Closing the Complexity Gap between FCFS Mutual Exclusion and Mutual Exclusion *

Robert Danek and Wojciech Golab

Department of Computer Science
University of Toronto
{rdanek,wgolab}@cs.toronto.edu

Abstract. First-Come-First-Served (FCFS) mutual exclusion (ME) is the problem of ensuring that processes attempting to concurrently access a shared resource do so one by one, in a fair order. In this paper, we close the complexity gap between FCFS ME and ME in the asynchronous shared memory model where processes communicate using atomic reads and writes only, and do not fail. Our main result is the first known FCFS ME algorithm that makes $O(\log N)$ remote memory references (RMRs) per passage and uses only atomic reads and writes. Our algorithm is also adaptive to point contention. More precisely, the number of RMRs a process makes per passage in our algorithm is $\Theta(\min(k, \log N))$, where k is the point contention. Our algorithm matches known RMR complexity lower bounds for the class of ME algorithms that use reads and writes only, and beats the RMR complexity of prior algorithms in this class that have the FCFS property.

1 Introduction

Coordinating access to shared resources is a key problem in programming multiprocessors. Mutual exclusion [1], also known as locking, is the approach most popular in practice for allowing multiple processes to access a shared resource safely. We consider this problem under the customary assumptions that processes are asynchronous (i.e., execute at arbitrary speeds) but do not fail. A mutual exclusion algorithm for a shared memory multiprocessor consists of a *trying protocol* (TP) and an *exit protocol* (EP) that surround the critical section (CS). The latter contains code that actually accesses the shared resource. A single execution of the TP, CS, and EP is called a *passage*. When a process is not inside the TP, EP, or CS, we say that it is in the non-critical section (NCS).

The trying and exit protocols ensure that at most one process can be in the critical section at a time, while also guaranteeing that processes wanting to access the resource can eventually do so. We can state the correctness properties of a mutual exclusion algorithm more precisely as follows:

* Research supported in part by the Natural Sciences and Engineering Research Council of Canada.

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

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

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

Note that to satisfy lockout freedom, we must make the (standard) assumption that every process is *live*, meaning that as long as it is outside the NCS, it continues to take steps until it returns to the NCS.

The above properties do not preclude situations where a process waits inside the trying protocol for a long time while other processes are repeatedly granted entry to the critical section. This may be undesirable, and a mutual exclusion algorithm that grants processes entry into the critical section in an order that is more fair may be preferred. One form of fairness is captured by the First-Come-First-Served (FCFS) property [2], which informally requires that processes are granted entry into the critical section in the order in which they execute the beginning of the trying protocol. To define this more precisely, we split the trying protocol into two parts: the first part, the doorway (DWY), which a process completes in a bounded number of its own steps; and a second part, the waiting room (WRM). We can now define FCFS as follows:

First-Come-First-Served (FCFS): If a process p finishes the doorway before a process $q \neq p$ starts the doorway, then q does not enter the CS before p does in the corresponding passages.

A natural way to measure the time complexity of a mutual exclusion algorithm is to count the number of memory accesses performed during a passage. This is problematic in the asynchronous model as a process may execute an unbounded number of memory accesses while busy-waiting for another process to clear the critical section. Instead, we measure time complexity by counting only the *remote memory references* (RMRs) performed during a passage, where an RMR is a memory access that traverses the processor-to-memory interconnect. We refer to this measure as an algorithm's RMR complexity.

To classify memory accesses as local or remote, we consider two shared memory architectures: the Distributed Shared Memory (DSM) model, and the Cache-Coherent (CC) model [3]. In the DSM model, each processor is associated with a memory module that it can access locally, and that others may access only remotely. In the CC model, on the other hand, any memory location can be made locally accessible by storing its contents in a local cache, which is kept up to date (by either updating or invalidating stale entries) by a *coherence protocol*. Different varieties of the CC model exist, all satisfying the following property under ideal conditions: once a variable is loaded into a cache, it remains cached at least until it is overwritten by another process.

Algorithms that perform all busy-waiting using local memory references (e.g., repeatedly testing the value of a cached variable) are known as *local-spin*; they

have bounded RMR complexity and offer practical performance benefits [4]. The RMR complexity of an ME algorithm may depend on the number of processes contending for access to the CS. *Point contention* describes this quantity precisely; for our purposes it is defined as the maximum number of processes simultaneously outside of the NCS during an execution fragment. An ME algorithm whose RMR complexity grows gradually with point contention is known as *adaptive* (to point contention).

