the cache coherent model) of our algorithm are $O(n)$. The two complexities are comparable to those of most other existing GME algorithms.

1.3 Roadmap

The rest of the text is organized as follows. We present the system model and describe the problem in section 2. In section 3 we describe our GME algorithm, prove its correctness and analyze its complexity. Section 4 describes the related work. We describe a way to bound the values of all the variables used in our GME algorithm in section 4. Finally, section 6 concludes the text and outlines directions for future work.

2 System Model and Problem Specification

2.1 System Model

We consider an asynchronous shared-memory system consisting of $n$ processes labeled $p_1, p_2, \ldots, p_n$. Each process also has its own private variables. Processes can only communicate by performing read, write and atomic (read-modify-write) instructions on shared variables. A system execution is modeled as a sequence of process steps. In each step, a process either performs some local computation affecting only its private variables or executes one of the available instructions (read, write or atomic) on a shared variable. Processes
take steps asynchronously. This means that in any execution, between two successive steps of a process, there can be an unbounded but finite number of steps performed by other processes. Every process is assumed to be live meaning that, if it has not terminated, then it will eventually execute its next step.

### 2.2 Synchronization Instructions

We assume the availability of three atomic instructions, namely fetch-and-increment (FAI), fetch-and-decrement (FAD) and load-linked/store-conditional (LL/SC).

A fetch-and-increment instruction takes a shared variable $x$ as input, returns the current value of $x$ as output, and, at the same time, increments the value of $x$ by one.

A fetch-and-decrement instruction takes a shared variable $x$ as input, returns the current value of $x$ as output, and, at the same time, decrements the value of $x$ by one.

Load-linked and store-conditional instructions are performed in pairs and behave in a similar manner as their simpler load and store counterparts, but with some additional features. A load-linked instruction takes a shared variable $x$ as input, and returns the current value of $x$ as output. A store-conditional instruction takes a shared variable $x$ and a value $v$ as inputs. If the value of $x$ has not been modified by any process since the associated load-linked instruction was performed on $x$, it overwrites the current value of $x$ with $v$, and returns true as output. Otherwise, it leaves $x$ unchanged, and returns false as output.

### 2.3 Problem Specification

In the GME problem, each process repeatedly executes four sections of code, namely non-critical section (NCS), entry section, critical section (CS), and exit section, as shown in Algorithm 1. Each critical section is associated with a type or a session. Critical sections belonging to the same session can execute concurrently, whereas critical sections belonging to different sessions must be executed serially. We refer to the code executed by a process from the beginning of its entry section until the end of its exit section as an attempt. Note that the session associated with a critical section may be different in different attempts (and is selected based on the needs of the underlying application). We say that a process is active if it is in one of its attempts.

Solving the GME problem involves designing code for entry and exit sections in order to ensure the following four properties are satisfied in each attempt:

(P1) **Group mutual exclusion** If two processes are in their critical sections at the same time, then they have requested the same session.

(P2) **Lockout freedom** If a process is trying to enter its critical section, then it is able to do so eventually (entry section is finite).

(P3) **Bounded exit** If a process is trying to leave its critical section, then it is able to do so eventually within a bounded number of its own steps (exit section is bounded).

**Algorithm 1: Structure of a GME Algorithm**

```
while true do
  Non-Critical Section (NCS)
  Entry Section  // try to enter critical section
  Critical Section (CS)  // execute critical section
  Exit Section  // exit critical section
end while
```
If a process is trying to enter its critical section and no process in the system is requesting a different session, then the (former) process is able to enter its critical session eventually within a bounded number of its own steps (entry section is bounded in the absence of a request for a different session).

2.3.1 Complexity Measures

We say that a request is outstanding until its process has finished executing the exit section. We use the following metrics to evaluate the performance of our GME algorithm:

Context switch complexity It is defined as the maximum number of sessions that can be established while a process is waiting to enter its critical section. It is also referred to as session switch complexity elsewhere [14, 5].

Solitary request step complexity It is defined as the maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding request during that period. It is also referred to as contention-free time complexity elsewhere [14, 5].

Concurrent entering step complexity It is defined as the maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding request for a different session during that period.

Remote memory reference (RMR) complexity It is defined as the maximum number of remote memory references required by a process in its entry and exit sections.

In addition, we also consider the memory footprint of the GME algorithm when the system contains multiple GME objects.

Multi-object space complexity It is defined as the maximum amount of space needed to instantiate and maintain a certain number of GME objects.

We analyze the RMR complexity of a GME algorithm is under cache-coherent (CC) model, which is the most common model used for RMR complexity analysis.

In the CC model, all shared variables are stored in a central location or global store. Each processor has a private cache. When a process accesses a shared variable, a copy of the contents of the variable is saved in the private cache of the process. Thereafter, every time the process reads that shared variable, it does so using its cached (local) copy until the cached copy is invalidated. Also, every time a process writes to a shared variable, it writes to the global store, which also invalidates all cached copies of the variable. In the CC model, spinning on a memory location generates at most two RMRs—one when the variable is cached and the other when the cached copy is invalidated.

An algorithm is called local-spinning if the maximum number of RMRs made in entry and exit sections is bounded. It is desirable to design algorithms that minimize the number of remote memory references because this factor can critically affect the performance of these algorithms [18].

3 The Group Mutual Exclusion Algorithm

3.1 The Main Idea

Our GME algorithm is inspired by Herlihy’s universal construction for deriving a wait-free linearizable implementation of a concurrent object from its sequential specification using
Roughly speaking, the universal construction works as follows. The state of the concurrent object is represented using (i) its initial state and (ii) the sequence of operations that have applied to the object so far. The two are maintained using a singly linked list in which the first node represents the initial state and the remaining nodes represent the operations. To perform an operation, a process first creates a new node and initializes it with all the relevant details of the operation, namely its type and its input arguments. It then tries to append the node at the end of the list. To manage conflicts in case multiple processes are trying to append their own node to the list, a consensus object is used to determine which of several nodes is chosen to be appended to the list. Specifically, every node stores a consensus object and the consensus object of the current last node is used to decide its successor (i.e., the next operation to be applied to the object). A process whose node is not selected simply tries again. A helping mechanism is used to guarantee that every process trying to perform an operation eventually succeeds in appending its node to the list.

We modify the aforementioned universal construction to derive a GME algorithm that satisfies several desirable properties. Intuitively, an operation in the universal construction corresponds to a critical section request in our GME algorithm. Appending a new node to the list thus corresponds to establishing a new session. However, unlike in the universal construction, a single session in our GME algorithm can be used to satisfy multiple critical section requests. This basically means that every critical section request does not cause a new node to be appended to the list. This requires some careful bookkeeping so that no “empty” sessions are established. Further, a simple consensus algorithm, implemented using LL/SC (atomic) instructions, is used to determine the next session to be established.

Herlihy’s universal construction has two main shortcomings. First, as more and more nodes are appended to the list, the list can grow unboundedly. Second, the value of one of the variables in the construction (sequence number associated with a node) can also grow unboundedly. We present techniques to address both of the shortcomings.

We describe our GME algorithm in three parts. First, we describe a basic GME algorithm that is only deadlock-free (some session request is eventually satisfied but a given request may be starved), but uses unbounded space. Second, we enhance the basic algorithm to achieve starvation freedom (every session request is eventually satisfied) using a helping mechanism. Third, we further enhance the algorithm to make it space-efficient by reusing nodes (so that the list has bounded length), but some variables are still unbounded. Note that all our algorithms are safe in the sense that they satisfy the group mutual exclusion property. In the next section, we describe a way to bound the values of all the variables used in our GME algorithm.

