Group Mutual Exclusion in O(log n) RMR

Vibhor Bhatt
Dartmouth College
Hanover, NH 03755
U.S.A.
vibhor@cs.dartmouth.edu

Chien-Chung Huang[*]
Max-Planck-Institut für Informatik
66123, Saarbrücken
Germany
villars@mpi-inf.mpg.de

ABSTRACT

We present an algorithm to solve the GROUP MUTUAL EXCLUSION problem in the cache-coherent (CC) model. For the same problem in the distributed shared memory (DSM) model, Danek and Hadzilacos presented algorithms of $O(n)$ remote memory references (RMR) and proved a matching lower bound, where n is the number of processes. We show that in the CC model, using registers and LL/SC variables, our algorithm achieves $O(min(\log n, k))$ RMR, where k is the point contention, which is so far the best. Moreover, given a recent result of Attiya, Hendler and Woelfel showing that exclusion problems have a $\Omega(\log n)$ RMR lower bound using registers, comparison primitives and LL/SC variables, our algorithm thus achieves the best theoretical bound.

Categories and Subject Descriptors

D [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms: Algorithms, Design, Performance, Theory

Keywords: Mutual Exclusion, Group Mutual Exclusion, Remote Memory Reference (RMR), Synchronization

1. INTRODUCTION

MUTUAL EXCLUSION is a classical problem in distributed computing introduced by Dijkstra in 1965 [12]. In MUTUAL EXCLUSION, processes repeatedly cycle through four sections of code–REMAINDER SECTION, TRYING SECTION, CRITICAL SECTION (CS), and EXIT SECTION, and the problem consists of designing the code for the TRYING SECTION and the EXIT SECTION such that no two processes are in the CS as the same time (Mutual Exclusion Property). To ensure liveness, it is often required to satisfy an additional property called starvation-freedom; if no process stays in the CS forever, then any process which enters the TRYING SECTION eventually enters the CS.

The GROUP MUTUAL EXCLUSION problem (GME), introduced by Joung [22], generalizes MUTUAL EXCLUSION. In GME, n processes request sessions and demand to enter the CRITICAL SECTION (CS). Only processes requesting the same sessions (belonging to the same "group") are allowed to be in the CS at the same time. More precisely, in GME, each process cycles through the following four sections: REMAINDER SECTION, TRYING SECTION, CRITICAL SECTION (CS), and EXIT SECTION. When in the REMAINDER SECTION, a process picks a session number (possibly different each time), and then executes the other three sections in order. The execution of these three sections is an *attempt*. Once an attempt is finished, the process again is in the REMAINDER SECTION. A process is called *active* if it is in one of its attempts. Two active processes are in *conflict* if their sessions differ, otherwise, they are *fellow* processes. We assume that processes crash only in the REMAINDER SECTION

An algorithm purporting to solve the GME problem should design codes for the TRYING SECTION and the EXIT SECTION so as to satisfy the following properties:

(P1). Mutual Exclusion: If two active processes are in the CS at the same time, then they are fellow processes.

(P2). Starvation Freedom: If no process stays in the CS forever, then any process that enters the TRYING SECTION eventually enters the CS.

(P3). Bounded Exit: If a process enters the EXIT SECTION, it enters the REMAINDER SECTION within a bounded number of its own steps.

(P4.) Concurrent Entering: There exists a bound b given by the algorithm, such that, a process in the TRYING SECTION takes at most b steps in the absence of conflicting processes before entering CS.

The property of concurrent entering, informally stated by Joung [22], and made precise by Hadzilacos [15], is a critical property of GME. Without it, any ordinary mutual exclusion algorithm solves GME easily. Its intent is to ensure concurrency, that is, active processes of the same session, in the absence of other conflicting processes, should be able to enter the CS without unnecessary synchronization among themselves.

It is natural to aim for designing "fast" exclusion (MUTUAL EXCLUSION, GME) algorithms. However, for these algorithms, it is not proper to measure the time complexity simply by the number of variable accesses. Researchers have reached the consensus that a more suitable measurement of complexity is the the number of *remote memory references (RMRs)* (see the survey [4]). In *Distributed Shared Memory* (DSM) machines, a reference to a shared variable X is considered remote if X is at a memory module of a different processor; in *Cache Coherent* (CC) machines, a reference to X by a process p is considered remote if X is not in p's cache.

[*]Research supported by an Alexander von Humboldt fellowship.

Research in exclusion algorithms and RMRs has led to design of many algorithms [1, 2, 5, 6, 10, 14, 17, 19, 25, 26, 29], lower bounds [3, 7, 11, 13], and an understanding of the limitations of various shared memory primitives [7, 11].

For GME, in the DSM model, Danek and Hadzilacos [16] presented two $O(n)$ algorithms and showed a matching lower bound. Remarkably, in the CC model, they presented an $O(\log n \log m)$ RMR algorithm, where m is the total number of possible sessions, thereby demonstrating a remarkable complexity separation for these two models. Is it possible to improve on this $O(\log n \log m)$ upper bound? And if it is, by how much?
In a recent paper [7], Attiya, Hendler and Woelfel proved a $\Omega(\log n)$ RMR lower bound for the mutual exclusion problem in both DSM and CC models, under the assumption that the algorithms use registers, comparison primitives, *LL*/*SC* variables. For the ordinary mutual exclusion problem, Yang and Anderson [29] gave an $O(\log n)$ algorithm, hence, the best theoretical upper and lower bounds match in mutual exclusion. As mutual exclusion is a special case of GME, we know that even in the CC model, the lower bound of GME is $\Omega(\log n)$[1]. A natural question is whether it is possible to achieve this RMR for GME?

Our Main Contribution

We give a positive answer to the above question by presenting an $O(\min(\log n, k))$ algorithm for GME in the CC model, where k is the point contention, thereby achieving the best possible theoretical bound. Our algorithm uses registers and *load-link* (*LL*)/ *store-conditional* (*SC*) variables. Besides our present algorithm, to our knowledge, the only algorithms achieving sublinear (in n) RMR are Danek and Hadzilacos' $O(\log n \log m)$ algorithm and Keane and Moir's $O(\log n)$ algorithm [23]. The former algorithm relies on the additional assumption that a failed compare-and-swap operation does not invalidate the local cached copies in the processes. We do not make this assumption. The latter algorithm, unfortunately, fails to satisfy concurrent entering, but instead satisfies a weaker property, called *concurrent occupancy*.[2]

Besides the basic properties P1-P4, our algorithm also satisfies other properties that we believe to be natural and intuitive. These additional properties are presented in the next section.

Fairness and Concurrency in GME

In GME, fairness and concurrency are two desirable, but unfortunately, incompatible, goals. On the one hand, we wish to enforce a certain order, that is "fair," among the conflicting processes to enter the CS; on the other hand, to enhance concurrency, under certain circumstances, we should allow processes bypass other conflicting processes to enter the CS without waiting. Obviously, whether fairness or concurrency should be more emphasized depends on the application. Previous work [15, 21] have stressed more on the side of fairness. In this work, we will instead emphasize concurrency, while still trying to maintaining a reasonable degree of fairness.

We first review how fairness is defined in the literature. The TRYING SECTION is conventionally divided into two parts: Doorway and Waiting Room. A process can execute the Doorway in a bounded number of its own steps. The time interval of a process executing its doorway then is used to impose a partial order among all active processes.

DEFINITION 1. *Let p and q be two active processes. If in p's current attempt, p finishes its doorway before q's current attempt begins its doorway, then p doorway-precedes q.*

The following fairness property for GME is formulated by Hadzilacos [15], which is generalized from a fairness property introduced by Lamport [24] for the original mutual exclusion problem.

(P5). FCFS (first-come-first-served): If process p doorway-precedes process q and the two processes request different sessions, then q does not enter the CS before p.

