View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Mutual Exclusion Algorithms with Constant RMR Complexity and Wait-Free Exit Code

Rotem Dvir[1] **and Gadi Taubenfeld**[2]

1 **The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel**
rotem.dvir@gmail.com
2 **The Interdisciplinary Center, P.O.Box 167, Herzliya 46150, Israel**
tgadi@idc.ac.il

Abstract

Two local-spinning queue-based mutual exclusion algorithms are presented that have several desired properties: (1) their exit codes are wait-free, (2) they satisfy FIFO fairness, (3) they have constant RMR complexity in both the CC and the DSM models, (4) it is not assumed that the number of processes, n, is a priori known, that is, processes may appear or disappear intermittently, (5) they use only $O(n)$ shared memory locations, and (6) they make no assumptions on what and how memory is allocated.

The algorithms are inspired by J. M. Mellor-Crummey and M. L. Scott famous MCS queue-based algorithm [13] which, except for *not* having a wait-free exit code, satisfies similar properties. A drawback of the MCS algorithm is that executing the exit code (i.e., releasing a lock) requires spinning – a process executing its exit code may need to wait for the process that is behind it in the queue to take a step before it can proceed. The two new algorithms overcome this drawback while preserving the simplicity and elegance of the original algorithm.

Our algorithms use exactly the same atomic instruction set as the original MCS algorithm, namely: read, write, fetch-and-store and compare-and-swap. In our second algorithm it is possible to recycle memory locations so that if there are L mutual exclusion locks, and each process accesses at most one lock at a time, then the algorithm needs only $O(L+n)$ space, as compared to $O(Ln)$ needed by our first algorithm.

1998 ACM Subject Classification C.2.4 Distributed Systems, F.1.1 Models of Computation

Keywords and phrases Mutual exclusion, locks, local-spinning, cache coherent, distributed shared memory, RMR complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.17

1 Introduction

Concurrent access to resources shared among several processes must be synchronized in order to avoid interference between conflicting operations. Mutual exclusion locks are still the de facto mechanism for concurrency control on shared resources: a process accesses the resource only inside a critical section code, within which the process is guaranteed exclusive access. The popularity of this approach is largely due to the apparently simple programming model of such locks, and the availability of lock implementations which are reasonably efficient.

Most of the mutual exclusion lock algorithms include busy-waiting loops. The idea is that in order to wait, a process *spins* on a flag register, until some other process terminates the spin with a single update operation. Unfortunately, under contention, such spinning may generate lots of traffic on the interconnection network between the process and the memory, which can slow other processes. To address this problem, it is important to distinguish

© Rotem Dvir and Gadi Taubenfeld;
licensed under Creative Commons License CC-BY

21st International Conference on Principles of Distributed Systems (OPODIS 2017).
Editors: James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão; Article No. 17; pp. 17:1–17:16

Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

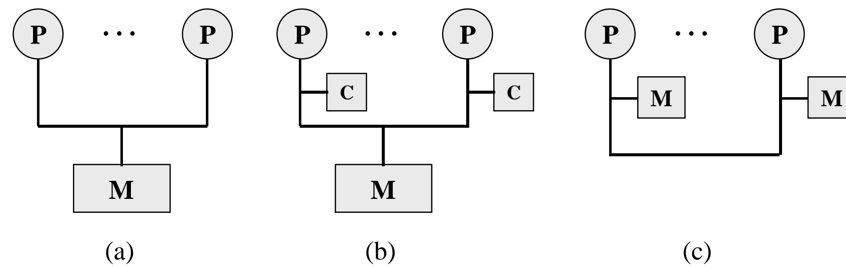

Figure 1 Shared memory models. (a) Central shared memory. (b) Cache Coherent (CC). (c) Distributed Shared Memory (DSM). P denotes processor, C denotes cache, M denotes shared memory.

between *remote* access and *local* access to shared memory, and to try to reduce the number of remote accesses as much as possible.

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

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

An algorithm that satisfies local spinning in a DSM system, is expected to perform well also when executed on a machine with *no* DSM. The reason is that each process spins only on memory locations on which no other process spins, thus eliminating hot-spot contention caused by busy-waiting.

The MCS lock, due to John Mellor-Crummey and Michael Scott, is perhaps the best-known and most influential local-spinning lock algorithm [13]. This important algorithm and several variants of it are implemented and used in various environments. For example, Java Virtual Machines use object synchronization based on variations of the MCS lock [7].

A code segment in an algorithm is *wait-free* if its execution by a process should require only a finite number of steps and must always terminate regardless of the behavior of the other processes. A drawback of the MCS lock is that releasing it is *not wait-free* and requires spinning – a process that is releasing the lock may need to wait for the process that is trying to acquire the lock to take a step before it can proceed. Thus, when there is high contention, a releasing process may have to wait for a long time until a process that is trying to acquire the lock is schedule. We present two new local-spinning algorithms which overcome this drawback while preserving the simplicity and elegance of the original MCS algorithm.

The two new mutual exclusion algorithms, which are inspired by the MCS algorithm, have several desired properties. These properties, formally defined in the next section, are: (1) their exit codes are wait-free, (2) they satisfy FIFO fairness, (3) they have constant RMR (remote memory reference) complexity in both the CC and the DSM models, (4) they do not require to assume that the number of participating processes, n, is a priori known, that is,

processes may appear or disappear intermittently, (5) they use only $O(n)$ shared memory locations, and (6) they make no assumptions on what and how memory is allocated[1].

Except for property 1 above, the other properties are satisfied also by the MCS algorithm. No previously published algorithm satisfies all these properties together.

Our algorithms use exactly the same atomic instruction set as the original MCS algorithm, namely: read, write, fetch-and-store and compare-and-swap. In our second algorithm it is possible to recycle memory locations so that if there are L locks, and each process accesses at most one lock at a time, then the algorithm needs only $O(L+n)$ space, as compared to $O(Ln)$ needed by our first algorithm.