### 3.1.1 Achieving Deadlock-Freedom

Central to our GME algorithm is a (list) node; it is used to maintain information about a session. A node stores the following information among other things: (a) the session represented by the node, (b) the instance identifier of the GME object to which the session belongs, (c) the state of the session, (d) the size of the session, and (e) the address of the next node in the list.

A session has three possible states: (i) open: it means that the session is currently in progress and new processes can join in, (ii) closed: it means the session is currently in progress but no new processes can join in, and (iii) adjourned: it means that the session is no longer in progress and has no participating processes. When a session is first established, it is in open state. It stays open as long as one of the following conditions still holds: (1) there is no conflicting request in the system, and (2) the request that established the session is
Algorithm 2: Data types and variables used.

1 struct Node {
   integer session; // session associated with the node
   integer instance; // instance identifier of the GME object
   \{bool,bool,bool,bool\} state; // four flags representing state
   integer size; // number of processes currently in the session
   NodePtr next, prev; // address of the next and previous node
   integer condition; // tuple consisting of two fields: owner and type
};

2 struct MetaData {
   NodePtr head; // address of the last node in the list
   integer number; // sequence number associated with the list
};

shared variables
   // next three variables are used to store list meta-data, one entry per GME object
3 head: array [1..m] of NodePtr;
4 lhsNumber: array [1..m] of integer, initially [0,...,0];
5 rhsNumber: array [1..m] of integer, initially [0,...,0];

// next variable is used to announce CS requests, one entry per process
6 announce: array [1..n] of NodePtr, initially [null,...,null];

// next variable is used to store snapshot of meta data, one entry per process
7 snapshot: array [1..n] of MetaDataPtr;

// next variable is used to store pools of nodes, two pools of 2n nodes per process
8 pool: array [1..n][1..2][1..2n] of NodePtr;

initialization
begin
foreach i ∈ [1..m] do
   head[i] := new Node; // create a new node
   head[i] → state := LEADERLESS | CONFLICT | VACANT; // session is adjourned
   head[i] → size := 0; // session has no processes
   head[i] → next := null; // node has no successor
   head[i] → condition := ⟨1,DIRTY⟩; // all other fields can be initialized arbitrarily
end foreach

foreach i ∈ [1..n], j ∈ [1..2], k ∈ [1..2n] do
   pool[i][j][k] := new Node; // create a new node
   pool[i][j][k] → condition := ⟨i,CLEAN⟩;
   designate pool[i][1] as active and pool[i][2] as passive;
end foreach

end

still outstanding, i.e., executing its critical section. Once both the conditions become false, the session moves to closed state. Note that, in closed state, the session may still have participants executing their critical sections. Finally, once all such participants have left the session, the session moves to adjourned state.

The size of a session refers to the number of processes that have joined or trying to join the session.

Each GME object has a separate linked list associated with it. Each list has a head, which points to the last node in the list. Initially, the head of each list points to a “dummy” node representing an adjourned session.

Whenever a process generates a critical section request, it creates a new node and initializes it appropriately. Specifically, the session state is set to open, the number of processes in the session is set to one, and the address of the next node is set to null. The process then
Algorithm 3: Entry and exit sections.

// code for entry section
24 Enter(integer myinstance, integer mysession)
25 begin
26 // initialize a node and announce the request to other processes
27 NodePtr mynode := announce[myid];
28 while true do
29   ReadHead(myinstance); // take an atomic snapshot of the meta-data
30   NodePtr current := snapshot[myid] → head; // find the last node in the list
31       if (current = mynode) then return; // join the session as a leader
32         if TestHead(myinstance) and isOpen(current → state) then
33           FAI(current → size); // increment the session size
34         else // either the session is no longer open or the head has moved
35           FAD(current → size); // decrement the session size
36         end if
37       else // a conflicting request has been generated
38         SetConditionFlag(myinstance, CONFLICT); // set CONFLICT flag
39         SetVacantFlag(current); // set VACANT flag if applicable
40       end if
41   while TestHead(myinstance) and not(IsAdjourned(current → state)) do
42     ; // spin
43 end while
44 AppendNextNode(myinstance); // append a new node to the list
45 end while
46 // code for exit section
47 Exit(integer myinstance)
48 begin
49   NodePtr current := snapshot[myid] → head; // find the last node in the list
50   NodePtr mynode := announce[myid]; // find the node for my request
51   if (current = mynode) then // I am the leader of the session
52     ReclaimNode(current → prev); // reclaim the predecessor node
53     current → prev := null;
54     SetConditionFlag(myinstance, LEADERLESS); // set LEADERLESS flag
55   else // I am not the leader of the session
56     ReclaimNode(mynode); // reclaim my own node
57   end if
58   FAD(current → size); // decrement the session size
59   SetVacantFlag(current); // set VACANT flag if applicable
60 end

repeatedly performs the following steps until it is able to enter its critical section.

(1) It locates the current last node of the linked list associated with the GME object.
(2) It next checks the session associated with the node. If the session is open and is compatible
with its own request, it attempts to join the session by incrementing the session size using
an FAI instruction. It then ascertains that the session is still in open state. If yes, it
enters its critical section. If no, it decrements the session size using an FAD instruction.
Algorithm 4: Functions operating on session state.

```
65 bool IsOpen(integer state) { return not(state & LEADERLESS) or not(state & CONFLICT); }

66 bool IsAdjourned(integer state) { return (state & VACANT); }

67 SetConditionFlag(integer instance, bool flag)
begin
NodePtr node := snapshot[myid] → head;
while true do
integer state := LL(node → state);
if (state & flag) then return;
if not(TestHead(instance)) then return;
if SC(node → state, state | flag) then return;
end while
end

68 SetVacantFlag(NodePtr node)
begin
integer state := LL(node → state);
if IsOpen(state) then return;
if (node → size ≠ 0) then return;
SC(node → state, state | VACANT);
end
```

(3) If it is unable to join the session in the previous step for any reason (e.g., the session was not compatible with its own request or was not open or was closed before it could join), it waits for the session to move to adjourned state.

(4) It then tries to append its own node as the next node in the list using an LL/SC instruction and, irrespective of the outcome, helps advance the head of the list to the next node. If it was able to append its own node to the list, it enters its critical section. Otherwise, it restarts from the beginning.

We say that a process enters its critical section as a leader if it was able to successfully append its own node to the list in the last step. Otherwise, we say that it enters as a follower. On leaving the critical section, a process performs the following steps.

(1) It decrements the session size using a FAD instruction.

(2) If it entered its critical section as a leader and there is a conflicting request in the system, it closes the session.

(3) If the session is closed and the session size is zero, it adjourns the session.

Note that, if a process enters its critical section as a follower, then its node is never appended to the list. As a result, the process can reuse the node for its next request.

Clearly, the algorithm is not starvation free since the node of a process may never be appended to the list.

3.1.2 Achieving Starvation-Freedom

To achieve starvation-freedom, we use the helping mechanism used in many wait-free algorithms. After creating a node and initializing it, the process announces its request to other processes by storing the node’s address in a shared array, which has one entry for each
Algorithm 5: Functions operating on list meta-data.