Previous works [15, 16, 21] have designed algorithms satisfying P5. However, for the sake of enhancing concurrency, we will not try to satisfy P5. Imagine the following scenario. A process p is in the CS with session s and another process q with a conflicting session s' is still in the Waiting Room. Furthermore, q doorway-precedes a large number of processes requesting session s. By P5, these latter processes cannot enter the CS before q (and q also has to wait until p leaves the CS, due to P1). For the sake of concurrency, it may be argued whether it is more desirable to let these late processes of session s to bypass q and enter the CS directly—while p is still in the CS.

(P6). Pulling: Suppose that process p requests session s and is in the CS at time t. Furthermore, p doorway-precedes all conflicting processes. If a process q also requests session s and is in the waiting room at t, then, q can enter the CS in a bounded number of its own steps.

The intuition here is that if a process doorway-precedes all conflicting processes, then its session has a clear precedence over all other sessions. Then its presence there in the CS should justify its fellow processes to join the CS directly.

Unlike the fairness requirement, which has been formulated as P5, the concurrency requirement is never precisely defined in the literature. Earlier works [16, 22, 23] have proposed the so-called "capturing mechanisms" to boost concurrency. Roughly speaking, these mechanisms allow a process to check whether other active processes are its fellow processes before it enters the CS itself, and if so, to help them enter. These mechanisms, however, do not fulfill any particular exact concurrency property, and in our context, also do not help to satisfy P6.

We remark that P6 is obviously incompatible with P5.[3] In view of this incompatibility, it would be desirable to relax P5 so that a reasonable property of fairness can still be maintained.

(P5'). Relaxed FCFS: If processes p and q request different sessions and p doorway-precedes q, then q does not enter the CS before p, unless there exists another active process r, whose current attempt overlaps with q's current attempt and which is a fellow process of q, and r is not doorway-preceded by p.

The intuition behind P5' is the following: even if process q is doorway-preceded by p, as long as q has some fellow process which

[1]More precisely, we note that this lower bound holds for the case that $m \geq n$. It is still open whether it is possible to beat this lower bound when m is small.

[2]Concurrent occupancy states the following: if a process p requests a session and no process requests a different session, then p eventually enters the CS (even if other processes do not leave the CS). For the subtlety regarding the definitions of concurrent entering/occupancy, we refer the readers to [15].

[3]We note that P5 is also incompatible with another property formulated by Jayanti et al. [21], called *strong concurrent entering*: if process p doorway-precedes all active processes that request a different session, then p enters the CS within a bounded number of its own steps. If an algorithm satisfies P6, one can easily construct scenarios where it violates strong concurrent entering.

	RMR (CC model)	P4	P5	P5’	P6	Hardware used
Joung[22]	∞	Yes	No	No	No	Read/Write
Keane & Moir [23]	$O(\log n)$	No	No	No	No	Read/Write
Hadzilacos [15]	$O(n)$	Yes	Yes	Yes	No	Read/Write
Jayanti & Petrovic & Tan [21]	$O(n)$	Yes	Yes	Yes	No	Read/Write
Danek & Hadzilacos [16] (“fair”)	$O(n)$	Yes	Yes	Yes	No	CAS, fetch&add
Danek & Hadzilacos [16] (“highly-concurrent”)	$O(n)$	Yes	No	Yes	No	CAS, fetch&add
Danek & Hadzilacos [16]	$O(\log n \log m)$	Yes	No	No	No	CAS, fetch&add
This work	$O(\min(\log n, k))$	Yes	No	Yes	Yes	CAS

Table 1: Summary of results. All algorithms satisfy P1, P2, and P3.

is not doorway-preceded by p, then q has *some* justification in by-passing p to enter the CS first. On the other hand, if at the time when q begins its doorway, not only q, but also all its active fellow processes are doorway-preceded by p, then q has no claim at all in by-passing p.

Observe that P5’ reduces to the original FCFS of Lamport in the context of ordinary mutual exclusion (i.e., when every process has its own unique session). Note that the highly concurrent algorithm of Danek and Hadzilacos [16] also satisfies P5’. We now summarize all the specifics of our algorithm.

THEOREM 1. *In the cache coherent model, suppose that the total number of sessions is a bounded number m. We can design a group mutual exclusion algorithm, shown in Figure 3, that satisfies (P1)-(P4), (P5’), and (P6). In this algorithm, each attempt of a process takes $O(\min(\log n, k))$ RMR, where n is the number of processes and k is the point contention. The algorithm uses $O(mn)$ number of shared variables that support* LL/SC *and read/write operations.*

Hardware Support

We assume that the hardware provides variables supporting read/write and *LL/SC* operations. The definition of *LL/SC* operations are given as follows:

- The operation *LL*($\mathcal{O}$) returns $\mathcal{O}$’s value.
- The operation *SC*($\mathcal{O}, v$) by a process “succeeds” if and only if no process has performed a successful *SC* on $\mathcal{O}$ since p’s latest *LL* operation on $\mathcal{O}$. If *SC*($\mathcal{O}, v$) succeeds, $\mathcal{O}$ is changed to v and returns true, otherwise $\mathcal{O}$’s value remains unchanged.

There are many practical implementations of *LL/SC* using compare&swap [20, 27, 28]. As a result our algorithm can be run on all modern architectures using *compare&swap*. In comparison the algorithms by Danek and Hadzilacos required both *compare&swap* and *fetch&add* [16].

Our algorithm makes use of some powerful objects such as the a priority process-queue and the counters. A priority process-queue supports *enqueue*, *dequeue* and *find-min* (*Min*) operations. A counter supports read and increment (inc) operations. These objects are implemented using the f-array introduced by Jayanti [18]. The following two theorems from [18] state the efficiency of implementing these objects using *LL/SC*.

THEOREM 2. *A linearizable, wait-free implementation of a counter, shared by n processes, is possible from registers and* LL/SC *words. The time complexity of read is $O(1)$ and the time complexity of $inc(d)$ is $O(\min(\log n, k))$, where k is the point contention during the increment operation. The space complexity is $O(n)$.*

THEOREM 3. *A linearizable, wait-free implementation of a priority process-queue, shared by n processes, is possible from registers and* LL/SC *words. The time complexity of Min is $O(1)$ and of a matching pair of operations–*enqueue *followed by* dequeue*—is $O(\min(\log n, k))$, where k is the point-contention during the execution of enqueue. The space complexity is $O(n)$.*

Related Work

Table 1 summarizes the various properties of the GME algorithms for the CC model in the literature. For the DSM model, Danek & Hadzilacos showed that a 2-session local spin GME algorithm satisfying P1-P4 requires $\Omega(n)$ RMR. They also gave the first GME algorithm to achieve $O(n)$ RMR in this model [16].

In the next two sections, we present our algorithm and give a sketch of the proof.

2. THE ALGORITHM

We begin by explaining the high-level ideas of our algorithm. We allow each session s to have two “sides”: side 0 and side 1. Processes of session s, in the TRYING SECTION, should decide which side it should join. The CS, at any point of time, belongs to a session and to one of its sides. If the current session of CS is s and is of side 0, only those processes of s who choose to join side 0 can get into the CS, while those of s who choose to join side 1 should wait. When all of the former have left the CS, if they find some conflicting processes of session s' (of whichever side) are waiting, some of them should take the responsibility of switching the current session in the CS to s' (with the proper side) and notifying those waiting processes of s' to enter.

The reason of introducing two different sides for the same session is, roughly speaking, to enforce certain liveness and fairness properties. If the ongoing session s is of side 0 and there are some conflicting processes that have been waiting long enough, then the new processes of session s should not join side 0. Instead, they should join side 1 and wait until next time.

How do we decide the relative order among all active processes (that is, the timing of their entering the TRYING SECTION)? And how do processes find out which active process has waited the longest and should enter the CS next? For the first question, we can use the notion of time-stamps. We have a global counter and every entering process increases it by 1, and then reads its value. This value serves as its time-stamp for this attempt. The second question is trickier. To find out the longest waiting process, a process can check the time-stamps of all processes, but this might entail $\Omega(n)$ RMR if implemented naively.

We make use of the following two objects in our algorithm, both of which can be implemented using the f*-array* introduced by Jayanti [18].