2 Preliminaries

2.1 Computational model

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

- *Read:* takes a shared registers r and simply returns its value.
- *Write:* takes a shared registers r and a value *val*. The value *val* is assigned to r.
- *Fetch-and-store* (FAS): takes a shared register r and a local register ℓ, and atomically assigns the value of ℓ to r and returns the previous value of r. (The fetch-and-store operation is also called *swap* in the literature.)
- *Compare-and-swap* (CAS): takes a shared register r, and two values: *new* and *old*. If the current value of the register r is equal to *old*, then the value of r is set to *new* and the value *true* is returned; otherwise r is left unchanged and the value false is returned.

Most modern processor architectures support the above operations.

2.2 Mutual exclusion

The mutual exclusion problem is to design an algorithm that guarantees mutually exclusive access to a critical section among n competing processes [3]. It is assumed that each process is executing a sequence of instructions in an infinite loop. The instructions are divided into four continuous sections: the remainder, entry, critical and exit. The entry section consists of two parts: the *doorway* which is *wait-free*, and the waiting part which includes one or more loops. A *waiting* process is a process that has finished its doorway code and reached the waiting part, and a *beginning* process is a process that is about to start executing its entry section. It is assumed that a process may crash[2] in its remainder section, but may not crash in its entry, critical or exit sections. It is also assumed that a process always leaves its critical section.

[1] For example, in [2] it is assumed that *all* allocated pointers must point to *even* addresses.
[2] A process that *fails by crashing* is a process that stops its execution in a definitive manner.

The *mutual exclusion problem* is to write the code for the entry and the exit sections in such a way that the following *two* basic requirements are satisfied.

- *Deadlock-freedom*: If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.
- *Mutual exclusion*: No two processes are in their critical sections at the same time.

Satisfaction of the above two properties is the minimum required for a mutual exclusion algorithm. For an algorithm to be fair, satisfaction of an additional condition is required.

- *FIFO (First-in-first-out)*: A beginning process cannot execute its critical section before a waiting process executes its critical section.
- *Strong FIFO*: A process that has not completed its doorway cannot execute its critical section before a waiting process executes its critical section.

All our algorithms satisfy the slightly stronger *strong FIFO* requirement. To simplify the presentation, when the code for a mutual exclusion algorithm is presented, only the entry code and exit code are described, and the remainder code and the infinite loop within which these codes reside are omitted.

2.3 Counting Remote Memory References

As already mentioned, for certain shared memory systems, it makes sense to distinguish between *remote* and *local* access to shared memory. Shared registers may be locally-accessible as a result of coherent caching, or when using distributed shared memory where shared memory is physically distributed among the processors.

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

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

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

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

3 The First Algorithm

Our first algorithm has the following properties: (1) its exit code is wait-free, (2) it satisfies strong FIFO fairness, (3) it has constant RMR complexity in both the CC and the DSM models, (4) it does not require to assume that the number of participating processes, n, is a priori known, (5) it uses only $O(n)$ shared memory locations, (6) it makes no assumptions on what and how memory is allocated, and (7) it uses exactly the same atomic instruction set as the original MCS algorithm.

3.1 An informal description

The algorithm maintains a queue of processes which is implemented as a linked list. Each *node* in the linked list is an object with pointer field called *next*, boolean field called *locked*, and status bit called *status*. Each process p has its own *two* nodes (i.e., elements), called $q_p[0]$ and $q_p[1]$, which in a DSM machine can be assumed to be stored in process p's local memory. In addition, a shared object called T (tail), points to the end of the queue.

Each time a process p wants to enter its critical section it uses alternately one of its two nodes. In its entry code a process threads itself (i.e., its node) to the end of the queue. Afterwards, p checks its state which can be one of the following: (1) it is alone in the queue, (2) its predecessor is in its exit section, or (3) its predecessor is either in its entry or critical section. In the first two cases, p can safety enter its critical section, in the later case p spins locally on its boolean *locked* field until it gets a signal from its predecessor that it is now at the head of the queue. Once p is at the head of the queue it can enter its critical section.

In its exit code, a process signals to its successor to enter its critical section. The main challenge is in implementing the part of the algorithm in which a releasing process signals its successor, since the threading cannot be done in one atomic operation and requires several remote accesses to the shared memory. This includes making T point to this process' node and making the process' predecessor know the threaded process is its successor.

In the MCS algorithm, to prevent a race condition, the releasing process is required in its exit code to wait until the threading is completed, and only then it may signal its successor and exit. As a result, the exit code of the MCS algorithm is *not* wait-free. To resolve this problem, we had to deal with a situation where the releasing process is in its exit section, but since the threading of its successor has not been completed yet the releasing process does not know who is its successor and thus has no way to signal anything directly to its successor (unless it waits for the threading to be completed, which is not an option in our case).

So, in its exit code p first assigns the value *unlocked* to its *status* variable. Since p may not know who is its successor, this assignment leaves a signal for p's successor that it may enter its critical section. However, this signal is done in p's memory space, so its successor cannot simply spin and wait for this signal, and checks for this signal only once.

Then, p checks if the threading of its successor (if there is one) is completed. If it isn't, there are two possibilities: (1) p is alone in the queue in which case p completes its exit code, or (2) p is not alone, in which case its successor will check and notice the signal p left in p's *status* variable. If the threading of its successor is completed, again there are two options: (1) the successor has already noticed the signal in p's *status* variable in which case p completes its exit code, or (2) its successor is spinning locally on its *locked* bit, in which case p terminates the waiting by setting its successor's *locked* bit to false. There are several race conditions which are resolved using `compare-and-swap` operations as explained later. The reason for using two nodes for each process is explained in details in Subsection 3.3.