// takes an atomic snapshot of the meta-data of a list
84 \textbf{READ\textsc{Head}}(\text{integer} \ \text{instance})
85 \textbf{begin}
86 \hspace{1em} // repeatedly read all the entries of the meta-data until sequence numbers match
87 \hspace{1em} \textbf{while} \ \text{true} \ \textbf{do}
88 \hspace{2em} \text{integer} \ \text{number} := \text{rhsNumber}[\text{instance}];
89 \hspace{2em} \text{NodePtr} \ \text{current} := \text{head}[\text{instance}];
90 \hspace{2em} \textbf{if} \ \text{number} \neq \text{lhsNumber}[\text{instance}] \ \textbf{then} \quad \text{// meta-data only partially updated}
91 \hspace{3em} \text{FIX\textsc{Head}}(\text{instance}, \text{number}); \quad \text{// fix the rest of the meta-data}
92 \hspace{2em} \textbf{else}
93 \hspace{3em} \text{snapshot}[\text{myid}] \rightarrow \text{number} := \text{number};
94 \hspace{3em} \text{snapshot}[\text{myid}] \rightarrow \text{head} := \text{current};
95 \hspace{3em} \textbf{if} \ (\text{snapshot}[\text{myid}] \rightarrow \text{head} = \text{head}[\text{instance}]) \ \textbf{then} \ \textbf{break};
96 \hspace{2em} \textbf{end} \ \text{if}
97 \hspace{1em} \textbf{end} \ \text{while}
98 \textbf{end}

// returns true if the meta-data of the list has not been updated since the snapshot was taken and false otherwise; note that the method returns true if the meta-data is only partially updated
99 \textbf{bool} \ \textbf{TEST\textsc{Head}}(\text{integer} \ \text{instance})
100 \textbf{begin}
101 \hspace{1em} \textbf{if} \ (\text{rhsNumber}[\text{instance}] \neq \text{snapshot}[\text{myid}] \rightarrow \text{number}) \ \textbf{then}
102 \hspace{2em} \textbf{return} \ \text{false}; \quad \text{// the sequence numbers do not match}
103 \hspace{1em} \textbf{else} \ \textbf{return} \ \text{true};
104 \hspace{1em} \textbf{end}
105 \textbf{end}

// fixes partially updated meta-data of a list
106 \textbf{FIX\textsc{Head}}(\text{integer} \ \text{instance}, \ \text{integer} \ \text{observed})
107 \textbf{begin}
108 \hspace{1em} \textbf{// read the relevant meta-data using LL instructions}
109 \hspace{2em} \text{NodePtr} \ \text{current} := \text{LL}(\text{head}[\text{instance}]);
110 \hspace{2em} \text{integer} \ \text{number} := \text{LL}(\text{rhsNumber}[\text{instance}]);
111 \hspace{2em} \text{NodePtr} \ \text{successor} := \text{current} \rightarrow \text{next}; \quad \text{// find the successor of the head}
112 \hspace{2em} \textbf{if} \ (\text{number} \neq \text{observed}) \ \textbf{then} \ \textbf{return}; \quad \text{// meta-data has been updated completely}
113 \hspace{2em} \textbf{if} \ (\text{successor} \neq \text{null}) \ \textbf{then} \ \text{SC}(\text{head}[\text{instance}], \ \text{successor}); \quad \text{// update the head}
114 \hspace{2em} \text{SC}(\text{rhsNumber}[\text{instance}], \ \text{observed} + 1); \quad \text{// update the RHS number}
115 \hspace{1em} \textbf{end}

// advances the head of a list to the given node
116 \textbf{ADVANCE\textsc{Head}}(\text{integer} \ \text{instance}, \ \text{NodePtr} \ \text{successor})
117 \textbf{begin}
118 \hspace{1em} \textbf{// read the LHS number using LL instruction}
119 \hspace{2em} \text{integer} \ \text{number} := \text{LL}(\text{lhsNumber}[\text{instance}]);
120 \hspace{2em} \textbf{if} \ (\text{number} = \text{snapshot}[\text{myid}] \rightarrow \text{number}) \ \textbf{then}
121 \hspace{3em} \text{SC}(\text{lhsNumber}[\text{instance}], \ \text{number} + 1); \quad \text{// update the LHS number}
122 \hspace{2em} \textbf{end} \ \text{if}
123 \hspace{2em} \textbf{FIX\textsc{Head}}(\text{instance}, \ \text{snapshot}[\text{myid}] \rightarrow \text{number}); \quad \text{// fix the rest of the meta-data}
124 \hspace{1em} \textbf{end}

process. When establishing a new session, instead of always trying to append its own node at the end of the list, it may try to help another process append its node to the list. We use a simple round-robin scheme to determine which process to help by storing a counter in the node. Every time a new node is appended to the list, its counter value is set to one more than that of its predecessor.

Note that, if a process enters its critical section as a follower, there is no need to append its node to the list. In that case, the process revokes its announcement; only unrevoked
Algorithm 6: Functions operating on a list node.

// get a free node, initialize it and announce it to other processes
121 INITIALIZENEXTNODE(integer instance, integer session)
122 begin
123 NodePtr node := a clean node from the active pool; // initialize node's instance
124 node → instance := instance;
125 node → session := session; // initialize node's session
126 FAI(node → size); // increment session size
127 node → next := null; // node has no successor
128 node → prev := null; // node has no predecessor
129 node → state := 0; // session is open with no condition flag set
130 node → condition := ⟨myid, DIRTY⟩; // mark the node as dirty
131 announce[myid] := node; // make the node visible to other processes
132 end

// get the next node to be appended to the list
133 NodePtr GETHENEXTNODE(integer instance)
134 begin
135 NodePtr mine := announce[myid]; // my node
136 NodePtr helpee := announce[⟨(snapshot[myid] → number) mod n + 1⟩]; // helpee's node
137 // ascertain that the helpee's node is usable
138 if (helpee = null) then return mine; // process has no outstanding request
139 if (helpee → instance ≠ instance) then // request is for a different GME object
140 return mine;
141 else if (helpee → state & RECLAIM) then return mine; // node has been revoked
142 if not(THEAD(instance)) then return mine; // head has moved
143 return helpee; // helpee's node passed all the tests
144 end

// append a new node to the list
145 APPENDNEXTNODE(integer instance)
146 begin
147 NodePtr current := snapshot[myid] → head; // get the last node in the list
148 NodePtr successor := GETNEXTNODE(instance); // choose a node to append
149 // set the next field of the current last node
150 NodePtr next := LL(current → next); // read the next field using LL instruction
151 if not(THEAD(instance)) then return; // head has moved
152 if (next = null) then SC(current → next, successor); // next field still not set
153 // set the previous field of the new last node
154 successor := current → next; // read the address that was written to the next field
155 if not(THEAD(instance)) then return; // head has moved
156 NodePtr prev := LL(successor → prev); // read the previous field using LL instruction
157 if not(THEAD(instance)) then return; // head has moved
158 if (prev = null) then SC(successor → prev, current); // previous field still not set
159 ADVANCEHEAD(instance, successor); // advance the head
160 end

// reclaim the node for later use
161 RECLAIMNODE(NodePtr node)
162 begin
163 announce[myid] := null; // request already fulfilled
164 node → state := LEADERLESS | CONFLICT | VACANT | RECLAIM; // mark the node as reclaimed
165 add the node to the active pool;
166 end

nodes can be appended to the list. To that end, we define a fourth state for a session, namely canceled. To revoke a node, a process changes the state of the session (request) in its node from open to canceled. Once a node has been revoked, no process tries to append it to the
Algorithm 7: Cleanup algorithm.

```
begin
    which := index of the passive pool;
    // mark all nodes in the pool as clean
    foreach i ∈ \[1 . . . 2n\] do
        pool[myid][which][i] → condition := ⟨myid, CLEAN⟩;
    end foreach
    // scan all the hazard pointers
    foreach i ∈ \[1 . . . n\] do
        node := snapshot[i] → head;
        // check if I own the node
        if (node → condition = ⟨myid, CLEAN⟩) then
            // mark the node as dirty
            node → condition.type := DIRTY;
        end if
    end foreach
    // collect all clean nodes in the passive pool toward the end of the array
    // switch the designations of the pools
end
```

Clearly, the algorithm is still space-inefficient since the list may grow unboundedly.