- Global Counter G: a linearizable counter that supports inc

```
Gmutex-Lock (s)
REMAINDER SECTION
1  d ←Inc-Get-Dir (s)
2  inc (G, 1)
3  tm ← Read(G)
4  Enqueue(Q, [p, s, d, tm])
5  Promote(s)
6  wait till Out[s] = d
CRITICAL SECTION
7  Dequeue(Q, [p, s, d, tm])
8  if Min(Q) ≢ (s, −)
9      In[s] ← $\overline{d}$
10 inc(C[s, d], −1)
11 Promote(s)
```

```
Inc-Get-dir(s)
1.1 inc(C[s, 0], 1)
1.2 inc(C[s, 1], 1)
1.3 d ← In[s]
1.4 inc(C[s, $\overline{d}$], −1)
1.5 return d
```

```
Promote(s)
5.1 $(s^c, d^c) \leftarrow LL(\mathcal{S})$
5.2 $(s^h, d^h) \leftarrow Min(Q)$
5.3 if $C[s^c, d^c] = 0 \wedge s^h \notin \{s^c, \perp\}$
5.4     if $SC(\mathcal{S}, (s^h, d^h))$
5.5         $Out[s^h] \leftarrow d^h$
```

Figure 1: First Attempt

(G, d) and read(G) operations. The former increases the current value of G by the amount of d while the latter returns the current value of G. Its initial value is 0.

- Priority process-queue Q: a data structure supporting enqueue, *Min* and dequeue operations, with the following restrictions: (i) an element can be dequeued only by the process that enqueued it, and (ii) after a process p enqueues an element, p must dequeue it before enqueuing a new one. The $Min(Q)$ operation returns the "smallest" element (whose meaning will be explained immediately) in Q. Initially Q is empty.

  In our algorithm, a process p in the TRYING SECTION enqueues the element $[p, s, d, tm]$, where s and d are its session number and the side it decides to join, and tm its time-stamp. Given two elements, $[p, s, d, tm]$ and $[p', s', d', tm']$, the former is smaller if either (1) $tm < tm'$, or (2) $tm = tm'$ and $p < p'$. The $Min(Q)$ operation then returns the pair (s, d), which is drawn from the smallest element $[p, s, d, tm]$. If there is no element in Q, $Min(Q)$ operation returns $(\perp, \perp)$.

We note that the $inc(G, d)$ and the three operations of Q take $O(\min(\log n, k))$ RMR, which thus form the bottleneck of our algorithm. In the following discussion, the process whose element is the smallest in Q is the *highest priority process* and its intended session is the *highest priority session*. Note that since both G and Q are implemented using f-arrays, they require *LL/SC* variables.

Besides the above two objects, we also use the following shared variables:

- $\mathcal{S} = (s, d)$, recording the ongoing session information in the CS. The first parameter indicates the session number s and the second its side d. This is a *LL/SC* variable. Initially $\mathcal{S} = (s', 0)$, where s' is some arbitrary session.

- $In[s]$, a binary register. It tells the new coming processes of session s which side of s they should join. Initially, $In[s] = 1$ for all sessions s.

- $Out[s]$, a binary register. It serves the purpose of the spinning lock, i.e., the processes of session s, immediately before entering the CS, should busywait on this variable. Initially, $Out[s] = 0$ for all sessions s.

- $C[s, 0]$, $C[s, 1]$, two counters, implemented using f-arrays. They indicate the number of processes of session s which are currently in the side 0 and side 1 respectively. Initially, $C[s, 0]$=$C[s, 1] = 0$ for all sessions s.

We have introduced all of the necessary ingredients. We now explain how to design algorithms based on them.

2.1 First Attempt

We first present a simple, though incorrect (violating P2 and P4), algorithm. It nonetheless contains many of our basic ideas. See Figure 1 for the code of our first algorithm, which shares similar ideas of the mutual exclusion algorithms of Jayanti [18, 21]. We give an informal description here. Once a process p of session s leaves the REMAINDER SECTION, it invokes the `Inc-Get-Dir` procedure to decide which side of s it should join. Inside this procedure, it first indiscriminately joins both sides (Lines 1.1 and 1.2). Then it checks the $In[s]$ variable to find out which side is the "right" side. Then it removes itself from the wrong side (Line 1.4) and returns.

Once process p decides which side of s to join, it increases the global counter (Line 2), gets a time-stamp (Line 3), and enqueues itself in the queue Q (Line 4). Then p invokes the `Promote` procedure. This procedure serves the purpose of updating the ongoing session in the CS if necessary. It is invoked in both the TRYING SECTION and the EXIT SECTION (Lines 5 and 12). In `Promote`, p checks whether the current session in the CS is "over" by looking at the counter $C[s^c, d^c]$ (Lines 5.1 and 5.3). If it is, then p finds out the highest priority session s^h (Line 5.2). If s^h is different from the current session s^c, p performs an *SC* operation on $\mathcal{S}$ to change it to (s^h, d^h), where d^h is the side that the highest priority process has decided to join. If p succeeds, it flips the $Out[s^h]$ variable to d^h.

Lines 1-5 constitute the doorway. Note that if a process p doorway-precedes another process q, the time-stamp it gets on Line 2 is bound to be smaller than the one q gets. After process p has finished the doorway, it busywaits on the $Out[s]$ variable until it is flipped to its own side.

Upon leaving the CS, process p removes itself from the queue Q (Line 7). At this point, if it discovers that its session s no longer has the highest priority, it flips $In[s]$ to the opposite side (of its own current side) (Lines 8-9). This flipping informs the new processes of session s that they should join the other side (See Line 1.3), and wait until the next time (that the session in CS is switched back to s again). Finally, process p decreases the counter of its session (Line 10) to announce it is almost done for this attempt. Then it again invokes the `Promote` procedure to switch the session in the CS if necessary.

Before we explain why this algorithm fails to satisfy P2 and P4, we point out a few of its key ideas, which will be reused and expanded in our later algorithms.

1. When a session s is over and the current highest priority session is $s' \neq s$, a number of processes may try concurrently to change the session in `Promote`. Due to the fact that they employ *LL/SC* operations on the variable $\mathcal{S}$, *only one of them* will succeed; moreover, not until it has updated the variable $Out[s']$ (Line 5.5) can those processes busywaiting on it enter the CS.
2. We maintain the following invariant: processes of session s who decide to join side d can be in the CS if and only if (1) $\mathcal{S} = (s, d)$, and (2) $Out[s] = d$.[4]

[4]We will not prove this invariant here, since this algorithm is more for illustration purpose, but the correctness of this invariant is easy to verify.

3. Once the current session s recorded in $\mathcal{S}$ no longer has the highest priority, one of the leaving processes will flip the $In[s]$ variable to the other side to prevent the current session s from going on forever (since the late processes of s will note this information in 1.3). This helps us to achieve P2 (not entirely) and P5'.

4. Suppose the current session of $\mathcal{S}$ is s and is of side d. As long as the highest-priority process p from session s remains in the CS, no leaving processes will flip $In[s]$ to $\overline{d}$. Thus, the new processes of s will choose to join side d (Line 1.3) and enter the CS directly. This helps us to achieve P6.

We now explain the trickier scenarios that cause our first algorithm to fail.

First Failure. Suppose that the current session is s' in $\mathcal{S}$ and there is no active process. Then a number of processes leave the REMAINDER SECTION and choose to join session s. They then all choose to join side 1, get time-stamps, and enqueue themselves into Q and execute `Promote`. Since they are the only active processes, they will notice that the current session s' is over and they will try to update it into s. However, only one of them, say process p, will succeed in updating $\mathcal{S}$ and only process p is allowed to update the $Out[s]$ variable. All the fellow processes of p will spin on $Out[s]$. If p stops taking steps (Line 5.5), then its fellow processes cannot enter the CS in their own steps. This scenario shows why our algorithm fails to satisfy concurrent entering.