3.2 The code of the algorithm and a detailed description

In the algorithms, the following symbols are used: "$*$" to indicate pointer of a specified type, "&" to obtain an object's address, and "." (dot) for integrated pointer dereferencing and field access. The code of the algorithm appears in Figure 1. A detailed explanation follows.

We start with the entry code. In **line 1**, out of its two nodes, p chooses the node to use in the current iteration, by inspecting *current*'s value. We notice that when p finishes the iteration, it toggles the value of *current* (in **line 16**). In **line 2**, p initializes its current node *next* pointer to NIL. During the execution *next* points to p's successor in the queue, and p

Algorithm 1 Program for process p.

Type: *QNode*: {next: QNode*, locked: bool, status ∈ {LOCKED, UNLOCKED}}
Shared: T: type QNode*, initially NIL // T points to the last item in the queue
$q_p[0,1]$: type QNode, both nodes initially {NIL, false, LOCKED}
// queue nodes belong to process p, and local to process p in the DSM model
Local: *pred*: type QNode*, initial value immaterial // process' predecessor
succ: type QNode*, initial value immaterial // process' successor
mynode: type QNode*, initially value immaterial // currently used node
current: ∈ {0,1}, initial value immaterial // index to current node

```
   Enter Code:
1  mynode := &q_p[current]                  // current node for this round, doorway begins
2  mynode.next := NIL
3  mynode.status := LOCKED
4  pred := FAS(T, mynode)                   // enter the queue, doorway ends
5  if pred ≠ NIL then                       // enter CS if no predecessor
6    mynode.locked := true                  // prepare to wait
7    pred.next := mynode                    // notify your predecessor
8    if CAS(pred.status, UNLOCKED, LOCKED) = false then
9      await (mynode.locked ≠ true) fi fi   // wait for your predecessor's signal
   Critical Section
   Exit Code:
10 mynode.status := UNLOCKED                // notify successor it can enter its CS
11 if mynode.next = NIL then                // if you don't have a successor
12   CAS(T, mynode, NIL)                    // set T back to NIL if you are last
13 else if CAS(mynode.status, UNLOCKED, LOCKED) then   // there is a successor
14       succ := mynode.next
15       succ.locked := false fi fi         // notify successor it can enter its CS
16 current := 1 − current                   // toggle for further use
```

has no successor yet. Later, the successor of p, if there is one, will update p's *next* pointer in **line 7**, and p will identify whether it has a successor, when executing **line 11**. In **line 3**, p initializes its current node *status* field to LOCKED. In **line 4**, p gets T's value, and assigns a pointer to its node into T. This line is the last line of the doorway, and it is where p threads its node to the queue and gets a pointer to its predecessor node (if there is one). In **line 5**, p validates whether it has a predecessor. If it doesn't, it means that p is first in the queue and can safely enter its critical section. If p has a predecessor, say q, it continues to **line 6**, where it initializes its *locked* variable to *true*, which means p cannot enter its critical section at the moment. In **line 7**, p notifies q that p itself is its successor. In **line 8**, p checks if q already enabled it to enter its critical section, by assigning UNLOCKED to *status*. If the *compare-and-swap* operation succeeds, p knows q already executed **line 10**, and exited its critical section, so p can enter its critical section. In case the *compare-and-swap* fails, p waits for its turn to enter its critical section in **line 9** by local-spinning on its *locked* variable, waiting for q to assign false to it (**line 15**).

Next we explain the exit code. In **line 10**, p assigns UNLOCKED to its *status* variable immediately after it finishes executing its critical section. At that time, p may not know who is its successor, so the first operation in the exit code is to leave a signal for the upcoming

successor, so that it will be able to enter its critical section when the time comes. In **line 11**, p checks if its successor (if there is one) has already notified who it is (that is, if its successor already executed **line 7**). If it didn't, p may be the only one in the queue. In **line 12** p checks whether T equals $mynode$, which is the node p inserted to the queue in **line 4**. If the *compare-and-swap* succeeds, p is indeed alone in the queue, so it assigns NIL to T, which returns the queue to its initial state. If the *compare-and-swap* in **line 12** fails, it means that there must be another process in the queue after p. Since p assigned UNLOCKED to its *status* variable, its successor should notice it and be able to enter its critical section later on. If p's successor has executed **line 7** before p has executed **line 11**, p will know who its successor is. In **line 13**, p checks whether its successor already let itself enter into its critical section after reading p's UNLOCKED value in **line 8**. If the successor hasn't done so yet, the *compare-and-swap* operation succeeds, and p lets its successor enter its critical section in **lines 14-15** by setting the bit its successor spins on to false. In **line 16**, p toggles its *current*, for the next iteration. The toggle is necessary to avoid deadlock.

3.3 Further explanations

In order to better understand the algorithm, we explain below three delicate design issues which are crucial for avoiding deadlocks.