Summary of results. Our main technical contribution is an FCFS ME algorithm based on reads and writes only, which has RMR complexity $O(\min(k, \log N))$ when point contention is k and there are N processes. The complexity of our algorithm is optimal, at least when $k \in O(\log \log N)$ [5] or $k \in \Theta(N)$ [6]. Prior algorithms either require stronger synchronization primitives, lack the FCFS property, or have suboptimal RMR complexity.

Our algorithm uses as building blocks two novel wait-free components: a set-like data structure called *SpecialSet*, and a ticket dispensing mechanism. The *SpecialSet* records a set of process IDs, and has two operations: InsertSelf(), which a process can use to add its ID to the set, and RemoveSelf(), which a process can use to remove itself from the set and also to learn the ID of exactly one other process in the set (if any). Our *SpecialSet* and the ticket dispenser are accessed according to certain restrictions on parallelism, which simplifies their implementation.

As a complexity upper bound, our algorithm has several implications regarding mutual exclusion:

(1) The worst-case RMR complexity of FCFS ME using only reads and writes is no greater than for ordinary (i.e., deadlock-free) ME; both problems are solvable using $O(\log N)$ RMRs per passage, matching the recent lower bound of Attiya, Hendler and Woelfel [6].
(2) FCFS ME can be solved using only reads and writes with RMR complexity adaptive to point contention, matching the linear lower bound of Kim and Anderson for $k \in O(\log \log N)$ [5] in addition to the logarithmic worst-case lower bound [6].
(3) As a consequence of (1) and (2), and the fact that the lower bounds on ME RMR complexity [6, 5] hold even if comparison primitives (such as Compare-And-Swap) are available, FCFS ME and adaptive FCFS ME are no more costly to solve (in terms of RMRs) using reads and writes only than using reads, writes, and comparison primitives. This strengthens somewhat a prior result of Golab, Hadzilacos, Hendler and Woelfel [7], which implies the analogous conclusion for ME algorithms that do not have FCFS or bounded exit.

2 Related Work

The mutual exclusion problem was first solved by Dijkstra [1], although his solution did not satisfy lockout freedom. Rather, it satisfied a weaker progress property, called deadlock freedom:

Deadlock Freedom: If some process p is stuck forever in the trying protocol, then some process $q \neq p$ executes through the critical section infinitely often.

FCFS mutual exclusion was first formulated and solved by Lamport in [2], where he presented his famous Bakery algorithm. Lamport [8] was also the first to study *fast* mutual exclusion. Fast mutual exclusion ensures that a process takes a constant number of steps entering the CS when there is no contention, but provides no performance guarantees otherwise. In *adaptive* mutual exclusion the performance of an algorithm instead degrades gradually as the contention for the CS increases. Adaptive mutual exclusion algorithms were presented by Styer [9], Choy and Singh [10], and Attiya and Bortnikov [11]. These algorithms are adaptive to metrics different from RMR complexity, and moreover, the RMR complexity of these algorithms is unbounded.

Yang and Anderson (YA) [12] presented the first mutual exclusion algorithm that uses only reads and writes and has RMR complexity $O(\log N)$. Kim and Anderson [13] (KA) later presented an adaptive mutual exclusion algorithm, also using only reads and writes, that used as building blocks parts of Lamport's fast mutual exclusion algorithm and the YA algorithm. Its RMR complexity is $O(\min(k, \log N))$, where k denotes point contention. This improves upon the adaptive algorithm of Afek, Stupp and Touitou [14].

Several lower bounds exist for the RMR complexity of mutual exclusion and adaptive mutual exclusion. Kim and Anderson [5] showed that the RMR complexity of adaptive ME algorithms based on reads and writes only must grow at least linearly with point contention up to $\Omega(\log \log N)$, which is matched by algorithm KA. Attiya, Hendler and Woelfel [6] later showed that the worst-case RMR complexity for the same class of algorithms is $\Theta(\log N)$, which is matched by algorithm YA. (This builds on prior results by Cypher [15], Anderson and Kim [16], and Fan and Lynch [17].) A related result by Golab, Hadzilacos, Hendler and Woelfel [7] implies that the $\Theta(\log N)$ lower bound is tight also for algorithms that use comparison primitives, such as Compare-And-Swap (CAS), and do not require FCFS or bounded exit.