Second Failure. Suppose that a process p of session s and of side 0 is in the CS and no other processes are active. Then p leaves the CS and dequeues itself from Q. As now Q is empty, p notices this fact and flips $In[s]$ to 1. At this point, p stops taking steps and a new process p', also of session s, enters the doorway, executes the lines 1.1 up to 1.3, and thus decides to join side 1. Now process p resumes taking steps and it will not try to update the session variable $\mathcal{S}$, since $C[s,0]=1$. So p simply returns to the REMAINDER SECTION. Then process p' starts taking steps, and when it executes `Promote`, it also will not attempt to update $\mathcal{S}$ in `Promote`, since the current highest priority session is still s (the only element in Q being enqueued by p' itself), and then p' just busywaits at Line 6 and starve.

How to modify our first algorithm to achieve P2 and P4 turns out to be a challenging task. For the two particular scenarios that we described, there can be a number of ways of fixing them. However, in our experience, many of these fixes are problematic, causing some unwanted new errors to arise. Here is a simple idea that tries to fix the second failure but unfortunately does not work.

Suppose that we change Line 8 as follows:

if $Min(Q) \notin \{(s,-),(\bot,\bot)\}$ **then**.

This modification is based on the following (spurious) reasoning: When the leaving process (of session s) notes that the queue Q is empty, it may as well leave the $In[s]$ as it is (the same side as its current side), instead of flipping it, since the new processes of s then can join its current side and go straight to the CS. Indeed, one can observe that this change in Line 8 fixes the problematic scenario we just described in the second failure.

However, this modification gives rise to another problematic scenario. Suppose that the leaving process (which is the only active process at this point) does not flip $In[s]$ when detecting Q to be empty. The process then executes `Promote` without modifying the session variable $\mathcal{S}$ and quits. At this point, $In[s] = Out[s]$. Suppose now process q of session s' enters the doorway and keeps on taking steps. It is easy to see that it will end up in the CS. Now suppose another process p' of session s enters the doorway and keeps on taking steps. Then p' also ends up in the CS, violating mutual exclusion.

```
Gmutex-Lock ($s$)
REMAINDER SECTION
1  $d \leftarrow$ Inc-Get-Dir ($s$)
2  $inc(G,1)$
3  $tm \leftarrow$ Read$(G)$
4  Enqueue$(Q,[p,s,d,tm])$
5  Promote($s$)
6  if $LL(\mathcal{S}) = (s,d,s)$
7      Out$[s] \leftarrow d$
8  wait till $Out[s] = d$
CRITICAL SECTION
9  Dequeue$(Q,[p,s,d,tm])$
10 if $Min(Q) \neq (s,-)$
11     $In[s] \leftarrow \overline{d}$
12 $inc(C[s,d],-1)$
13 Promote($s$)
```

```
Inc-Get-dir($s$)
1.1 $inc(C[s,0],1)$
1.2 $inc(C[s,1],1)$
1.3 $d \leftarrow In[s]$
1.4 if $Min(Q) \in \{(s,-),(\bot,\bot)\}$
1.5     if $LL(\mathcal{S}) = (s,\overline{d},-)$
1.6         $d \leftarrow \overline{d}$
1.7 $inc(C[s,\overline{d}],-1)$
1.8 return $d$
```

```
Promote($s$)
5.1 $(s^c,d^c,-) \leftarrow LL(\mathcal{S})$
5.2 $(s^h,d^h) \leftarrow Min(Q)$
5.3 if $C[s^c,d^c] = 0 \wedge s^h \notin \{s^c,\bot\}$
5.4     if $SC(\mathcal{S},(s^h,d^h,s))$
5.5         Out$[s^h] \leftarrow d^h$
```

Figure 2: Second Attempt

2.2 Second Attempt

Our second algorithm can be found in Figure 2. Its main difference from the first one is the addition of Lines 6-7 and 1.4-1.6. We ask the readers to ignore Lines 1.4-1.6 temporarily (as if they did not exist).

Recall that in the scenario described in the first failure, the problem is that the process p which successfully updates the variable $\mathcal{S}$ to s may fall asleep before updating $Out[s]$. This situation is allowed if p itself is of a different session $s' \neq s$. But if p is also of session s, we run the risk of violating concurrent entering. A natural thing to try is the following: if other processes of session s find out that $\mathcal{S}$ is last written by one of their fellow processes, they should be allowed to update $Out[s]$ themselves, just in case that that fellow process who successfully updated $\mathcal{S}$ has been slacking at Line 5.5.

The above rationale suggests the following modification to the session variable $\mathcal{S}$: we include a third component, which records the session of the process which lastly successfully updates $\mathcal{S}$ (See Line 5.4). It should be clear that Lines 6-7 implement the intuition we gave in the last paragraph, resolving the problematic scenario in the first failure.

We now turn to Lines 1.4-1.6. Originally we used $In[s]$ as an absolute indicator of the side that a new process of session s should join. Due to the second failure, we should allow certain exceptions to this rule; when the new process executing the `Inc-Get-Dir` procedure belongs to the current session recorded in $\mathcal{S}$. Especially, in that scenario, things go wrong when (1) Q is currently empty, (2) $\mathcal{S} = (s,d)$, and (3) $In[s] = \overline{d}$. When all the three conditions hold, the system is in an ambiguous stage, in the sense that the current session may or may not be about to change. Therefore, in this case, the new process of s executing `Inc-Get-Dir` should use extra information when deciding the side it is to join instead of only looking at the variable $In[s]$. Lines 1.4-1.6 are designed based on this idea, and it can be easily shown that the problematic scenario in the second failure will not arise now.

There exists another scenario in the first algorithm that processes starve: (1) the smallest element in Q is of session s, (2) $\mathcal{S} = (s,d)$, and (3) $In[s] = \overline{d}$, when the new process of s executes `Inc-Get-Dir`. This scenario, though more complicated, is not difficult to construct. Because of this scenario, in Line 1.4, we also include the condition $\mathit{Min}(Q) = (s, -)$.

The modification we have done so far help to achieve P2. However, concurrent entering turns out to be a far more difficult property to satisfy. The addition of Lines 6-7 and the third component of $\mathcal{S}$ only fix the case that we described in the first failure: when a set of new processes enter the doorway and they all belong to the same session. We now describe a scenario that reveals the genuine difficulty of achieving concurrent entering.

Third Failure Suppose that process p of session s is in the CS and a conflicting process q of session s' is busywaiting at Line 8. Process p then leaves the CS and keeps on taking steps until it is poised at 5.4 (to write s' into $\mathcal{S}$). Now a new process q', also of session s', enters the doorway and keeps on taking steps until it successfully executes Line 5.4 and then it falls asleep before it executes Line 5.5. Then p proceeds to execute Line 5.4 and its *SC* operation is bound to fail, since $\mathcal{S}$ has been changed by q' since p's last *LL* operation on it. Thus p quits the EXIT SECTION. q and q' are now the only active processes and they both belong to session s'. As long as q' does not execute Line 5.5, then q cannot enter the CS in its own steps. Thus concurrent entering is violated.[5]

As with the first two failures, it is not difficult to slightly twist our second algorithm to avoid the above scenario. But half-measures, in our experience, fix one problem and cause other problems to pop up. To achieve our goal, we need more radical changes to our algorithm.

2.3 Final Algorithm

The scenario in the third failure illustrates the difficulty we face. When a session is over, both the processes in the TRYING SECTION and in the EXIT SECTION will try to update it in `Promote` (and this feature cannot be forfeited, otherwise, we have troubles satisfying starvation freedom). Things are fine if the leaving process "wins" in updating $\mathcal{S}$, since it will duly flip the Out variable before it goes to the REMAINDER SECTION. However, if the process in the TRYING SECTION wins and the process is of the same session as the one that is just written into $\mathcal{S}$, concurrent entering might be in jeopardy.

A tempting idea to handle this situation is that if the leaving process fails in its *SC* operation, then it should not just leave. Instead, it should check one more time whether the Out variable has been duly changed to the same side as recorded in $\mathcal{S}$. But this idea brings with itself an issue that is very "dangerous": *if we allow processes in the* TRYING SECTION *and in the* LEAVING SECTION *to modify the Out variables no matter they succeed in updating $\mathcal{S}$ or not, it may happen that a leaving process is poised to flip $Out[s]$ and falls asleep even though the ongoing session in $\mathcal{S}$ has been changed to sessions other than s.* This situation is undesirable, since the Out variables serve as the "gateway"—processes busywait on them before getting into the CS. We do not want processes to be poised to change them unless the circumstances so required (i.e., it is about time that the ongoing session should be changed).