### 3.1.3 Achieving Space-Efficiency

We achieve space-efficiency in two steps. We first describe a way to reclaim a node whose session has been adjourned or canceled and hence no longer needed. We next describe a way to safely reuse a node that has been reclaimed.

Note that, if a process enters its critical section as a follower, then it can simply reclaim its revoked node and use it later for its next request. So, hereafter, we focus on nodes that are actually used to establish a session and describe when and how they can be reclaimed and reused. Intuitively, our node reclamation strategy is based on the one used in the queue-based locking algorithms presented in \([2, 17]\), where our node reuse strategy is based on hazard-pointers, a well known strategy for garbage collection presented in \([19]\).

**Reclaiming a Node**

If a process enters its critical section as a leader, then it cannot reclaim its own node on leaving its critical section. Among many reasons, other processes that joined its session and may still be in their critical sections. Instead, a leader, on leaving its critical section, relinquishes the ownership of its node and claims the ownership of its node’s predecessor. Node reclamation is relatively straightforward in \([2, 17]\) because, unlike in our algorithm, only one process is holding a reference to a node when it is reclaimed. In our algorithm, multiple processes may be holding a reference to the predecessor node because of the helping mechanism used to achieve wait-freedom. Note that a node may be appended to the list by any process in the system and not necessarily by its owner. This necessitates more bookkeeping and careful coordination among processes to avoid race conditions.

To enable node reclamation, when a node is appended to the list, we create a *backlink* from the appended node to its predecessor. When the owner of the appended node (also the leader of the session associated with the node) leaves its critical section, it can use this
backlink to access the predecessor and claim the predecessor as its own node. Also, we assign
a label to each node to distinguish between different instances of critical section requests
(since a node may now be associated with multiple requests). Every time node is reclaimed,
the label value is incremented. This enables processes to detect whether or not helping is
still required and, if not, abort their helping.

### Reusing a Node

After a node has been reclaimed, it may not be possible to reuse it to establish a session
right away. This is because other processes may hold references to it and may increment the
session size variable only to realize later that the session they are trying to join has adjourned;
the variable is then decremented back to correct the error. While this does not affect the
safety of the algorithm, it may interfere with the concurrent entering property, which
requires a process to enter its critical section in the absence of any outstanding conflicting
request within a bounded number of its own steps. These spurious increments may prevent a
session from being closed for arbitrarily long time.

To address this problem, we first describe a cleanup algorithm based on the notion of
hazard pointers that can be used by a process to identify a subset of nodes it can reuse from
among those it has already reclaimed. The cleanup algorithm assumes that each process has
one hazard pointer and a pool of up to $2n$ reclaimed nodes. To identify which nodes can
be reused, the process marks all the nodes as *clean*. It then scans the hazard pointers of
every process and any node whose address is found in a hazard pointer is marked as *dirty*.
At the end of the scan, any node that is still clean can be safely reused. Clearly, this cleanup
algorithm guarantees that at most $n$ nodes in the pool are marked as dirty and, thus, at
least $n$ nodes in the pool are identified as clean. Further, the worst-case step complexity
of the algorithm is $O(n)$. Later, we specify which local variable maintained by a process
constitutes a hazard pointer.

Suppose, at the beginning, a process starts with a set of $2n$ clean nodes. Note that,
whenever a process relinquishes the ownership of a node (because it entered a session as a
leader) it reclaims the ownership of another node, which is marked as dirty. Whenever the
count for dirty nodes at a process reaches $2n$, it uses the cleanup algorithm to convert at
least $n$ dirty nodes to clean nodes. This however increases the worst-case step complexity an
exit section to $O(n)$. But the amortized step complexity of an exit section still remains $O(1)$
because, after executing the cleanup algorithm once, a process does not have to execute it
again for at least $n$ subsequent critical section requests.

To maintain the worst-case step complexity of an exit section at $O(1)$, we make the
following two modifications. First, the cleanup algorithm is executed *incrementally* over
$O(n)$ requests so that only $O(1)$ steps need to be performed in every exit section on behalf
of the algorithm. Second, a process maintains two pools of $2n$ nodes each. One of the pool
is designated as active, whereas the other pool is designated as passive. An active pool,
in the beginning, is guaranteed to have at least $n$ clean nodes; it is used to service critical
section requests by providing clean nodes. A passive pool, in the beginning, may or may not
have any clean nodes; it is incrementally processed as explained earlier over a span of $O(n)$
instances of exit section. Once the processing is complete, the designations of the two pools
are switched.
3.2 Algorithm Details and Formal Description

In this section, we give a pseudocode of our GME algorithm, after describing certain details to help understand the pseudocode.

3.2.1 Algorithm details

Representing a session state

We use four flags to represent session state: (1) LEADERLESS flag to indicate that the session leader has left its critical section, (2) CONFLICT flag to indicate that some process has made a conflicting request, (3) VACANT flag to indicate that the session is empty or vacant, and (4) RECLAIM flag to indicate that the node associated with the session has been reclaimed. We refer to the first two flags as condition flags, the third flag as status flag and the fourth flag as usability flag. The status flag is set only after both the condition flags have been set. Thus a session is closed if both its access flags are set. It is adjourned if its status flag is set. Finally, a node can be reused if its usability flag is set, which also means that the node has been revoked and the associated session request canceled. For convenience, when the usability flag is set, we set the remaining three flags as well to simplify the algorithm. Thus if the status flag is set, then both the condition flags are also set; if the usability flag is set, then the status flag as well as both the condition flags are also set. All the four flags are stored in a single word hence the value of session state can be easily read atomically.

Representing a linked-list

Each linked-list is associated with the following meta-data: (1) a pointer, referred to as head, to keep track of the last node in the list, and (2) a sequence number to detect whether or not the head has been modified since it was last read. Note that, due to node reuse, even if the head of a list is found to be pointing to the same node at two different times, it may have been modified—possibly multiple times—during this period. The sequence number is also used during helping to determine the node of which process should be appended to the list next. The meta-data of a linked list is updated by performing an advance operation on it.

Taking an atomic snapshot of the meta-data of a linked list

In our GME algorithm, we need to take an atomic snapshot of the meta-data of a linked list (associated with a given GME object). Intuitively, a snapshot is atomic if it contains an instantaneous view of the meta-data. This snapshot is used at multiple points in the algorithm to detect whether or not the head of a linked list has undergone a change. Our snapshot algorithm does not need to be wait-free. It instead needs to provide a weaker progress guarantee. Specifically, if the snapshot algorithm does not overlap with any advance operation on the meta-data, then it should eventually terminate.

To that end, we use a simple well-known idea for taking an atomic snapshot. We maintain two copies of the sequence number, referred to as LHS and RHS sequence numbers. To update the meta-data, a process first increments the LHS number, then switches the head pointer and finally increments the RHS number. To take an atomic snapshot of the meta-data, a process first reads the RHS number, then reads the head pointer and finally reads the LHS number. If the values of the two numbers match, then the snapshot algorithm terminates. Otherwise, the process repeats the steps again.