1. *Why each process p needs two nodes $q_p[0]$ and $q_p[1]$?* The *current* variable is used to avoid deadlock in the following execution: assume each process has one node instead of two. Suppose process p is in its critical section, and process q finished its doorway. p resumes and executes its exit code (lines 10, 11). p finishes its exit code while q is in the queue, but q hasn't informed p who it is yet. p leaves its *status* variable with the value UNLOCKED, so that q will be able to enter its critical section. p starts another iteration before q resumes and executes line 4. Another process q' executes its entry code, such that q' is p's successor. Notice that, in that execution, q and q' share p's node as their predecessor (from two different iterations of p). If q' executes line 7 before q, q can override the assignment of q' and assigns a pointer to its node into p's *next* variable. q' now moves on and waits in line 9, but there is no process to free q' and a deadlock occurs. This problem is resolved by having each process own two nodes. We only need two nodes for each process, since the algorithm satisfies FIFO. p's successor enters its critical section before p enters its critical section in its next iteration. After p's successor enters its critical section, p's node won't be needed anymore, and p will be able to reuse it.
2. *Is the order of lines 6 and 7 important?* Line 6 must be executed before line 7, and their order should not be changed. Assume we have two processes, p and q, where q is p's predecessor and we change the order of lines 6 and 7. We let p execute line 7 and suspend it before executing line 6. In such a case, q can start executing its exit code. q will notice it has a successor, and q will move on to execute line 15. Then, p will continue to execute its code, and executes line 6. p assigns *true* to *locked* and misses q's signal to enter its critical section, and a deadlock occurs.
3. *Is the order of lines 10 and the test in the if statement in line 11 important?* Line 10 must be executed before the exited process checks in line 11 whether it has a successor. If the UNLOCKED assignment is executed afterwards, a deadlock may occur. Assume we have two processes, p and q, where q is p's predecessor. q executes the first line of the exit code, which is (after switching) "**if** $mynode.next = \text{NIL}$ **then**". Assume p hasn't executed line 7 yet, so the condition is true. Meanwhile, process p executes lines 5-9 and waits at line 9. Process p cannot skip the *compare-and-swap* since q's *status* variable is LOCKED. q finishes its exit code without signaling to p, and thus p will spin in line 9 forever, causing a deadlock.

3.4 Correctness proof

The following notions and notations are used in the proof.

1. **Doorway**: Process p is considered to be in its *doorway* while executing statements 1-4.
2. **The ith iteration**: Process p during its ith iteration (i.e, its ith attempt to enter its critical section) is denoted by p^i.
3. **Follows, predecessor, successor**: Consider an execution e. q^j *follows* p^i in e if and only if p^i finishes its doorway before q^j. p^i is the *predecessor* of q^j in e if and only if q^j follows p^i, and no other process finishes its doorway between the time p^i finished its doorway to the time q^j finishes its doorway. If p^i is the *predecessor* of q^j then q^j is said to be the *successor* of p^i.

▶ **Lemma 1.** *For every process p at iteration i, p^i has at most one predecessor.*

Proof. The fact that a process may have only a single predecessor, follows from that fact that the last step of the doorway (line 4) is an *atomic* fetch-and-store operation which updates T and *pred*. ◀

▶ **Lemma 2.** *Assume that for every p^i and q^j, if p^i is the predecessor of q^j then p^i enters its critical section before q^j enters its critical section. Then, for every p^i and q^j, if p^i is the predecessor of q^j then p^i enters its critical section before any process that follows q^j enters its critical section.*

Proof. Proof by induction on the number of processes m that follow q^j. In the base case, when $m = 1$, there is only one process, say r^k, that follows q^j. This means that q^j is the predecessor of r^k. Therefore, according to the assumption made in the first part of the lemma, q^j enters its critical section before r^k enters its critical section. Since p^i enters its critical section before q^j, and q^j enters its critical section before r^k, by transitivity p^i enters its critical section before r^k.

We assume that the lemma holds for $m-1$ processes that follow q^j, and prove that it also holds for the mth process that follows q^j. Let the mth process be r^k. We denote r^k's predecessor as $\hat{r^k}$. Notice that $\hat{r^k}$ is the $m-1$ process that follows q^j. Thus, by the induction hypothesis, p^i enters its critical section before $\hat{r^k}$ enters its critical section. $\hat{r^k}$ is the predecessor of r^k, and thus, according to the assumption made in the first part of the lemma, $\hat{r^k}$ enters its critical section before r^k enters its critical section. Since p^i enters its critical section before $\hat{r^k}$ enters its critical section and $\hat{r^k}$ enters its critical section before r^k enters its critical section, by transitivity p^i enters its critical section before r^k. ◀

▶ **Lemma 3.** *For every p^i and q^j such that p^i is the predecessor of q^j, p^i is the only process that can assign false to q^j's mynode.locked.*

Proof. By Lemma 1, q^j has at most one predecessor. So, p^i is q^j's only predecessor. Clearly, except for p^i, any other process that q^j follows will not be able to write to q^j's *mynode.locked*. Thus, throughout the algorithm, the only processes that write to q^j's *mynode.locked* are q^j itself and p^i. q^j does it at line 6 and p^i does it at line 15. At line 6, q^j assigns true to *locked*, thus, the only process that assign false is p^i. ◀

▶ **Lemma 4** (STRONG FIFO). *If p^i finishes its doorway before q^j (begins or) finishes its doorway, then p^i enters its critical section before q^j enters its critical section.*

Proof. p^i finished the doorway before q^j finishes the doorway, therefore p^i executes line 4 before q^j does. There are two options:

1. q^j is p^i's successor.
2. q^j follows p^i, but q^j is not p^i's successor.

According to Lemma 2, once we prove that p^i enters its critical section before q^j in case 1, then it would immediately follow that p^i enters its critical section before q^j in case 2 as well.

Assume q^j is p^i's successor and assume to the contrary that q^j enters its critical section before p^i. There are two cases:

1. q^j executes line 4 after p^i executes line 12. In which case, p^i entered its critical section before q^j, which contradicts the assumption.
2. q^j executes line 4 before p^i executes line 12. Therefore, T still points to p^i's *qnode* when q^j executes line 4 and q^j gets p^i's *qnode* to its *pred* variable. Thus *pred* is not NIL, and q^j continues and executes line 8. Here we have two options as well:
   a. The *compare-and-swap* operation in line 8 succeeds. The *compare-and-swap* succeeds only if q^j's *pred.status* is equal to UNLOCKED. *pred* is p^i's *qnode*, p^i assigned its status LOCKED value at the beginning of p^i's entry code. For it to change to UNLOCKED, p^i should execute line 10 in the exit code. This means that p^i entered its critical section before q^j did, contradicting the assumption.
   b. The *compare-and-swap* operation in line 8 fails. q^j continues to line 9 and local-spins on *locked*. The only way q^j can enter its critical section is when some other process writes false to *locked*. According to Lemma 3, p^i is the only process that can assign false to q^j's *locked*. Notice that the only line where p^i does this is at line 15, which is in its exit code. In this case, p^i entered its critical section before q^j, contradicting the assumption.