The above idea and its subsequent concern bring in the most subtle part of our design. See Figure 3 for our final, and correct algorithm. Its major differences from the previous one include (1) the separation of `Promote` procedure, now one for processes in the TRYING SECTION and one for those in the EXIT SECTION, and (2) the addition of Lines 9-12, which, roughly speaking, serve as a second gateway (the first being the Out variables).

The `Promote-Try` procedure is essentially the same as `Promote` in the previous algorithms. On the other hand, the `Promote-Exit` procedure is a big departure, whose design is based on the idea we gave previously. It makes use of the following variables:

1. *Ex*: a variable indicating the session of the processes that *may* be still lingering in the EXIT SECTION. This variable is an *LL*/*SC* variable, initialized to false.

2. $\mathit{Exclear}[s]$: a binary variable, indicating whether the processes in the TRYING SECTION need to wait on those processes of s which may be still lingering in the EXIT SECTION. This variable is a register and is initialized to true.

We ask the readers to ignore Lines 9-12 and 16.5 and 16.14 temporarily.

In `Promote-Exit`, a process begins by erasing whatever session that has been recorded in the *Ex* variable (Line 16.2). It applies *LL* (Line 16.3) and *SC* (Line 16.9) operations on the *Ex* variable, and only if its *SC* operation is successful does it proceed to Lines 16.10-16.14. The reason of introducing *Ex* and using *LL*/*SC* operations on it is because we want to guarantee that *at most one* process in the EXIT SECTION can proceed to Lines 16.10-16.14 so as to modify $\mathcal{S}$ and the Out variables. Moreover, note that leaving processes in `Promote-Exit` decrease the counter of their own session (Line 16.6) *after* they have *LL*ed the *Ex* variable at Line 16.3. If any of the leaving processes succeeds in Line 16.9, no more process can perform a successful *SC* operation on *Ex* until all processes of the current session have gone to the REMAINDER SECTION.

If a process p succeeds in *SC*ing its own session into *Ex*, p proceeds to update the session variable $\mathcal{S}$ (Line 16.10) based on its knowledge of the highest priority session (Line 16.7). If it succeeds in its *SC* operation, it updates the Out variable of the new session. Even if it fails in its *SC* operation at Line 16.10, it still double checks $\mathcal{S}$ (Line 16.12). In the case that $\mathcal{S}$ is written by some process q in the other procedure `Promote-Try` and q also belongs to the new session, p takes the responsibility of updating the Out variable, just in case that q falls asleep before it executes Line 5.5. It can be seen that the above mechanism fixes the scenario that we described in the third failure.

We now explain the purpose of Lines 9-12, 16.5, and 16.14. As the variable $Out[s]$ can be written by either a process in `Promote-Try` or in `Promote-Exit`, the processes of sessions s, even if they get past Line 8, have no clue how $Out[s]$ is changed. In the case that $Out[s]$ is updated in `Promote-Try`, some processes may still linger in `Promote-Exit`. In this case, these processes of s should either (1) wait until the lingering processes in `Promote-Exit` all leave, or (2) "invalidate" them so that in the future, even if these lingering processes take steps, no harm will be done (i.e., they will not be able to modify $\mathcal{S}$ and the Out variables).

First ignore Lines 9 and 10. If these processes of s note that *Ex* is true, then some leaving process (indeed, exactly one) must have successfully performed an *SC* operation on it (Line 16.9). Then these processes of s should wait (Line 12) until that leaving process has really quit (Line 16.14).

To see that Lines 9 and 10 are necessary, observe that if any process of s succeeds in *SC*ing at Line 10, then none of the processes in `Promote-Exit` will succeed in Line 16.9 and thus, we can be sure that they will not be able to modify $\mathcal{S}$ or the Out variables afterwards. On the other hand, if *all* processes of s fail in their *SC* operations on Line 10, then we can be sure that exactly one process in `Promote-Exit` has succeeded in Line 16.9. In this case, we just let all processes of s make sure that this lingering process in `Promote-Exit` has really finished before they get into the CS (Line 12).

[5]We remark that there are other scenarios of different "flavor" to cause this algorithm to violate P4. We do not attempt to enumerate all of them here.

$\mathcal{J}$ is the set of all possible sessions.
Shared variables common to all sessions:
Q is a priority process-queue supports *enqueue*, *dequeue* and *Min* operations, initially empty.
$\mathcal{S} \in \mathcal{J} \times \{0,1\} \times \mathcal{J}$ is a *LL/SC* variable initialized to $(s,0,s)$ for some $s \in \mathcal{J}$.
$Ex \in \{\mathcal{J} \cup \mathit{false}\}$ is a *LL/SC* variable initialized to *false*.
$G \in \mathbb{N}$, a counter supporting *inc* and *Read* operations, initialized to 0.
Shared variables for session $s \in \mathcal{J}$:
$In[s] \in \{0,1\}, Out[s] \in \{0,1\}$ read/write variables, $In[s]$ is initialized to 1 and Out[s] to 0.
$Exclear[s] \in \{\mathit{true}, \mathit{false}\}$ is a read/write variable initialized to *true*. We also assume $Exclear[\mathit{false}]$ to be always *true*.
$C[s,0], C[s,1] \in \mathbb{N}$, counters supporting *inc* and *Read* operations, initialized to 0.

```
Gmutex-Lock ($s$)
REMAINDER SECTION
1   $d \leftarrow$ Inc-Get-Dir ($s$)
2   $inc(G, 1)$
3   $tm \leftarrow$ Read$(G)$
4   Enqueue$(Q, [p, s, d, tm])$
5   Promote-Try($s$)
6   if $LL(\mathcal{S}) = (s, d, s)$
7       Out$[s] \leftarrow d$
8   wait till $Out[s] = d$
9   if $\neg LL(Ex)$
10      $SC(Ex, false)$
11  if $false \neq s^e = LL(Ex)$
12      wait till $Exclear[s^e]$
CRITICAL SECTION
13  Dequeue$(Q, [p, s, d, tm])$
14  if $Min(Q) \neq (s, -)$
15      $In[s] \leftarrow \overline{d}$
16  Promote-Exit($s$)
```

```
Inc-Get-dir($s$)
1.1 $inc(C[s,0], 1)$
1.2 $inc(C[s,1], 1)$
1.3 $d \leftarrow In[s]$
1.4 if $Min(Q) \in \{(s,-), (\bot,\bot)\}$
1.5     if $LL(\mathcal{S}) = (s, \overline{d}, -)$
1.6         $d \leftarrow \overline{d}$
1.7 $inc(C[s,\overline{d}], -1)$
1.8 return $d$
```

```
Promote-Try($s$)
5.1 $(s^c, d^c, -) \leftarrow LL(\mathcal{S})$
5.2 $(s^h, d^h) \leftarrow Min(Q)$
5.3 if $C[s^c, d^c] = 0 \wedge s^h \neq s^c$
5.4     if $SC(\mathcal{S}, (s^h, d^h, s))$
5.5         Out$[s^h] \leftarrow d^h$
```

```
Promote-Exit($s$)
16.1  $LL(Ex)$
16.2  $SC(Ex, false)$
16.3  $LL(Ex)$
16.4  $LL(\mathcal{S})$
16.5  $Exclear[s] \leftarrow false$
16.6  $inc(C[s,d], -1)$
16.7  $(s^h, d^h) \leftarrow Min(Q)$
16.8  if $C[s,d] = 0 \wedge s^h \neq \bot$
16.9      if $SC(Ex, s)$
16.10         if $SC(\mathcal{S}, (s^h, d^h, s))$
16.11             Out$[s^h] = d^h$
16.12         if $\mathcal{S} = (s', d', s'') \wedge s' = s''$
16.13             Out$[s'] \leftarrow d'$
16.14         $Exclear[s] \leftarrow true$
```

Figure 3: The final algorithm. Lines 1-7 constitute the doorway.