Jayanti [18] presented the first FCFS adaptive mutual exclusion algorithm. It has RMR complexity $O(\min(k, \log N))$, and makes use of LoadLinked and StoreConditional – a pair of synchronization primitives stronger than reads and writes.

Taubenfeld [19] also presented an FCFS adaptive mutual exclusion algorithm. This algorithm is a modification of Lamport's Bakery algorithm, and uses only reads and writes. Its RMR complexity, however, is $O(k^2)$, which is suboptimal in light of our results.

3 FCFS Algorithm and High-level Description

Our algorithm (shown below in Figure 1) has the following high-level structure. In the doorway, a process receives a ticket from a wait-free ticket dispenser (line 4) that incurs $O(\min(k, \log N))$ RMRs per invocation. The dispenser is

similar to a modular atomic counter, which returns tickets with increasing values from a bounded interval. As the dispenser is not actually atomic, processes that invoke the dispenser concurrently may receive the same ticket. Also, even though the dispenser returns tickets from a bounded interval, the interval is large enough to ensure that tickets are not reused too soon. After a process p obtains a ticket, it enters the waiting room (lines 5–16) where it adds itself to a priority queue (Q) ordered by ticket (line 11). To ensure that FCFS is not violated, p waits to reach the front of the queue before entering the CS (line 15). Once p is done with the CS, p removes itself from the queue (line 18), and notifies its successor (lines 20–21).

Figure 1: FCFS Mutual Exclusion Algorithm for process $p \in \{1, ..., N\}$

shared variables:
Set: *SpecialSet*, *Q*: *PriorityQueue*, *Head*: **array**$[1..N]$ **of Boolean**

(In the DSM model, *Head*$[p]$ is local to process p)

private variables:
ticket: $\{0, .., 7N-1\}$, *tmp_id*: **integer**

```
1  loop
2      NCS
3      Set.InsertSelf()                                  // Doorway begins.
4      ticket := ObtainTicket()                            // Doorway ends.
5      LOCK()
6          Head[p] := false
7          tmp_id := Set.RemoveSelf()
8          if tmp_id ≠ ⊥ then
               // Enqueue process tmp_id with ''dummy'' ticket.
9              Q.Insert((tmp_id, −1))
10         Q.Remove((p, −1))          // Remove (p, −1) from queue if present.
11         Q.Insert((p, ticket))        // Reinsert p with ''proper'' ticket.
12         tmp_id := Q.FindMin()       // Get the head process in the queue.
13         Head[tmp_id] := true           // Notify head process to advance.
14     UNLOCK()
15     await Head[p] = true          // Wait to reach the head of the queue.
16     LOCK()
17         CS                                          // The critical section.
18         Q.Remove((p, ticket))        // Remove p from the priority queue.
19         DoneWithTicket()
20         tmp_id := Q.FindMin()
21         if tmp_id ≠ ⊥ then
22             Head[tmp_id] := true        // Notify next process to advance.
23     UNLOCK()
24 end loop
```

We use an *auxiliary lock* (lines 5, 14, 16, 23) to serialize operations on Q. This allows us to implement Q with a min-heap, which has time complexity $O(\log k)$. The ME algorithm that we use for the auxiliary lock is Kim and Anderson's algorithm [13], which has RMR complexity $O(\min(k, \log N))$.

The priority queue has standard operations Insert, Remove, and FindMin, and its entries are pairs of the form (process ID, ticket). Insert is idempotent, and Remove has no effect if attempting to remove an item that is not in the queue. FindMin() returns the process ID whose corresponding ticket is minimal (i.e., the head element), or $\perp$ if the queue is empty. What it means for a ticket to be minimal in a collection of tickets, and more generally how tickets are ordered, is explained in detail in Section 5.

Processes use the Boolean array *Head* to notify another process when it becomes the head of the queue. A process can become the head of the queue after another process removes itself from the queue in the exit protocol (line 18), or after the queue is modified in the waiting room (lines 10–11).