We proved that if p^i finishes the doorway before q^j finishes the doorway, then there is no valid scenario where q^j enters its critical section before p^i. Therefore, p^i enters its critical section before q^j does, implying that the algorithm satisfies strong FIFO. ◀

▶ **Lemma 5.** *For every p^i and q^j, if p^i is the predecessor of q^j such that p^i and q^j are not in their critical sections at the same time, then p^i and r^k are not in their critical sections at the same time, for any other process r^k that follows q^j.*

Proof. Proof by induction on the number of processes m that follow q^j. In the base case, $m = 1$: there is only one process, say r^k, that follows q^j. This means that q^j is the predecessor of r^k. Assume p^i is in its critical section. Since q^j is not in its critical section at the same time as p^i and by Lemma 4 the algorithm satisfies strong FIFO, therefore q^j enters its critical section after p^i. Also by Lemma 4, r^k enters its critical section after q^j. Therefore, r^k cannot be in its critical section at the same time as p^i. Next, we assume that the lemma holds for $m-1$ processes that follow q^j, and prove for m processes that follow q^j. Let the mth process be r^k, and let r^k's predecessor be $\hat{r^k}$. Notice that $\hat{r^k}$ is the $m-1$ process following q^j, and therefore by the induction hypothesis p^i and $\hat{r^k}$ are not in their critical sections at the same time. Thus, since by Lemma 4 the algorithm satisfies strong FIFO, r^k is not in its critical section at the same time as p^i. ◀

▶ **Lemma 6** (Mutual Exclusion)**.** *No two processes are in their critical sections at the same time.*

Proof. We assume that there are two processes, p^i and q^j at their critical sections at the same time, and show it leads to a contradiction. Assume, without loss of generality, that p^i enters its critical section before q^j. According to Lemma 4 the algorithm satisfies strong FIFO, so p^i finishes its doorway before q^j, and therefore p^i executes line 4 before q^j. There are two options:

1. q^j is p^i's successor.
2. q^j follows p^i, but q^j is not p^i's successor.

By Lemma 5, once we prove that p^i and q^j are not in their critical sections at the same time in case 1, then it would immediately follow that p^i and q^j are not in their critical section at the same time in case 2. Let's prove case 1: Assuming q^j is p^i's successor, there are two options:

1. q^j executes line 4 after p^i executes line 12. In this case, p^i exits its critical section before q^j enters its critical section, which contradicts the assumption.
2. q^j executes line 4 before p^i executes line 12. Therefore, T still points to p^i's *qnode* when q^j executes line 4 and q^j gets p^i's *qnode* assigned to its *pred* variable. Thus *pred* is not NIL, and q^j continues and executes line 8. In line 8, q^j checks whether p^i's status is UNLOCKED. Since p^i executed line 4 before q^j executed line 4, p^i also executed line 3 before q^j executed line 8. According to the assumption p^i still hasn't exited its critical section, so its status is still LOCKED. Therefore the *compare-and-swap* fails and q^j continues to line 9. q^j assigned true to its *locked* variable in line 6, and local-spins in line 9, waiting for some process to let it enter its critical section. By Lemma 3, p^i is the only process that can assign false to q^j's *locked*. p^i does this at line 15, which means that for q^j to enter its critical section, p^i has to execute its exit code. This contradicts the assumption that p^i and q^j are in their critical sections at the same time. ◀

▶ **Lemma 7.** *For every p^i and q^j such that q^j is p^i's predecessor, once q^j assigned the value false to p^i's locked variable, this value cannot be overwritten, until p^i completes its exit code.*

Proof. By Lemma 1, p^i has only one predecessor, so q^j is p^i's only predecessor. Throughout the algorithm, the only processes that write to p^i's *locked* variable are p^i itself and q^j. p^i does this at line 6 and q^j does this at line 15. We will show that if q^j executes line 15, p^i must have already executed line 6 before that. q^j executed line 15, therefore q^j must have executed the *"else"* clause of the *"if"* statement at line 11. So the *"if"* condition was false, and q^j's *mynode.next* was not equal to NIL. q^j has only one successor which is p^i. Since q^j's *next* variable was not NIL, p^i has already executed line 7 and assigned its node to q^j's *next*. This implies that p^i has already executed line 6, in which it assigns true to *locked*. Therefore p^i executed line 6 before q^j executed line 15, and since there is no other process that writes to p^i's *locked* variable, the (false) value was not overwritten. ◀

▶ **Lemma 8** (Deadlock-freedom). *If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section.*

Proof. Assume to the contrary that some group of processes P are in their entry code and none of them can ever access its critical section. Let p^i be the first process in P to complete its doorway. Thus, all the other processes in P follow p^i. The fact that p^i is not being able to enter its critical section, means that p^i is local-spinning at line 9, since all the other lines in its entry code do not contain any loops, are wait-free and can be completed in a constant number of p^i steps. Any other execution path would lead p^i to its critical section and contradicts the assumption. From here it follows that:

- When p^i completed its doorway, *pred* was not NIL. Therefore, there exists process q^j such that q^j finished the doorway before p^i, but hasn't executed line 12 that removes it from the queue.
- p^i *compare-and-swap* operation at line 8 fails, therefore q^j *status* was equal to LOCKED. There are two options:
  1. q^j hasn't executed line 10.
  2. q^j has already executed line 13, where the *compare-and-swap* operation ended successfully for q^j.