3. PROOF SKETCH

In this section, we give a proof sketch for the Theorem 1. Our proof sketch relies on the correctness of some invariants. Due to space constraints, we present these invariants without their formal proofs. First we give some notations and definitions required to express these invariants.

3.1 Notations and definitions

- $\{H\}$ denotes the set of processes with their program counters from set H, where H is a set of program counter. For example, $\{2-8\}$ denotes the set of processes with their program counters between Lines 2 and 8.
- For a process p, $p.x$ denotes the value of the local variable x in context. For example, $p.s$ and $p.d$ denote the session s and the side d of p.
- $\mathcal{LS}$ is a set of processes in $\{5.2-5.4\} \cup \{16.5-16.10\}$ such that if $p \in \mathcal{LS}$ then no successful *SC* has been performed on $\mathcal{S}$ since p performed the last *LL* on $(\mathcal{S})$.
- $\mathcal{LE}$ is a set of processes in $\{16.4-16.9\}$ such that if $p \in \mathcal{LE}$ then no successful *SC* has been performed on *Ex* since p performed *LL*(*Ex*) (Line 16.3).
- When we say a process p is of side d' if $p \in \{1.4-16.6\}$ and $p.d$ is equal to d' or p is at Line 1.6 and $p.d = \overline{d'}$. Note that if p is of session s and of side d', then $C[s,d'] > 0$.
- We denote $p[L]$ in $\mathcal{A}$, as the time at which p executes Line L in its attempt $\mathcal{A}$. Whenever the attempt $\mathcal{A}$ is clear form the context, we simply use $p[L]$.
- When we say $(s,d) \in Q$, we mean that there is an element $[p,s,d,tm] \in Q$.

Now we present the invariants used in proving the properties of the algorithm.

Invariant 1. If there is no process from session s and side d active, then $C[s,d] = 0$. Furthermore, if no process from session s is active, then $C[s,d] = C[s,\overline{d}] = 0$ and $(s,-) \notin Q$.

Invariant 2. If some process p from session s is in $\mathcal{LS}$, $\mathcal{S} = (s^c, d^c, -)$ and p is about to *SC* (s^h, d^h, s) into $\mathcal{S}$ at time t, then

a. there is no process from session s^c and side d^c at t.

b. there is some process from session s^h and side d^h, but no process from session s^h and side $\overline{d^h}$ at t.

c. $s^h \neq s^c$

Invariant 3. If $\mathcal{S} = (s, d', w)$ and $Out[s] \neq d'$, then there exists a a process $q \in \{5.5, 16.11\}$ of session w.

Invariant 4. If a process is about to set $Out[s]$ to d (Lines 5.5, 16.11 or 16.13) at time t, then $\mathcal{S} = (s, d, -)$ at t.

Invariant 5. If there is a process $p \in \{9-16.6\}$ from session s and side d, then $\mathcal{S} = (s, d, -)$ and $Out[s] = d$.

Invariant 6. If a process p of session s and side d is at Line 12, $Exclear[Ex = s^e] = \mathit{false}$, then, there is exactly one process at Lines 16.10-16.13 and it belongs to session $s^e \neq s$.

Invariant 7. If some process $p \in \mathcal{LE}$ of session s and of side d is at Line 16.9, then $\mathcal{S} = (s, d, -)$ and there is no process of session s and side d.

Invariant 8. If a process p of session s and side d is at Line 8, such that $Min(Q) = (s, d)$, $\mathcal{S} = (s', d', -)$, $C[s', d'] = 0$ and $s \neq s'$, then there exists a process with session s' in the exit section, or at Lines 1.8 to 5.4.

The following lemma will be useful when we prove the main theorem.

LEMMA 1. *If at time t, process p of session s and of side d is in* $\{2-8\}$ *and* $\mathcal{S} = (s, \overline{d}, -)$*, then at t, there is a process* $q \in \{5-8\}$ *from session* $s' \neq s$ *such that q.tm < p.tm. Furthermore, there is no process in the CS at t which door-way precedes q.*

PROOF. The following two claims will be used to prove this lemma. Throughout these proofs, when we say p executes Line L, we mean p executes Line L in its attempt active at t.

CLAIM 1. *p did not execute Line 1.6 in its current attempt.*

PROOF. Suppose, for a contradiction, that p did execute Line 1.6. As p does not change its side after Line 1.6 and p is of side d at t, it means that p found $\mathcal{S}$ equal to $(s, d, -)$ at Line 1.5. But $\mathcal{S} = (s, \overline{d}, -)$ at t, it means that some process *SC*ed $(s, \overline{d}, -)$ into $\mathcal{S}$ during the interval $(p[1.5], t)$. But note that p is of side d in the interval $(p[1.5], t]$. The previous two facts together contradict Invariant 2. □

CLAIM 2. *When p executes Line 1.4,* $Min(Q) = (s', -, -)$*, where* $s \neq s'$*.*

PROOF. By the previous claim one can see that p is from side d throughout the interval $(p[1.4], t]$. As $\mathcal{S}$ is equal to $(s, \overline{d}, -)$ at t, by Invariant 2 (b), $\mathcal{S} = (s, \overline{d}, -)$ through $(p[1.4], t]$. Hence, if p executed Line 1.5 it would have executed Line 1.6, which contradicts the previous claim. This shows that when p executes Line 1.4, $Min(Q) = (s', -, -)$, where $s \neq s'$. □

By the previous claim one can see that request of some process q (of session $s' \neq s$) is in Q at $p[1.4]$. As a process only removes its request from Q at Line 13. By Invariant 5 and the fact that $\mathcal{S} = (s, \overline{d}, -)$ in the interval $(p[1.4], t]$, it means that q's request is still in Q at t and $q \in \{5-8\}$ at t. As p will get a bigger time-stamp than q, it also means that $q.tm < p.tm$ at t.

Also note that if there is some process p' from session s in the CS at t and doorway precedes q, then $p'.tm < q.tm$ and $(p', s) \in Q$ throughout the interval $[p[1.4], t]$. But this contradicts the previous claim.

3.2 Proof of Theorem 1

It is easy to see the algorithm satisfies **Bounded exit** (P3). **Mutual exclusion** (P1) follows easily from Invariant 5. We defer the proof of **Relaxed FCFS** (P5') to the full version. In this subsection we will prove that the algorithm satisfies **Concurrent Entering** (P4), **Starvation freedom** (P2), **Pulling** (P6) and has $O(min(\log n, k))$ RMR complexity.

LEMMA 2 (**Concurrent Entering**). *If process p is in the* TRYING SECTION*, it performs at most b steps in absence of any active conflicting processes before entering the CS, where b is some bound.*

The proof of this lemma follows from Claims 3 and 5 below.

CLAIM 3. *At time t, if a process p of session s and of side d is at Line 8 and no conflicting processes are active at time t, then* $Out[s] = d$ *at t.*

PROOF. As no processes conflicting with p are active at time t, by Invariant 1 and the fact that $Q \neq \phi$ (because Q has the request (s, d) from p), $Min(Q) = (s, -)$ at t. By Invariants 1 and 8 and the fact that $Min(Q) = (s, -)$, one can see that $\mathcal{S} = (s, d', w)$, for some $d' \in \{0, 1\}$. Also, as p (from side d) is at Line 8, by Lemma 1 one can see that $d' = d$, hence $\mathcal{S} = (s, d, w)$ at t.

Now we will prove this claim by contradiction, i.e., say $Out[s] \neq d$ at t. By Invariant 3, there exists a process at Lines 5.5 or 16.11 of session w. If $w \neq s$, then some conflicting process is active at t, which is a contradiction. So the interesting case is at time t, $\mathcal{S} = (s, d, s)$. (Recall that for this particular case we introduced *Ex* and *Exclear* variables in final correct algorithm). The following claim will be crucial to complete the proof of Claim 3.