To achieve concurrent entering property, before restarting (which only happens if the meta-data is undergoing a change), a process helps complete the advance operation on the
meta-data before (re)reading the RHS number again. Without this helping, the concurrent entering property does not hold as illustrated by the following scenario. Suppose all processes are request the same session, which is different from the last established session. A situation may occur in which one process has successfully appended a new node to the linked list and is now updating the meta-data, while others are trying to obtain an atomic snapshot of the meta-data. Until the meta-data update is complete, other processes may be forced to perform an unbounded (but finite) number of steps thereby violating the concurrent entering property.

**Helping an advance operation on the meta-data of a linked list complete**

A process invokes this helping procedure when it is taking a snapshot of the meta-data and detects that the meta-data is currently being modified. Specifically, it detects that the LHS number in the meta-data has been incremented but not the RHS number. As part of helping, the process needs to do the following: (1) update the head to the next node in the list if not already updated, and (2) increment the RHS number.

To that end, the processes reads the head and the RHS number (in that order) using LL instructions. It also reads the next field of the node pointed to by the head (using a simple read instruction). It then verifies that the RHS number has not yet been incremented. Otherwise, it implies that the advance operation on the meta-data has completed and no further helping is required. If the next field (read earlier) contains a non-null address, then the head is updated to the address read in the next field using an SC instruction. Finally, the RHS number is incremented using an SC instruction.

**Updating a field in a node**

At multiple points in our GME algorithm, a process needs to update a field in a node (e.g., a state flag or a pointer field), where the node is such that it was observed by the process to be the last node of the linked list at some point earlier (via a snapshot). In all cases except for one, the update should succeed only if the meta-data of the list has not changed since the snapshot was taken. The only exception is when the status flag of session state needs to be set; in this case, the update should succeed if the session associated with the *current incarnation* of the node is closed and vacant even if the node has been reused possibly multiple times since the snapshot was taken.

In the normal case, we proceed as follows. The process first reads the field (to be updated) using an LL instruction. It then verifies that the field still needs to be updated. It next verifies that no complete advance operation has been performed on the meta-data so far by ascertaining that the RHS number in the meta data matches the number stored in the snapshot. It finally modifies the field using an SC instruction.

In the exception case, we proceed as follows. The process first reads the session state using an LL instruction. It then verifies that the status flag is not yet set. It next verifies that the session is vacant by ascertaining that the session size is zero. It finally sets the status flag in the session state using an SC instruction.

We refer to the first type of update as *contextual update* and second type of update as *non-contextual update*.

**Hazard pointer**

As far as cleanup algorithm is concerned, the head field in the snapshot record of a process constitutes a hazard pointer. Thus, after a process takes a snapshot of a list’s meta-data,
it verifies that the list’s head pointer is still the same. If not, it discards the snapshot and restarts the snapshot algorithm.

### 3.2.2 Pseudocode of the algorithm

The pseudocode of the algorithm is given in Algorithms 2 to 6. In the pseudocode, symbols ‘|’ and ‘&’ denote bitwise-or and bitwise-and operators, respectively. The pseudocode is divided into five separate sections.

The first section, given in Algorithm 2, describes the data structures and the shared variables used in the algorithm. Any other variable used in the pseudocode, which is not listed as a shared variable, is either an input argument or a local variable of the method in which it is referenced. In the pseudocode, we have used arrays to store meta-data for multiple GME objects for convenience only. They can be replaced with any other mechanism that allows a process to access the meta-data of the linked list associated with a given GME object.

The second section, given in Algorithm 3, describes the code for entry and exit sections of the algorithm. The code for each section basically follows the steps outlined in Section 3.1.1. The code also includes steps for setting the appropriate flag in the session state at relevant places in lines 40, 43, 44, 58 and 63. The code uses several auxiliary methods, which are defined in the remaining three sections.

We categorize the auxiliary methods into three groups. The third, fourth and fifth sections describe methods that operate on session state, list meta-data and list node, respectively.

The third section, given in Algorithm 4, defines four methods: (a) IsOpen, which examines LEADERLESS and CONFLICT flags to determine if the session is open, (b) IsAdjudged, which examines VACANT flag to determine if the session is adjourned, (c) SetConditionFlag, which sets the given condition flag (to one) using the strategy for updating a node field described earlier, and (d) SetVacantFlag, which sets VACANT flag (to one).

The fourth section, given in Algorithm 5, defines three methods: (a) ReadHead, which takes an atomic snapshot of a list’s meta-data using the snapshot strategy described earlier, (b) TestHead, which ascertains that a list’s meta-data has not been updated since the snapshot was taken, (c) FixHead, which fixes a list’s partially updated meta-data, and (d) AdvanceHead, which updates a list’s entire meta-data. In TestHead, list’s meta-data is considered to be updated only if both the sequence numbers have been incremented.

The fifth section, given in Algorithm 6, defines four methods: (a) InitializeNewNode, which gets a free node (from a pool of unused nodes), initializes it and announces the request to other processes, (b) GetNextNode, which selects a node to append to the list, (c) AppendNextNode, which appends a new node to the list, and (d) ReclaimNode, which reclaims a node for later use. In AppendNextNode, after selecting a node to append, process first sets up the forward link (modify the next field of the current last node), then sets up the backward link (modify the previous field of the new last node) and finally advances the head. The next and previous fields are modified using the strategy for updating a node field described earlier. In ReclaimNode, all fields in the node are updated using simple write instructions. Note that all other processes trying to update a field in the node will either abort or fail.

### 3.3 Correctness Proof

The safety property uses the following three lemmas.
Lemma 1. A process starts executing its critical section as a follower only if the session it joins is compatible with its request and the session is still open after it incremented the session size.

Proof. After incrementing the session size, a process joins the session only after ascertaining that the meta-data of the linked list has not changed and the session is still open.

Lemma 2. If some process in a session has not decremented the session size, then the session cannot adjourn.

Proof. If a process enters its critical section as a leader, then the session size is incremented even before its node is appended to the list. If a process enters its critical section as a follower, then, from lemma 1, the session was open after the process incrementing the session size.

Clearly, when the session closes, the value of the session size is greater than or equal to the number of processes in the session that are executing their critical sections. And, no process sets the status flag in the session state until the session size reaches zero.

Lemma 3. No new node can be appended to the list until the session associated with the current last node has adjourned.

Proof. A process with a conflicting request spins until it detects that either a new session has been established or the current session has adjourned.

The above lemmas imply that, as long as a process is executing its critical section, the session cannot adjourn and no new node can be appended to the list. Further, only those processes whose request is compatible with the session can join the session. Thus we have

Theorem 1 (group mutual exclusion). The GME algorithm satisfies the group mutual exclusion property.

Lemma 4. The number of times the while-do loop in ReadHead method (lines 86 to 95) is executed is bounded by one plus the number of nodes appended to the list during the loop execution.

Proof. When taking a snapshot, if a process finds that the two sequence numbers do not match, it helps complete the advance operation before executing the next iteration of the while-do loop.

Lemma 5. Once a flag in a session state has been set, it is reset only after the session has adjourned and a new session has been established.

Proof. The flags in a session state are reset only when the associated node is reclaimed. A node can only be reclaimed after a new node has been appended to the list and the meta-data of the list has been updated.

Lemma 6. The while-do loop in SetConditionFlag method (lines 70 to 74) is executed only $O(1)$ times per invocation of the method.