Since we assumed that p^i is the first process to complete its doorway among the waiting processes, and q^j is p^i's predecessor, it follows that q^j would eventually be able to enter its critical section. Therefore, q^j will necessarily exit its critical section and begin the exit code. In both cases above, it is easy to see that the execution path of q^j in the exit code leads it to line 15, where q^j writes false to p^i's *locked* variable. In addition, according to Lemma 7, the value of p^i's *mynode.locked* field is never overwritten until p^i completes it exit code. Therefore, at some point p^i will notice that $mynode.locked = false$ and can continue to its critical section. This contradicts the assumption that p^i will not enter its critical section. ◀

▶ **Lemma 9** (Constant RMR complexity). *The RMR complexity of Algorithm 1 is $O(1)$ in both the CC and DSM models.*

Proof. By inspecting the algorithm, it is easy to count steps and see that except the busy-waiting loop in line 9, it takes constant number of steps for a process to enter and exit its critical section. Thus, it is sufficient to prove that, for every p^i, p^i performs $O(1)$ RMRs at line 9, because this is the only busy-waiting loop in the algorithm. We will prove that while the process is executing the loop at line 9, it performs only a constant number of remote memory references, in both models:

- **DSM model**: p^i spins on *mynode.locked*. *mynode* can be equal to either $q_p[0]$ or $q_p[1]$ as follows from line 1 in the algorithm. Both of them initialized as local to process p^i's memory and thus the algorithm performs $O(1)$ RMRs in the DSM model.
- **CC model**: We prove that in one iteration of a process, there is at most one cache invalidation. Before spinning on *mynode.locked*, its value migrates to p^i's local cache, since p^i assigned to it at line 6. It is updated by another process only once, at line 15. When a process updates *mynode.locked*, p^i will have a cache invalidation and p^i will execute one remote memory reference to read the new value of *mynode.locked*. Since the new value is necessarily equal to false, p^i stops spinning on *mynode.locked* and proceeds to its critical section. By Lemma 7, there is no other process that writes to p^i's *mynode.locked* in the current iteration anywhere else in the algorithm. Therefore, there is only one remote memory reference during the loop execution, and the algorithm has $O(1)$ RMR complexity in the CC model. ◀

▶ **Lemma 10** (Wait-free exit code). *Every process finishes its exit code within a bounded number of its own steps.*

Proof. Since the exit code is a straight-line code which does not contain either loops or await operations, it immediately follows that any execution of the exit code will be completed in a bounded number of a process' own steps. ◀

▶ **Theorem 11.** *Algorithm 1 satisfies mutual exclusion, deadlock freedom, wait-free exit, strong FIFO fairness, and constant RMR complexity. Furthermore, it does not require to assume that the number of processes, n, is a priori known, it uses only $O(n)$ shared memory locations, it makes no assumptions on what and how memory is allocated, and it uses exactly the same atomic instruction set as the original MCS algorithm.*

Proof. The properties mutual exclusion, deadlock freedom, wait-free exit, strong FIFO fairness and constant RMR complexity, follows from Lemma 6, Lemma 8, Lemma 10, Lemma 4 and Lemma 9, respectively. The other properties are easily verified by inspecting the code of the algorithm. ◀

4 The Second Algorithm

The second algorithm satisfies the same properties as the first algorithm, as listed at the beginning of Section 3. In addition, in the second algorithm it is possible to recycle memory locations so that if there are L locks, and each process accesses at most one lock at a time, the algorithm needs only $O(L+n)$ space, as compared to $O(Ln)$ needed by the first algorithm.

To simplify the presentation, we will assume that processes have unique identifiers. However, for each process p, the value of one of its pointers is a unique number which can be used as process p's unique identifier instead of assuming that p itself is the unique identifier. We elaborate more on this issue in Subsection 4.3, after the algorithm is presented.

4.1 An informal description

As in the previous case, the algorithm maintains a queue of processes which is implemented as a linked list. Each process p has two *different* data elements that complete each other and together represent a single node. The two data elements are called: q_p which resides in p local memory and access to it is considered local access, while $node_p$ is handed over from an exiting process to its successor at the end of the exit code, and access to it is considered remote memory access. q_p, which is not part of the queue, is a record with pointer field called *qnode* which initially points to the $node_p$ element, and boolean field called *locked*. $node_p$ is the element that initially p tries to thread into the linked list in its entry code.

In addition, a shared object called T (tail), points to the end of the queue. Initially, when the queue is empty, T points to a *dummy* node, called $node_0$, that enables the first process which succeeds to enter the queue, to proceed to its critical section.

In its entry code a process threads itself (i.e., thread the element $q_p.qnode$ points to) to the end of the linked list. A process has several ways to enter its critical section: it can enter immediately if it is alone in the queue or if its predecessor is in its exit section, otherwise, it has to spin locally until its predecessor assigns false to *locked*.

In the exit code p first assigns the value p (its ID which is different than 0) to its *status* field of its node in the linked list. Since, p may not know who is its successor, this assignment leaves a signal for p's potential successor that it may enter its critical section. Then, p checks if it has a successor. If it doesn't then p completes its exit code. Otherwise, if p's successor has already noticed p's *status* variable equals p, process p completes its exit code. If its successor is spinning locally on its *locked* bit, p terminates the waiting by setting its successor *locked* bit to false. In *all* the above cases, p always leaves its current node in the queue (because this node includes the *status* field with the value p which indicates that its critical section is free) and, before p completes its exit code, it removes from the queue and takes ownership of the node of its predecessor (which is the current dummy node).