CLAIM 4. *If* $\mathcal{S} = (s, d, s)$ *and* $Out[s] \neq d$ *at t, then there is some process q of session* $s' \neq s$ *such that the following hold :*

(a) q executes Line 16.6 *in the interval* $(p[5.1], t)$*.*

(b) $C[s'', -] = 0 \wedge Min(Q) = (s, d), \forall s'' \neq s$ *throughout the interval* $(q[16.6], t]$*.*

(c) $\mathcal{S}$ *changes exactly once in the interval* $(q[16.6], t)$*.*

PROOF. We will prove part (a) by contradiction. As $C[s'', -] = 0, \forall s'' \neq s$ at t, if there is no process conflicting with p which performs Line 16.6 in the interval $(p[5.1], t)$, then $C[s'', -] = 0, \forall s'' \neq s$ throughout the interval $(p[5.1], t)$. Hence, in the interval $(p[5.1], t)$, $C[s'', -] = 0 \wedge Min(Q) = (s, d), \forall s'' \neq s$.

Now we will prove that $\mathcal{S} = (s, d, s)$ at $p[6]$. We will prove this by further two cases depending upon the value of $\mathcal{S}$ at $p[5]$. Say $\mathcal{S} = (s, -, -)$ at $p[5]$. As $\mathcal{S} = (s, d, s)$ at t, we first claim that $\mathcal{S} = (s, d, s)$ throughout the interval $(p[5.1], t)$. This is true because for $\mathcal{S}$ to change at some time during the interval $(p[5.1], t)$, by Invariant 2c, $\mathcal{S}$ will have to change to $(s^*, -, -)$ first, where $s^* \neq s$. But this contradicts Invariant 2b because in the interval $(p[5.1], t)$, $C[s^*, -] = 0$. Now consider the case when $\mathcal{S} = (s'', -, -)$ and $s'' \neq s$ at $p[5]$. As in the interval $(p[5.1], t)$, $C[s'', -] = 0 \wedge Min(Q) = (s, d), \forall s'' \neq s$. So p (or some other fellow process) will set $\mathcal{S} = (s, d, s)$ in `Promote-try`.

Hence, $\mathcal{S} = (s, d, s)$ at $p[6]$. This means that p will execute Line 6 and set $Out[s] = d$ at Line 7. Hence, $Out[s] = d$ when p moves to Line 8. As $Out[s] \neq d$ at t, so some process changes $Out[s]$ at some time $t' \in (p[7], t)$. Also by Invariant 4, $\mathcal{S} = (s, \overline{d}, -)$ at t'. But this means that $\mathcal{S}$ was changed from $\mathcal{S} = (s, d, s)$ to $\mathcal{S} = (s, \overline{d}, -)$ while p is at side d which contradicts Invariant 2a. Hence, part (a) is proved. Now, as there exists some process with session different than s that executes Line 16.6 ($inc(C[s, d], -1)$) in the interval $[p[5.1], t]$, let q be the last such process and say it is from session s'. Then the proof of part (b) simply follows for this process q.

By Invariant 5, $\mathcal{S} = (s', -, -)$ at $q[16.6]$ and by our assumptions $\mathcal{S} = (s, -, -)$ at t. As all the processes in the interval $(16.6, t)$ are of session s and of side d. By Invariant 2, $\mathcal{S}$ can change at most once in the interval $(q[16.6], t)$ and that also to (s, d, s). Hence, we get the the part (c). □

Now we concentrate on steps taken by q before it leaves the exit section (and note that these steps must be taken before t, as there is no active process conflicting with p at t). When q starts executing beyond Line 16.6, it would find $C[s', d'] = 0$ and $Min(Q) = (s, d)$. Then q tries to *SC* s' into *Ex* (Line 16.9), it can succeed or fail.

If it succeeds, it will go on to Line 16.10. As we know that $\mathcal{S} = (s, d, s)$ at t and by part (c) of the claim above, one can see that *SC* by q at Line 16.10 will fail and $\mathcal{S} = (s, d, s)$ at Line 16.12. Hence q will set $Out[s] = d$ before leaving the exit section.

If q fails at Line 16.9 then it means that some process performed a successful *SC* on Ex since $q[16.6]$. If the successful *SC* was performed by some process at Line 16.9, then by Invariant 7 one can see that this process (which is of session s') performs Line 16.9 after $q[16.6]$. Hence, the same arguments as applied when q succeeds at Line 16.9 hold. If the successful *SC* was performed some process at Line 10, then by Invariant 5, one can see that $Out[s] = d$. Hence we have shown a contradiction for all the cases and complete the proof of Claim 3.

CLAIM 5. *If a process p of session s and side d is at Line 12 at time t, and no conflicting processes are active at t, then $Exclear[s^e] = true$.*

PROOF. This claim is just a consequence of Invariant 6. □

LEMMA 3 (**Starvation Freedom**). *If a process is in the* TRYING SECTION, *then it eventually enters the CS.*

PROOF. We will prove by contradiction. Suppose that a nonempty set of processes S_T are in the TRYING SECTION forever. As processes take an ever-increasing time stamps at Line 3, one can see that eventually $Min(Q) = (s, d)$ forever, where s and d is session and side of a process p in S_T. We will prove that p eventually enters the CS, which contradicts our assumption that S_T is nonempty. We first consider the case where p is looping at Line 12 forever.

CLAIM 6. *If a process p is at Line 12, then eventually Exclear $[s^e] = true$. Furthermore, once $Exclear[s^e]$ is set to true, it is not changed to false while p is at Line 12.*

PROOF. The first part is a straightforward consequence of Invariant 6. Also by Invariant 6, $s^e \neq s$. We will prove the second part of this claim by contradiction. Say a process q sets $Exclear[s^e]$ to *false* at Line 16.5 while p is at Line 12. By Invariant 5, when a process q from session s^e sets $Exclear[s^e]$ to *false* at Line 16.5, the variable $\mathcal{S} = (s^e, -, -)$. But this contradicts the Invariant 5 as p from session $s \neq s^e$ is at Line 12. □

So if p is in TRYING SECTION forever, it will loop at Line 8 forever. The following two claims together also rule out this possibility. We first prove that eventually $\mathcal{S} = (s, d, -)$ forever.

CLAIM 7. *If $Min(Q) = (s, d)$ for all time after t, then eventually $\mathcal{S} = (s, d, -)$ at some time $t' \geq t$.*

PROOF. We will prove this claim by contradiction. Suppose that $\mathcal{S} \neq (s, d, -)$ for all times after t. By Lemma 1 and the fact that $Min(Q) = (s, d)$, one can see that $\mathcal{S} \neq (s, \overline{d}, -)$ for all time after t. Hence by the above assumptions, we get, $\mathcal{S} \neq (s, -, -)$ for all times after t.

First we claim that $\mathcal{S}$ changes at most once after t. We prove this claim as follows. Say that $\mathcal{S}$ is changed for the first time after t at $t' \geq t$, i.e., some successful *SC* was performed on $\mathcal{S}$ at t'. Then for any process r to change $\mathcal{S}$ after t' at Line 5.4 (respectively, 16.10), r has to perform the $LL(\mathcal{S})$ at Line 5.1 (respectively, 16.4) after t'. After the $LL(\mathcal{S})$ at Line 5.1 (respectively, 16.4), r will find $Min(Q) = (s, d)$ at Line 5.2 (respectively, 16.7), hence it can change $\mathcal{S}$ only to $(s, d, -)$, which is a contradiction. So this means $\mathcal{S}$ is fixed to $(s', d', -)$ forever after some time $t' \geq t$ and for some $s' \neq s$.