Our algorithm contains additional features, not described above, to handle the following race condition: process p finishes the doorway before q starts the doorway, but then q adds itself to Q before p. By the FCFS property, p should enter the CS before q, but until p is added to Q, q cannot tell (from the state of Q alone) whether it should enter the CS before or after p. To prevent q from entering the CS prematurely, we use special "dummy tickets" and a shared object, *Set*, of a set-like type called *SpecialSet*. At the beginning of the doorway, at line 3, a process q adds itself to *Set*. In the waiting room, at line 7, q removes itself from *Set*, and also learns the ID of one other process $p \neq q$ in *Set*, if it exists ($\perp$ otherwise). If p exists, then p must be in the trying protocol at or before the lock at line 5. In that case, q adds p to Q at line 9 with a "dummy" ticket -1, which is smaller than any "proper" ticket returned by the ticket dispenser at line 4. The insertion of p into Q in this way guarantees that p will be ahead of q in Q until p executes the locked code at lines 6–13, where it replaces its dummy ticket in Q with its proper ticket (lines 10–11). This ensures that q cannot advance into the CS prematurely.

The set operations (line 3 and line 7), and the ticket dispenser operations (line 4 and line 19), are explained in more detail in Sections 4 and 5, respectively.

4 *SpecialSet* – A Set-Like Data Structure

In this section, we describe the data type of the shared object *Set* used in our mutual exclusion algorithm. We refer to this type as *SpecialSet*, because its state is represented by a set but it supports only a few set operations, and only in restricted ways.

The sequential specification of *SpecialSet* is as follows. The state of *SpecialSet* is a set of process IDs. Two operations are used to access *SpecialSet*:

- InsertSelf() adds the ID of the calling process to the set, and returns nothing.

– RemoveSelf() removes the caller's ID from the set, and returns the ID of one other process in the set, if it exists, otherwise returns $\perp$.

Processes must access *SpecialSet* according to the following etiquette:

Condition 1.

(a) The calls to InsertSelf() *and* RemoveSelf() *made by any process occur in an alternating sequence, beginning with* InsertSelf()*, and ending with* RemoveSelf()*; and*

(b) Operation RemoveSelf() *is executed in mutual exclusion.*

For our purposes, it suffices to make the implementation of *SpecialSet* for N processes linearizable and wait-free, with step complexity $O(\min(k, \log N))$, where k denotes point contention. (Note that by Condition 1, if a process has completed InsertSelf() but not yet started its subsequent call to RemoveSelf(), then it is enabled to execute another step, and so we count it in evaluating point contention.)

Below we describe a simple but non-adaptive implementation of *SpecialSet* for N processes, with step complexity $O(\log N)$. Then, we give an informal overview of how the implementation can be made adaptive using existing ideas.

4.1 Non-Adaptive Implementation

The data structure underlying the implementation of *SpecialSet* for N processes is a full binary tree of height $\lceil \log N \rceil$. Each node in the tree stores a process ID or $\perp$, initially $\perp$. We denote the value stored at node n by $NodeVal[n]$. In addition to the tree, we use an array $MyNode[1..N]$ of pointers to tree nodes or $\perp$, initially all $\perp$. For any process ID p, we will refer to $MyNode[p]$ as *p's node*. Informally, if $MyNode[p] = n_p$ for some tree node n_p then p is in the set and p uses tree nodes on the path between n_p and the root node to record information about itself. In the non-adaptive implementation, n_p will be a unique and statically determined leaf node, referred to as *p's leaf node*.

Figure 2: Variables used in *SpecialSet* implementation.

shared variables:
$NodeVal$: **array**$[1..N]$ **of** process ID or $\perp$, **initially** all $\perp$
$MyNode$: **array**$[1..N]$ **of** pointer to tree node or $\perp$, **initially** all $\perp$

Figure 3: InsertSelf() for process $p \in \{1, \ldots, N\}$

```
25 MyNode[p] := ID of p's leaf node
26 InsertHelper(p)
```

Figure 4: InsertHelper(z)

```
27 l := MyNode[z]
28 foreach node n on path from l to root do
29 |  NodeVal[n] := z
30 end
```