Proof. A new iteration of the while-do loop is executed only if the $SC$ instruction performed on the session state fails. The failure occurs only if one of the flags in the session state has been set by another $SC$ instruction or all the flags would have been reset by a write instruction. The first case can only occur at most two times. The second case indirectly implies that the condition flag had been set earlier by another process. Thus a process executes at most three iterations of the while-do loop.
Note that the body of exit section (Exit method) includes up to one invocation of ReadHead, SetConditionFlag, ReclaimNode and SetVacantFlag methods. Only the first two methods contain a loop. The while-do loop in the ReadHead method is only executed once. Using the above lemmas, we can prove that

**Theorem 2 (bounded exit).** The GME algorithm satisfies the bounded exit property.

We now prove the concurrent entering property.

**Lemma 7.** An open session closes only if the system contains a conflicting request.

**Proof.** For a session to close, CONFLICT in the session state must be set, which can only be set by a process with a conflicting request.

**Lemma 8.** A session can adjourn only after it has closed.

**Proof.** For a session to adjourn, both the condition flags (LEADERLESS and CONFLICT) must be set in the session state. This implies that the session must be closed before it can be adjourned.

**Lemma 9.** If there is no conflicting request during the execution of the entry section, then the inner while-do loop in Enter method at lines 16 to 17 is executed only once.

**Proof.** The session stays open until a conflicting request is generated.

As part of joining a session as a follower, a process first increments the session size and then rechecks if the session is still open. If not, it decrements the session size immediately without executing its critical section. We refer to such an increment as spurious. Note that spurious increments may prevent a session from moving to adjourned state. However, the delay is finite because a process may spuriously increment the size of a session at most once. We prove that such spurious increments can only occur certain conditions.

We assign a unique incarnation number to a node every time it is used to establish a session. It is given by a tuple consisting of two entries: (i) the sequence number stored in the meta-data when the node is appended to the list, and (ii) the identifier of the GME object that the list is associated with. Additionally, every time a process increments the session size in a node, we assign a unique amendment number to the increment step. It is also given by a tuple consisting of two entries: (i) the sequence number stored in the process’ snapshot record, and (ii) the identifier of the GME object associated with the process’ request. We have:

**Lemma 10.** The amendment number of an increment step of a process matches the incarnation number of the node on which the step is performed.

**Proof.** If the two numbers do not match, then it implies that the node has been reused to establish another session. Let p, X and a denote the process, node and the amendment number in the lemma statement, respectively. Let q denote the new owner of X for the incarnation number immediately following a. As per our algorithm, before q reuses X it scans the snapshot record of every process. Let t denote the time at which q read the head field in p’s snapshot record as part of the cleanup algorithm. Clearly, at time t, q did not find the address of X stored in p’s snapshot record. Thus, we can infer that the head field of p’s snapshot record was set to the address of X at some time after t. Note that, since X was reclaimed before t, the sequence number in the meta data of the list is guaranteed to be strictly greater than that in a at time t and later. As per our algorithm, after recording the
snapshot and before performing the increment step, \( p \) checks to make sure that the meta-data in the list has not changed, which is guaranteed to fail. This implies that \( p \) will not perform the increment step with amendment number \( a \)—a contradiction.

The above lemmas imply that, in the absence of any conflicting request, all requests for the same GME object are fulfilled by a single node. Only in the beginning when the first such request is generated, the session already established at that point may need to be closed and adjourned, which only takes a bounded number of steps because there will be no spurious increments, and a new session be established by appending a new node to the list. Thus we have

\[ \textbf{Theorem 3 (concurrent entering).} \quad \text{The GME algorithm satisfies the concurrent entering property.} \]

For the lockout freedom property, we need the following additional lemmas.

\[ \textbf{Lemma 11.} \quad \text{If the system contains a conflicting request while a session is in progress, then the session eventually closes.} \]

\[ \textbf{Proof.} \quad \text{All processes with a conflicting request eventually invoke } \text{SetConditionFlag} \text{ method to set } \text{CONFLICT} \text{ in the session state, which terminates only after the flag has been set. Further, when the leader of the session leaves its critical section, it invokes } \text{SetConditionFlag} \text{ method to set } \text{LEADERLESS} \text{ in the session state, which terminates only after the flag has been set.} \]

\[ \textbf{Lemma 12.} \quad \text{Once a session is closed, its size can be incremented spuriously at most } n \text{ times.} \]

\[ \textbf{Proof.} \quad \text{Each process is responsible for at most one spurious increment to the session size.} \]

\[ \textbf{Lemma 13.} \quad \text{Once a session is closed, eventually the session size becomes zero and stays zero thereafter until a new session is established.} \]

\[ \textbf{Proof.} \quad \text{After a session closes, no new process can join the session. Every process that is in the session at the point the session closes eventually leaves the session.} \]

\[ \textbf{Lemma 14.} \quad \text{A closed session is eventually adjourned.} \]

\[ \textbf{Proof.} \quad \text{Whenever a process either sets one of the condition flags in the session state or decrements the session size, it attempts to set the status flag afterwards. The result then follows from lemmas 5 and 13.} \]

\[ \textbf{Lemma 15.} \quad \text{Once a session is adjourned, a new session is eventually established.} \]

\[ \textbf{Proof.} \quad \text{A session closes (and hence adjourns) only if there is a conflicting request in the system. Clearly, this implies that, after a session is adjourned, at least one process in the system tries to append a new node to the list (and establish a new session).} \]

\[ \textbf{Lemma 16.} \quad \text{After a process has announced its request, at most } n+1 \text{ nodes can be appended to the list until its request is fulfilled.} \]
Proof. Every time a new node is appended to the list and its meta-data updated, the sequence number in the meta-data (represented physically by two separate sequence numbers) increases by one. Let the value of the sequence number at the point when a process, say \( p_i \), announces its request be denoted by \( x \). Among the next \( n \) values, given by \( x+1, x+2, \ldots, x+n \), at least one value \( y \) is such that \((y \mod n) + 1 = i\). Clearly, when the sequence number value reaches \( y \), every process that tries to append a new node to the list chooses the node for \( p_i \) as the one to append. ◀

Finally, we have

▷ Theorem 4 (lockout freedom). The GME algorithm satisfies the lockout freedom property.

3.4 Complexity Analysis

▷ Theorem 5 (context switch complexity). At most \( n+1 \) sessions can be established while a process is waiting to enter its critical section.

Proof. The result follows from lemma [16] ▷

▷ Theorem 6 (concurrent entering step complexity). The maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding request for a different session during that period is \( O(1) \).

Proof. All outstanding requests in the system are for the same session (of the same GME object). If the current session of the GME object is not compatible with an outstanding request of a process, then (1) the session closes and then adjourns, (2) a new node is appended to the list, and (3) the meta-data of the associated linked list is updated. (in that order), all within \( O(1) \) steps of the process. Thereafter the process is able to join the session within \( O(1) \) of its own steps. Finally, a process is able to leave the session within \( O(1) \) of its own steps. ◀

As a corollary, the above theorem implies that

▷ Theorem 7 (solitary request step complexity). The maximum number of steps a process has to execute in its entry and exit sections provided no other process in the system has an outstanding request during that period is \( O(1) \).

▷ Theorem 8 (RMR step complexity). The maximum number of remote memory references required by a process in its entry and exit sections is \( O(n) \).