4.2 The code of the algorithm and a detailed description

We assume that each process has a unique identifier which is different than 0. The code of the algorithm appears in Figure 2. It is important to notice that there are some statements in the algorithm with multiple memory references. We use this style to keep the algorithm short and simple. Only one shared memory location can be accessed in one atomic step! For example, the statement in line 1, "$q_p.qnode.next$:= NIL" is equivalent to "$localTemp := q_p.qnode$; $localTemp.next$:= NIL", and the statement in line 7 "$pred.next := q_p.qnode$" is equivalent to "$localTemp := q_p.qnode$; $pred.next := localTemp$". A detailed explanation follows.

Algorithm 2 Program for process p.

Type: $QNode$: {next: QNode*, local: LocalNode*, status: integer, pid: integer}
$LocalNode$: {qnode: QNode*, locked: bool}
Constant:
ZERO = 0 // it is assumed that 0 is not a process id
Shared: $node_0$: type QNode, initially {NIL, NIL, 0, 0}
// a dummy node, enables the first process to enter its critical section
T: type QNode*, initially $\&node_0$ // T points to $node_0$
$node_p$: type QNode, initial values are immaterial
q_p: type LocalNode*, qnode initially $\&node_p$, locked initial value is immaterial
// q_p belongs to process p, and local to process p in the DSM model
Local: $pred$: type QNode*, initial value is immaterial // process' predecessor
$predPid$: type integer, initial value is immaterial // process' predecessor ID

Enter Code:
1 $q_p.qnode.next :=$ NIL // initialization, doorway begins
2 $q_p.qnode.pid := p$ // process ids are unique and different than 0
3 $q_p.qnode.local = q_p$
4 $q_p.qnode.status :=$ ZERO
5 $q_p.locked := true$
6 $pred :=$ **FAS**$(T, q_p.qnode)$ // enter the queue, doorway ends
7 $pred.next := q_p.qnode$ // notify your predecessor
8 $predPid := pred.pid$
9 **if CAS**$(pred.status, predPid,$ ZERO$) = false$ **then**
10 **await** $(q_p.locked \neq true)$ **fi** // wait for your predecessor's signal
Critical Section
Exit Code:
11 $q_p.qnode.status := p$ // notify successor it can enter its CS
12 **if** $q_p.qnode.next \neq$ NIL **then** // if you have a successor
13 **if CAS**$(q_p.qnode.status, p,$ ZERO$)$ **then**
14 $q_p.qnode.next.local.locked := false$ **fi fi** // notify successor it can enter its CS
15 $q_p.qnode := pred$ // use predecessor's node for the next iteration

We start with the entry code. In **lines 1-5**, p initializes its node. Notice p initializes all of its fields except $q_p.qnode$, which was already initialized before the execution began. In **line 6**, p executes *fetch-and-store* to enter the queue. p gets T's value, which points to the last item in the queue (which is also p's predecessor), and assigns its *qnode* to T. Notice that p assigns only its $node_p$ to T, while q_p can be accessed via $node_p$ if needed. This is the end of the doorway. In **line 7**, p notifies its predecessor that p is its successor. In **line 8**, p copies its predecessor's process ID to a local variable, to be used as an argument to the *compare-and-swap* operation later. In **line 9**, p checks whether its predecessor has already signaled p to enter its critical section. The predecessor does it by assigning its process ID to its *status* in **line 11**. If the predecessor already assigned its process ID but did not change it back to ZERO yet (**line 13**) then the *compare-and-swap* succeeds and p can enter its critical section. If it fails, p continues to **line 10** and starts spinning locally on its *locked* variable, waiting for it to change to false so it can enter its critical section.

Next we explain the exit code. In **line 11**, p assigns its process ID to its *status* variable, and signals its potential successor that it can enter its critical section. In **line 12**, p checks whether it has a successor. If *next* equals NIL it doesn't have a successor and it continues to **line 15**. If *next* is not NIL, it means that some process q already assigned itself as p's successor at **line 7**. In such a case, p continues to **line 13**, checking whether q has already executed **line 9** and let itself enter its critical section. If q executed **line 9** and q's *compare-and-swap* ended successfully, p's *compare-and-swap* in **line 13** fails and p continues to **line 15**. If p's *compare-and-swap* ends successfully, it means q hasn't entered its critical section yet. Since p assigned ZERO to its *status* in **line 13**, p must let q enter its critical section by setting *locked* to false, and does so at **line 14**. In **line 15**, p assigns its predecessor's node to itself, and leaves its node to its successor. On p's next iteration, it will use its predecessor's node, thus p will not override the *status* in its previous node when initializing its *qnode* at the beginning of its next iteration. Therefore, p's successor will be able to read and use p's *status* when needed and find out that it can enter its critical section.

4.3 Further explanations

In order to better understand the algorithm, we explain below several crucial design issues.

1. *Do we really need to explicitly assume that the processes have unique identifiers?* No, this is done only to simplify the presentation. In the first algorithm, it is not assumed that processes have unique identifiers. However, each process p has two unique memory nodes $q_p[0]$ and $q_p[1]$, and it is possible to consider $\&q_p[0]$ as the unique identifier of process p. Similarly, in algorithm 2, the value of the pointer q_p is a unique number which can be used as process p's unique identifiers. That is, in Algorithm 2, it is possible to replace p with q_p (assuming $q_p \neq 0$) everywhere (i.e., in lines 2,11,13). This implies that also in Algorithm 2 there is no need to explicitly assume that processes have unique IDs.
2. *How does a process that does not need to spin know that it is at the head of the queue?* Whenever the values of the *status* field and of the *pid* field, of the first node (i.e., the dummy node) in the queue are equal, the process its node is the successor of the dummy node can safely enter its critical section. Initially, $node_0$ is the dummy node and $node_0.status = node_0.pid = 0$, thus, the first process that threads itself into the queue, can immediately enter its critical section. Also, when a process, say p, completes its critical section, it always leaves its *current* node in the queue and takes ownership of the node of its predecessor (which is the dummy node). Thus, its current node becomes the new dummy node with both *status* and *pid* fields equal p, which will not block the next process in line, since the *compare-and-swap* operation in line 9 would succeed.
3. *Can't we simply use UNLOCKED in lines 2,11 and 13, as done in algorithm 1, instead of using the unique process identifier p?* No, this is crucial for avoiding deadlocks. We explain it by example. Consider the following scenario: There are two processes, p and q.
   - p starts first and enters its critical section;
   - q starts, run until after line 7 and becomes p's successor in the queue;
   - p executes the code until after line 12. Since $next \neq$ NIL, p enters the *if* statement;
   - q continues and enters its critical section;
   - q continues, finishes its exit code and takes p's current *qnode* for its next iteration;
   - q starts its entry code, and executes until after line 11; p and q has the same *qnode*!