Figure 5: RemoveSelf() for process $p \in \{1, \ldots, N\}$

```
   Output: process ID or ⊥
31 l := MyNode[p]
32 foreach node n on the path from l to root do
33 |  if n has a sibling node then
34 |  |  n' := sibling of n
35 |  |  q := NodeVal[n']
36 |  |  if q ≠ ⊥ and MyNode[q] ≠ ⊥ then
37 |  |  |  InsertHelper(q)
38 |  |  |  MyNode[p] := ⊥
39 |  |  |  return q
40 |  |  end
41 |  end
42 end
43 MyNode[p] := ⊥
44 return ⊥
```

The InsertSelf() access procedure for process p first determines p's leaf node at line 25, and then passes control to the helper function InsertHelper(p), which is also used by RemoveSelf(). (Here the ID of p's leaf node is statically determined, but in the adaptive version of the algorithm it is not.) In function InsertHelper(p), the calling process traverses the binary tree from p's node to the root and writes p's ID at each node visited.

The RemoveSelf() access procedure works as follows. The caller, say process p, first determines its tree node, say l, by reading $MyNode[p]$. Next, p traverses the tree from l to the root. For each node visited, p reads the ID stored at the sibling node ($O(\log N)$ nodes in total). For each process ID encountered, say q, p checks whether q's node is not $\perp$. If the latter condition holds, then p stops its traversal immediately after inspecting q's node, and executes InsertHelper(q). (Here $q \neq p$ holds because p's leaf node is statically determined.) By calling InsertHelper(q) at this point, p ensures that if there are any nodes between the current node and the root that contain the ID p, they will be overwritten with an ID that is still in the set. If this were not done, then future calls to RemoveSelf() might behave as though there are no remaining items in the set, and erroneously return $\perp$. Finally, p's execution of RemoveSelf() overwrites $MyNode[p]$ with $\perp$ and returns q. Otherwise, if no such q is found, then upon reaching the root node, p's execution of RemoveSelf() overwrites $MyNode[p]$ with $\perp$ and returns $\perp$.

4.2 Adaptive Implementation

The non-adaptive implementation of *SpecialSet* described above can be altered so that its step complexity becomes adaptive to k – the point contention (as defined earlier for executions involving a *SpecialSet* object). The main idea is to choose p's node so that it has distance $O(\min(k, \log N))$ from the root, which

is difficult since p's node must be unique among all processes that are in the set. One approach is to build the tree dynamically using splitter-like objects, which are based on Lamport's "fast path" mechanism. Anderson and Kim used such objects to construct an adaptive ME algorithm based on reads and writes only [13]. The RMR complexity of this algorithm is $O(\min(k, \log N))$, and key portions of it have step complexity $O(\min(k, \log N))$.

Rather than using pieces of the Anderson and Kim algorithm to create our adaptive implementation of *SpecialSet*, we execute the entire ME algorithm in our implementation and extract certain useful information from that execution. This allows us to re-use complex synchronization machinery directly rather than modifying it and re-proving its correctness properties. The wait-free portion of the trying protocol of the Anderson-Kim algorithm is executed inside InsertSelf(), and the remainder in RemoveSelf(). Since RemoveSelf() is executed in mutual exclusion by Condition 1, this means that the executing process will never busy-wait inside the Anderson-Kim algorithm. (In fact, we can replace the locks used therein with "no-ops".)

5 Ticket Dispenser

Our mutual exclusion algorithm internally uses numbered tickets, much like Lamport's bakery algorithm [2]. Tickets are obtained by calling function ObtainTicket(), which is used in conjunction with function DoneWithTicket() according to the following etiquette:

Condition 2. *The calls to* ObtainTicket() *and* DoneWithTicket() *made by any process occur in an alternating sequence, beginning with* ObtainTicket()*, and ending with* DoneWithTicket()*.*

Informally, we can think of ObtainTicket() as returning a (not necessarily unique) element of some pool of free tickets, and DoneWithTicket() as cleaning up some internal state once a process is done using a particular ticket. (Using a pair of functions in this way makes the ticket dispenser somewhat more complex to specify, but easier to implement.)

We say that a process is *participating* in the ticket dispenser if it has begun its call to ObtainTicket() but not yet completed its subsequent call to DoneWithTicket(). If a participating process has completed its call to ObtainTicket(), then we say that it *holds* the ticket returned by that call. A ticket is *active* if it is held by some process, otherwise it is *inactive*. Tickets satisfy the following properties:

Specification 1.