Proof. For every new node appended to the list, a process performs at most \( O(1) \) read or write instructions outside of the inner while-do loop in Enter at lines 46 to 47. While spinning in the loop, it reads the RHS sequence number in the meta-data and the session state in the node repeatedly. Any change in the RHS sequence number causes the process to quit the loop. This implies that reading the RHS sequence number repeatedly in the loop is responsible for only \( O(1) \) remote references per list node. The session state consists of four flags which, once set, are reset only after a new node is appended to the list. Thus reading the session state repeatedly in the loop is also responsible for only \( O(1) \) remote references per list node. The result then follows from lemma [16] ▷

▷ Theorem 9 (multi-object space complexity). The space complexity of our GME algorithm is \( O(m+n\ell) \) space, where \( n \) denotes the number of processes, \( m \) denotes the number of GME objects and \( \ell \) denotes the maximum number of locks a process needs to hold at the same time.
Proof. Our algorithm uses only $O(m + n)$ space for managing $m$ GME objects, where $O(n)$ space is shared among all $m$ GME objects. In addition, each process needs only $O(\ell)$ space, where $\ell$ denotes the maximum number of GME objects (or locks) a process needs to hold at the same time.

4 Optimization: Bounding Variable Values

The GME algorithm described in the previous section uses only two variables per GME object that can assume unbounded values, namely the sequence numbers that are part of the meta-data of the linked list associated with a GME object. The only operation performed on them is an increment by one. Note that, as argued in section 3.3, a process with an outstanding request can observe at most $n + 1$ consecutive values for sequence numbers after which the request is guaranteed to be fulfilled. Thus, if we use module $2n$ arithmetic for the increment operation in lines 111 and 117 then the resultant algorithm will still work correctly.

Note that, in addition to the sequence numbers, our algorithm uses pointer fields. The size of a pointer field can be easily bounded as follows. Clearly, the system needs at most $m(n + 1)$ distinct nodes (at most $n + 1$ nodes per GME object). Further, each node stores only a constant number of variables, including two pointer fields. All the non-pointer fields together need only $\log m + \log n + \log s + 4$ bits, where $s$ denotes the maximum number of distinct sessions for any GME object. It can be shown that it is sufficient for a pointer field to use only $O(\log m + \log n + \log s)$ bits to support $m(n + 1)$ distinct nodes.

5 Related Work

Several algorithms have been proposed to solve the GME problem for shared-memory systems in the last two decades [12, 14, 5, 21, 11, 3, 1, 6]. Most of the earlier algorithms use only read and write instructions whereas many of the later algorithms use atomic instructions as well. Different algorithms provide different fairness, concurrency and performance guarantees.

Many GME algorithms use a traditional or an abortable mutual exclusion (ME) algorithm as a subroutine. The GME algorithm proposed by Keane and Moir in 14 uses a traditional ME algorithm as an exclusive lock to protect access to entry and exit sections of the algorithm. As such, this algorithm does not satisfy bounded exit and concurrent entering properties. The GME algorithms presented in 3 1 use an abortable ME algorithm as a subroutine. The main idea is that a process can enter its critical section using multiple pathways: (i) as a “leader” by establishing a new session, or (ii) as a “follower” by joining an existing session. The first case occurs if the process is able to acquire the exclusive lock. The second case occurs if the process learns that a session “compatible” with its own request is already in progress in which case it aborts the ME algorithm and joins that session. Both pathways are explored concurrently and, as soon as one of them allows the process enter its critical section, the other one is abandoned.

5.1 Fairness and Concurrency Guarantees

In many (group) mutual exclusion algorithms, the entry section consists of two distinct subsections: a doorway and a waiting-room. A doorway is the wait-free portion of the entry section that a process can complete within a bounded number of its own steps. A waiting-room of the entry section is the portion where a process is blocked until it is its turn to execute its critical section.
We say that two active processes are fellow processes if they are requesting the same session (of the same GME object) and conflicting processes if they are requesting different sessions (of the same GME object).

We say that an active process \( p \) doorway-precedes another active process \( q \) if \( p \) completes the doorway before \( q \) enters the doorway. Besides the four properties listed in section 2, a GME algorithm may satisfy one or more of the properties listed below. These properties, which were defined in [5, 11, 1], describe additional guarantees that a GME algorithm may provide.

(P5) **Strong Concurrent Entering** If a process \( p \) has completed its doorway, and \( p \) doorway-precedes every active conflicting process, then \( p \) enters its critical section within a bounded number of its own steps.

(P6) **First-Come-First-Served (FCFS)** If \( p \) and \( q \) are two conflicting processes such that \( p \) doorway-precedes \( q \), then \( p \) enters its critical section before \( q \).

(P7) **Relaxed FCFS** If \( p \) and \( q \) are two conflicting processes such that \( p \) doorway-precedes \( q \) but \( q \) enters its critical section before \( p \), then there exists another process \( r \) whose current attempt overlaps with that of \( q \) such that \( q \) and \( r \) are fellow processes \( p \) does not doorway-precede \( r \).

(P8) **First-In-First-Enabled (FIFE)** If \( p \) and \( q \) are two fellow processes such that \( p \) doorway-precedes \( q \) and \( q \) enters its critical section before \( p \), then \( p \) can enter its critical section within a bounded number of its own steps.

(P9) **Pulling** Suppose \( p \) and \( q \) are two fellow processes such that \( p \) is currently in its critical section and doorway-precedes all conflicting processes. If \( q \) is currently in the waiting room, then \( q \) can enter its critical section within a bounded number of its own steps.

| Algorithm                      | P2 | P3 | P4 | P5 | P6 | P7 | P8 | P9 |
|-------------------------------|----|----|----|----|----|----|----|----|
| Joung [12]                    | ✔  | ✔  | ✔  | ✗  | ✗  | ✗  | ✗  | ✗  |
| Keane & Moir [14]             | ✔  | ✗  | ✗  | ✔  | ✗  | ✗  | ✗  | ✗  |
| Hadzilacos [5]                | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✗  | ✗  |
| Takamura & Igarashi [21, Algorithm 1] | ✗  | ✔  | ✗  | ✔  | ✗  | ✔  | ✗  | ✗  |
| Takamura & Igarashi [21, Algorithm 2] | ✔  | ✗  | ✗  | ✔  | ✗  | ✔  | ✗  | ✗  |
| Takamura & Igarashi [21, Algorithm 3] | ✔  | ✗  | ✗  | ✔  | ✗  | ✔  | ✗  | ✗  |
| Jayanti et al. [11, Algorithm 1] | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✗  | ✗  |
| Jayanti et al. [11, Algorithm 2] | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✗  | ✗  |
| Jayanti et al. [11, Algorithm 3] | ✔  | ✔  | ✔  | ✔  | ✔  | ✔  | ✗  | ✗  |
| Danek & Hadzilacos [3, Algorithm 1] | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✔  | ✗  |
| Danek & Hadzilacos [3, Algorithm 2] | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✔  | ✗  |
| Danek & Hadzilacos [3, Algorithm 3] | ✔  | ✔  | ✔  | ✔  | ✔  | ✔  | ✔  | ✗  |
| Bhatt & Huang [1]             | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✔  | ✗  |
| He et al. [6, Algorithm 1]    | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✔  | ✗  |
| He et al. [6, Algorithm 2]    | ✔  | ✔  | ✔  | ✗  | ✔  | ✔  | ✔  | ✗  |
| Our Algorithm [This Work]     | ✔  | ✔  | ✔  | ✔  | ✔  | ✔  | ✔  | ✗  |

**Table 1** Fairness and concurrency properties satisfied by different algorithms. Note that all algorithms satisfy P1.
Algorithm | Space Complexity | Solitary Request Step Complexity | Concurrent Entering Step Complexity | RMR Step Complexity
--- | --- | --- | --- | ---
Yang & Anderson’s Algorithm 1 | $O(n)$ | × | $O(\log n)$ | $O(\log n)$
Mellor-Crummey & Scott’s Algorithm | $O(1)$ | ✓ | $O(1)$ | $O(n)$

Table 2 Complexity measures for ME algorithms used by some GME algorithms.