   Now, here is the difference between using process identifiers and UNLOCKED: p continues,
   - When process identifiers are used, the *compare-and-swap* operation in line 13 *fails* and p takes its predecessor's *qnode* for its next iteration and completes its exit code.

- When UNLOCKED is used, the *compare-and-swap* operation in line 13 *succeeds* for p and the shared qnode for p and q now contains the status ZERO. p will continue to line 14, and p will complete its exit code. q executes its exit code and since it has no successor, it skips lines 12-14 and exits. We got into a situation where the queue is empty, and T points to a dummy node with status value ZERO! The next process to enter will be spinning in line 10 forever.

The structure of the correctness proof of Algorithm 2 is similar to that of Algorithm 1.

5 Related Work

Mutual exclusion algorithms were first introduced by Edsger W. Dijkstra in [3]. Since then, numerous implementations have been proposed [15, 7, 17]. The first queue-based local-spinning mutual exclusion algorithms for the CC model were presented in [1, 6]. The algorithm from [1] used the `fetch-and-increment` operation, while the algorithm from [6] used the `fetch-and-store` (swap) operation. In these two algorithms different processes may spin on the same memory location at the different times. Their RMR time complexity in the CC model is a constant, while their time complexity in the DSM model is unbounded.

The famous MCS algorithm is from [13]. Unlike the previous two algorithms, the MCS algorithm satisfies local spinning in *both* the CC model and the DSM model. In [9], a simple correctness proof of the MCS lock is provided. An extension of the MCS Algorithm that solves the readers-writers problem is presented in [14]. In [16] queue-based algorithm is presented, which uses unbounded space, in which it is possible for a spinning process to "become impatient" and leave the queue before acquiring the lock. A recoverable version of the MCS algorithm, in which processes can fail and recover, was presented recently [4].

Another queue-based lock was developed by Craig [2] and, independently by Magnusson, Ladin and Hagersten [11, 12]. As the MCS lock, the queue is implemented as a linked list, but with pointers from each process to its *predecessor*. The algorithm uses `fetch-and-set` operations and may outperform the MCS lock on cache-coherent machines. Its time complexity in the CC model is a constant, while its time complexity in the DSM model is unbounded.

A variant of the above algorithm from [2], with constant time complexity in both the CC and the DSM models was presented in [2]. For this variant to work, it must be assumed that *all* allocated pointers point to *even* addresses. This assumption enables to pack two shared registers into a single 32-bit word so that it is possible to atomically swap the two registers as a unit. In [10], four local-spin mutual exclusion algorithms for the DSM model using `fetch-and-set` operations were presented; all these algorithms use arrays of fixed size, assume that the number of processes is a priori known, and are not suitable for a model where processes may appear or disappear intermittently.

6 Discussion

We have presented two new mutual exclusion algorithms, that overcome a drawback of the famous MCS algorithm, while preserving its simplicity, elegance and properties. It would be interesting to design additional new algorithms, which would be based on our algorithms, that implement other types of locks, such as readers-writers locks [14], abortable locks [8, 16] and recoverable locks [4, 5], and would have constant RMR complexity, satisfy the wait-free exit code property and other desired properties.

References

1. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):6–16, January 1990.
2. T.S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report TR-93-02-02, Dept. of Computer Science, Univ. of Washington, February 1993.
3. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
4. W. Golab and D. Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing*, pages 211–220, 2017.
5. W. Golab and A. Ramaraju. Recoverable mutual exclusion. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 65–74, 2016.
6. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computers*, 28(6):69–69, June 1990.
7. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, 2008. 508 pages.
8. P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proc. 22nd ACM Symp. on Principles of Distributed Computing*, pages 295–304, July 2003.
9. T. Johnson and K. Harathi. A simple correctness proof of the MCS contention-free lock. *Information Processing Letters*, 48(5):215–220, 1993.
10. H. Lee. Local-spin mutual exclusion algorithms on the DSM model using fetch&store objects. Mater thesis, University of Toronto, 2003.
11. P.S. Magnusson, A. Landin, and E. Hagersten. Efficient software synchronization on large cache coherent multiprocessors. Technical Report T94:07, Swedish Institute of Computer Science, February 1994.
12. P.S. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the 8th International Symposium on Parallel Processing*, pages 165–171, April 1994.
13. J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Systems*, 9(1):21–65, 1991.
14. J.M. Mellor-Crummey and M.L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices*, 26(7):106–113, 1991.
15. M. Raynal. *Algorithms for mutual exclusion.* The MIT Press, 1986. Translation of: Algorithmique du parallélisme, 1984.
16. M.L. Scott. Non-blocking timeout in scalable queue-based spin locks. In *Proc. 21th ACM Symp. on Principles of Distributed Computing*, pages 31–40, July 2002.
17. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Pearson / Prentice-Hall, 2006. ISBN 0-131-97259-6, 423 pages.