(a) The domain of tickets is the set of integers modulo mN for some integer $m \geq 3$.
(b) At any time, the set of tickets that are active is confined to some interval of fewer than $mN/2$ consecutive integers modulo mN.

We will use (a) and (b) as follows to define a total order on the set of tickets that are simultaneously active. Given two active tickets i and j, where $i < j$, we will say that i is *less than* j (denoted $i \lhd j$) if $j - i < mN/2$, otherwise we will say that i is *greater than* j (denoted $i \rhd j$). (We will also use $\trianglelefteq$ and $\trianglerighteq$ to denote weak inequalities.) Finally, if $i = j$ then we will say i is *equal to* j. For technical reasons, we also define a special *dummy* ticket, denoted -1, which can be compared against and is less than any active ticket. We say that two tickets are *comparable* if they are simultaneously active (or one or both is -1), and *incomparable* otherwise. Finally, note that our mutual exclusion algorithm compares tickets only implicitly, inside the priority queue.

Having defined an ordering among simultaneously active tickets, we are now ready to specify the correctness properties of ObtainTicket().

Specification 2. *Consider any execution at the end of which distinct processes p and q hold tickets t_p and t_q, respectively. Let C_p and C_q denote the calls to* ObtainTicket() *that generated these tickets, respectively.*

- *If C_p occurred before C_q, then $t_p \lhd t_q$.*
- *If C_p occurred after C_q, then $t_p \rhd t_q$.*
- *If C_p and C_q were concurrent, then the ordering among t_p and t_q is arbitrary*

To simplify the implementation of the operations ObtainTicket() and DoneWithTicket(), we restrict concurrent executions of these functions as follows:

Condition 3.

(a) Function DoneWithTicket() *is executed in mutual exclusion.*
(b) Moreover, if processes p and q are participating simultaneously and hold tickets t_p and t_q, respectively, where $t_p \lhd t_q$, then p subsequently completes a call to DoneWithTicket() *before q does. (In other words, p stops participating before q does.)*

Condition 4. *For any execution fragment during which some process p is (contiguously) participating in the ticket dispenser, every other process participates at most three times during that execution fragment.*

Condition 5. *For any execution fragment during which some process p is (contiguously) executing inside* ObtainTicket(), *if another process q executes* ObtainTicket() *(partially or completely) during that fragment, then q does not subsequently call* DoneWithTicket() *before p finishes its call to* ObtainTicket() *under consideration.*

For our purposes, an implementation of the ticket dispenser must satisfy the following: given that Conditions 2–5 hold, Specifications 1–2 must hold, and the InsertSelf() and RemoveSelf() operations must have step complexity $O(\min(k, \log N))$, where k denotes point contention. (Note that by Condition 2, if a process has completed ObtainTicket() but not yet started its subsequent call to DoneWithTicket(), then it is enabled to execute another step, and so we count it in evaluating point contention.)

5.1 Adaptive Implementation

Next, we describe an implementation of the ticket dispenser that is adaptive in the number of participating processes.

Figure 6: Variables used in ticket dispenser implementation.

```
shared variables:
  Tickets: array[0..7N-1] of {INUSE, FREE}
  initially Tickets[0..(3N-1)] = FREE and Tickets[3N..(7N-1)] = INUSE

  lastTicket: 0..7N-1 initially 7N-1

private variables:
  ticket: 0..7N-1 uninitialized
```

Figure 7: Implementation of ObtainTicket().

```
45 first := lastTicket
46 i := 1
   // Find upper bound on the smallest FREE ticket.
47 while i < 3N ∧ Tickets[(first + i) mod 7N] = INUSE do
48 |  i := min {3N, i × 2}
   // Now do binary search to find the ticket.
49 last := first + i
50 while first < last do
51 |  midpoint := ⌊(first + last)/2⌋
52 |  if Tickets[midpoint mod 7N] = INUSE then
53 |  |  first := midpoint + 1
54 |  else
55 |  |  last := midpoint
   // At this point first = last holds.
56 ticket := first mod 7N
57 Tickets[ticket] := INUSE
58 return ticket
```

Figure 8: Implementation of DoneWithTicket().

```
   // Reset a ticket that was previously active.
59 Tickets[(ticket + 3N) mod 7N] := FREE
60 lastTicket := ticket
```