Now we will concentrate on the processes from session s' and side d'. First by Invariant 3 and Invariant 4, one can see that eventually $Out[s'] = d'$ forever. Hence, we know that for all times after some time $t^s \geq t' \geq t$, $Min(Q) = (s, d), \mathcal{S} = (s', d', -), Out[s'] = d'$, where $s' \neq s$. Now we claim that if a process q from session s' and side d' is in the CS at time $t^q \geq t^s$, and if q attempts for CS again from session s', it will attempt from side $\overline{d'}$. More precisely when q again executes Line 2 after t^q with session s', its local variable d will be set to $\overline{d'}$. This is true because when q executes Line 15 for the first time after t^q, as $Min(Q) = (s, d)$ it will set $In[s']$ to $\overline{d'}$ (Line 16). The next time it executes Line 1.3 with session s', it will get $\overline{d'}$ from $In[s]$ and it will also not go into Lines 1.5-1.6 as $Min(Q) = (s, d)$. Hence when q reaches Line 2, it will have side $\overline{d'}$.

Also note that because $Out[s'] = d'$ forever, if a process of session s' and side d' is in TRYING SECTION beyond Line 1.8, then it will eventually enter the CS. Combining the previous two facts, and the facts that there are only finitely many processes in the system and processes only crash in remainder section, one can see that, eventually $C[s', d'] = 0$ and all the active processes of session s' have side $\overline{d'}$ and are looping at Line 8 forever. More formally, there is a time $t'' \geq t^s$ such that for all times after t'', $Min(Q) = (s, d), \mathcal{S} = (s', d', -), C[s', d'] = 0$ and if there is any active process from session s', it is at Line 8. So we arrive at a contradiction to Invariant 8 and a proof of our claim. □

CLAIM 8. *If a process p from session s and side d is at Line 8 and $\mathcal{S} = (s, d, -)$ then eventually $Out[s]$ is set to d by some process. Furthermore, once $Out[s]$ is set to d, it is not changed while p is at Line 8.*

PROOF. The first part is follows trivially from Invariants 3 and 4. For the second part, again by Invariant 4, when a process executes any of the Lines 5.5, 16.11, 16.13 to change $Out[s]$ to d', the $\mathcal{S}$ variable at that point is $(s, d', -)$. But by Invariant 2(a), $\mathcal{S}$ does not change from $(s, d, -)$ while p is at Line 8. □

LEMMA 4 (**Pulling**). *If at time t a process p of session s and of side d is at Lines 8-12 and some process q from session s is in the CS, such that q doorway precedes all active processes conflicting with p at t, then p takes a bounded number of steps to enter the CS.*

PROOF. We prove this lemma by proving the following two claims.

CLAIM 9. *If p is at Line 8 at time t, then $Out[s] = d$.*

PROOF. As q is in the CS, by Invariant 5, $\mathcal{S} = (s, d', -)$ and $Out[s] = d'$ where d' is the side of q. Also, as q door-way precedes all active conflicting processes and p is at Line 8, by Lemma 1, side of p, i.e., $p.d$ is equal to d'. Hence, $Out[s] = d$, and by claim 8, $Out[s]$ will be equal to d while p is at Line 8. □

CLAIM 10. *If p is at Line 12 at time t, then $Exclear[Ex] =$ true.*

PROOF. As q entered the CS at some time before t, it would have found $Exclear[Ex] = true$ at some time $t' < t$. (Note that if Ex is equal to *false* at $q[11]$, $Exclear[Ex]$ is still *true* by definition). So for some process r to set $Exclear[Ex] = false$ in the interval (t', t), r has to execute a successful *SC* at Line 16.9. By Invariants 1 and 7 and the fact that $\mathcal{S} = (s, d, -) \wedge C[s, d] > 0$ in the interval (t', t), one can see that this cannot happen. Hence, $Exclear[Ex] = true$ at time t. Also by claim 6, $Exclear[Ex]$ will remain *true* while p is at Line 12. □

LEMMA 5 (**Remote Memory Reference**). *The remote memory reference of the algorithm in cache coherent architecture is $O(\min(\log n, k))$, where n is the number of processes and k is the point contention.*

PROOF. The main complexity bottleneck of the doorway and exit section are the inc, *Enqueue*, *Dequeue* and *Min* operations. As stated in the Theorems 2 and 3, these operations when implemented using f-arrays [18] take $O(\min(\log n, k))$ RMR, where n is the number of processes and k is the point contention. As these operations are executed only a constant number of times, to show that the total RMR of the algorithm is also $O(\min(\log n, k))$, we need to show that the RMR associated with the loops at Lines 8 and 12 is also $O(\min(\log n, k))$. By the Claims 6 and 8 one can see that RMR associated with these loops is indeed just 2. □

4. CONCLUSION AND OPEN PROBLEMS

In this work we present an algorithm to show that in the CC model, the upper bound and the lower bound of RMR match in the group mutual exclusion problem, just as in the ordinary mutual exclusion problem. We would like to point out a few future research directions and the limitation of our result.

Our algorithm relies on the assumption that the set of possible sessions is known a priori (note that Danek and Hadzilacos' algorithm [16] has the same limitation). Furthermore, it uses unbounded counters. It would be desirable to design algorithms that can do without these while still ensuring the same RMR.

There may be other desirable properties of GME that can be formulated and achieved. One example is the FIFE (first-in-first-enabled), which states that if p and p' request the same session and the former doorway-precedes the latter, and if the latter is in the CS, then the former should be able to enter the CS in its own bounded number of steps. Unlike FCFS (P5), this property is compatible with P6. Unfortunately, our algorithm does not satisfy this property.

Another interesting question is whether we can further improve the complexity by using primitives like *fetch&add*. We already know that constant RMR algorithms for mutual exclusion exist [5, 14, 25]. So is it possible to design a constant RMR GME algorithm using *fethc&add* ? Interestingly for a special case of GME called the READER-WRITER PROBLEM [9], constant RMR solutions do exist [8].

5. ACKNOWLEDGMENT

We thank the anonymous referees for many helpful comments and Ranganath Kondapally for reading early drafts. Special thanks to Prasad Jayanti for his many useful discussions with the second author in the early stage of this work. The properties P5' and P6 are also due to him.

6. REFERENCES

[1] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *In Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, 2001.

[2] J. H. Anderson and Y.-J. Kim. Fast and scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, London, UK, 1999. Springer-Verlag.

[3] J. H. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distrib. Comput.*, 15(4):221–253, 2002.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[6] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 91–100, New York, NY, USA, 2000. ACM.

[7] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 217–226, New York, NY, USA, 2008. ACM.

[8] V. Bhatt and P. Jayanti. Constant RMR solutions to reader writer synchronization. Submitted to *PODC '10 : Proceedings of the Twenty-Ninth annual symposium on Principles of distributed computing*.

[9] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10):667–668, 1971.

[10] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 148–156. IEEE, 1993.

[11] R. Cypher. The communication requirements of mutual exclusion. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 147–156, New York, NY, USA, 1995. ACM.

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

[13] R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 275–284, New York, NY, USA, 2006. ACM.

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

[15] V. Hadzilacos. A note on group mutual exclusion. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 100–106, New York, NY, USA, 2001. ACM.

[16] V. Hadzilacos and R. Danek. Local-spin group mutual exclusion algorithms. In *DISC'04: Proceedings of the 18th International Symposium on Distributed Computing*, pages 71–85. Springer Berlin / Heidelberg, 2004.

[17] D. Hendler and P. Woelfel. Randomized mutual exclusion in $O(\log n / \log \log n)$ RMRs. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 26–35, New York, NY, USA, 2009. ACM.

[18] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270 – 279, 2002.

[19] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 295–304, New York, NY, USA, 2003. ACM.

[20] P. Jayanti and S. Petrovic. Efficient and practical constructions of ll/sc variables. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 285–294, New York, NY, USA, 2003. ACM.

[21] P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 275–284, New York, NY, USA, 2003. ACM.

[22] Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 51–60, New York, NY, USA, 1998. ACM.

[23] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32, New York, NY, USA, 1999. ACM.

[24] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 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] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 106–113, New York, NY, USA, 1991. ACM.

[27] M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. In *DISC'04: Proceedings of the 18th International Symposium on Distributed Computing*, pages 144–158. Springer Berlin / Heidelberg, 2004.

[28] M. Moir. Practical implementations of non-blocking synchronization primitives. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228, New York, NY, USA, 1997. ACM.

[29] J.-H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9:9–1, 1994.