Table 3 Complexity measures of GME algorithms excluding those in [11, 3] that use an abortable ME algorithm as a subroutine.

5.2 Complexity Measures

Tables 3 and 4 shows the complexity measures of different GME algorithms with respect to the metrics described in section 2.

Note that complexity measures for the algorithm in [14], which uses a traditional ME
### Table 4: Complexity measures of the GME algorithms in [11, 3] using the three abortable mutex algorithms.

| Algorithm | Abortable ME Algorithm | Space Complexity | Solitary Request Step Complexity | Concurrent Entering Step Complexity | RMR Step Complexity | Bounded Shared Variables |
|-----------|------------------------|------------------|----------------------------------|-------------------------------------|---------------------|--------------------------|
|           | Jayanti et al. [11, Algorithm 2] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✓ |
|           | Jayanti et al. [11, Algorithm 3] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 1] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✓ |
|           | Danek & Hadzilacos [11, Algorithm 2] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✓ |
|           | Danek & Hadzilacos [11, Algorithm 3] | $O(mn^2s)$ | $O(n \log s)$ | $O(n \log s)$ | $O(n \log s)$ | ✗ |
|           | Jayanti et al. [11, Algorithm 2] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Jayanti et al. [11, Algorithm 3] | $O(mn)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 1] | $O(m + n^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 2] | $O(m + n^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 3] | $O(mns)$ | $O(n \log s)$ | $O(n \log s)$ | $O(n \log s)$ | ✗ |
|           | Jayanti et al. [11, Algorithm 2] | $O(mn^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Jayanti et al. [11, Algorithm 3] | $O(mn)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 1] | $O(mn + n^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 2] | $O(mn + n^2)$ | $\Omega(n)$ | $\Omega(n)$ | $O(n)$ | ✗ |
|           | Danek & Hadzilacos [11, Algorithm 3] | $O(mns)$ | $O(\log s)$ | $O\left(\frac{\log s \times s}{\min\{\log n, k\}}\right)$ | $O\left(\frac{\log s \times s}{\min\{\log n, k\}}\right)$ | ✗ |

- $n$: number of processes
- $m$: number of GME objects
- $s$: number of different types of sessions
- $k$: point contention of the request

To analyze the performance of the Keane and Moir’s GME algorithm, we consider two different traditional ME algorithms, namely (i) a tree-based algorithm by Yang and Anderson [22], which has small RMR step complexity, and (ii) a queue-based algorithm by Mellor-Crummey and Scott [18], which has small space complexity as well as small solitary request step complexity. Both use bounded space variables. Table 2 displays the performance of the two ME algorithms with respect to various complexity measures.

To analyze the performance of a GME algorithm that uses an abortable ME algorithm as a subroutine, we consider the following abortable ME algorithms, namely (i) Bakery algorithm by Lamport [15] modified to support aborts [11], (ii) n-bit FCFS algorithm by Lamport [16], and (iii) an algorithm by Jayanti [11]. The second (middle) algorithm uses bounded space variables whereas the other two do not. Note that the six GME algorithms in [11, 3] can be combined with each of the three abortable ME algorithm to yield eighteen GME algorithms with potentially different complexity measures. For clarity, the complexity...
measures of these eighteen GME algorithms are given in table 41 and those of the remaining GME algorithms are given in table 3.

6 Conclusion and Future Work

In this work, we have presented four GME algorithms for an asynchronous shared memory system, each successively building on and addressing the limitations of the previous algorithm. Specifically, the last algorithm uses bounded space variables and satisfies the four most important properties of the GME problem, namely group mutual exclusion, lockout freedom, bounded exit and concurrent entering. At the same, it has $O(1)$ step-complexity for when the system has no conflicting requests, and $O(1)$ space-complexity per GME object when the system contains $\Omega(n)$ GME objects.

To the best of our knowledge, our algorithm is the first GME algorithm that has constant complexity for both cases. Finally, as is the case for most existing GME algorithms, our GME algorithm has $O(n)$ context switch complexity as well as $O(n)$ remote memory reference complexity.

As future work, we plan to extend our GME algorithm so that (i) it is adaptive in the sense its context switch and remote memory reference complexities depend on the actual contention rather than the maximum contention, and (ii) it provide stronger fairness and/or concurrency guarantees such as first-come-first-served (FCFS) 5, first-in-first-enabled (FIFE) 11, strong concurrent entry 11 and pulling 1 among others. We also plan to experimentally evaluate the performance of various GME algorithms under a variety of scenarios.

References

1 V. Bhatt and C. C. Huang. Group Mutual Exclusion in $O(\log n)$ RMR. In Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC), pages 45–54, July 2010.
2 T. S. Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical report, Department of Computer Science, University of Washington, 1993.
3 R. Danek and V. Hadzilacos. Local-Spin Group Mutual Exclusion Algorithms. In Proceedings of the 18th Symposium on Distributed Computing (DISC), pages 71–85, October 2004.
4 F. E. Fich, D. Hendler, and N. Shavit. On the Inherent Weakness of Conditional Primitives. Distributed Computing (DC), 18(4):267–277, 2006.
5 V. Hadzilacos. A Note on Group Mutual Exclusion. In Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC), August 2001.
6 Y. He, K. Gopalakrishnan, and E. Gafni. Group Mutual Exclusion in Linear Time and Space. In Proceedings of the 17th International Conference on Distributed Computing And Networking (ICDCN), January 2016.
7 M. Herlihy. Wait-Free Synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1):124–149, January 1991.
8 M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. ACM Transactions on Programming Languages and Systems (TOPLAS), 15(5):745–770, 1993.
9 M. Herlihy and N. Shavit. The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann, 2012.
10 P. Jayanti. Adaptive and Efficient Abortable Mutual Exclusion. In Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC), pages 295–304, July 2003.
11 P. Jayanti, S. Petrovic, and K. Tan. Fair Group Mutual Exclusion. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, July 2003.
12 Y.-J. Joung. Asynchronous Group Mutual Exclusion. *Distributed Computing (DC)*, 13(4):189–206, 2000.
13 Y.-J. Joung. The Congenial Talking Philosophers Problem in Computer Networks. *Distributed Computing (DC)*, pages 155–175, 2002.
14 P. Keane and M. Moir. A Simple Local-Spin Group Mutual Exclusion Algorithm. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 23–32, 1999.
15 L. Lamport. A New Solution of Dijkstra’s Concurrent Programming Problem. *Communications of the ACM (CACM)*, 18(8):453–455, August 1974.
16 L. Lamport. The Mutual Exclusion Problem: Parts I and II. *Journal of the ACM (JACM)*, 33(2):313–348, 1986.
17 P. Magnussen, A. Landin, and E. Hagersten. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the International Parallel and Processing Symposium (IPPS)*, pages 165–171. ACM Press, April 1994.
18 J. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
19 M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(6):491–504, 2004.
20 K. Platz. *Saturation in Lock-Based Concurrent Data Structures*. PhD thesis, Department of Computer Science, The University of Texas at Dallas, 2017.
21 M. Takamura and Y. Igarashi. Group Mutual Exclusion Algorithms Based on Ticket Orders. In *Proceedings of the Annual International Conference on Computing and Combinatorics (COCOON)*, pages 232–241, July 2003.
22 J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing (DC)*, 9(1):51–60, 1995.