The algorithm uses a shared circular array *Tickets* of length $7N$, whose entries represent the state of the correspondingly numbered tickets. Each entry is either INUSE or FREE, indicating, as we explain later, whether the corresponding ticket is active. The shared variable *lastTicket* stores the ticket that was held by the last process that stopped participating in the ticket dispenser, i.e., the last process that called DoneWithTicket(), and is used by ObtainTicket() to

efficiently find a FREE ticket. ObtainTicket() uses a two-stage search mechanism to determine the next FREE ticket. First, the algorithm attempts to find an interval of consecutive tickets, starting at *lastTicket*, that contains a FREE ticket. This is done at lines 45–49 by searching rightwards from *lastTicket* in steps of exponentially increasing size, up to a distance of $3N$. Starting at *lastTicket* ensures that the search is adaptive to point contention, k, and taking steps of exponentially increasing size bounds the total number of steps taken to be $O(\log k)$. We only need to search up to a distance of $3N$ from *lastTicket*, since, by Condition 4, every other process participates at most three times while the search is being done. This means there will be at most $3(N-1)$ INUSE tickets after *lastTicket*.

Once a FREE ticket is found, the interval from *lastTicket* to the FREE ticket is guaranteed to contain at least one FREE ticket. However, there may be another FREE ticket earlier in the interval. The algorithm performs a binary search of the interval at lines 50–55 to pinpoint such a ticket if it exists. The ticket computed is stored in the private variable *ticket* at line 56, and marked INUSE at line 57.

Function DoneWithTicket() simply resets a previously-active ticket at line 59 (so that it can be reused later), and then updates *lastTicket* at line 60.

6 Correctness of FCFS ME Algorithm

In this section we provide a very high-level overview of why the FCFS ME Algorithm defined in Figure 1 is correct, and why it has RMR complexity $O(\min(k, \log N))$.

The correctness of the FCFS ME algorithm relies on the correctness of the *SpecialSet* and ticket dispenser implementations outlined in Sections 4 and 5. These implementations are correct only if they are used according to the etiquette outlined in Conditions 1–5. Our proof that these conditions hold in Figure 1 relies on the ME algorithm satisfying FCFS. Our proof for FCFS, however, relies on the correctness of the *SpecialSet* and ticket dispenser, which leads to a cycle of dependencies. We deal with this cycle through careful induction on the length of the *execution history*. (An execution history is an alternating sequence of *states* and process steps, where a state consists of the values assigned to all private and shared variables in the system, and a step is a shared memory operation by a process.)

The proof proceeds in two parts. The first part shows that FCFS holds in any execution history in which the *SpecialSet* and ticket dispenser are correct. The second part uses induction to show that Conditions 1–5 hold in any execution history, and hence that the *SpecialSet* and ticket dispenser are correct. We proceed in reverse, sketching the second part of the proof first, and then sketching the remaining details.

Lemma 1. *Conditions 1–5 hold in any execution history of the algorithm.*

Proof sketch. By inspection of the ME algorithm in Figure 1, Conditions 1–3(a) hold. To show that Conditions 3(b)–5 hold, we use induction on the length of the execution history H. In the initial state of H, no process has taken a step,

and so the conditions hold trivially. We assume that the conditions hold up to some state s in the execution history, and show that the conditions also hold in the next state s' after s. Suppose, by way of contradiction, that the conditions do not hold in state s'. Since all conditions are satisfied up to s, it can be shown that exactly one condition is not satisfied in s'. Due to space limitations, we only argue for a contradiction when Condition 4 does not hold. In this case, there must be some process p contiguously participating in the ticket dispenser while another process q participates four times. Process q must have started participating for the fourth time when it took a step between s and s'. It turns out (by Condition 5) that p must have finished executing ObtainTicket() before q went through the CS when it participated in the ticket dispenser the second time. Thus, during q's third time participating, p will have finished the doorway before q starts it. FCFS holds prior to s', and so q cannot execute through the CS and participate a fourth time until p has executed through the CS. But this means that when q participates for the fourth time, p will no longer be participating contiguously, contradicting the assumption that it is.

Lemma 2. *The algorithm satisfies mutual exclusion.*

Proof. The lemma follows from the correct use of the auxiliary lock, which surrounds (among other things) the CS.

Lemma 3. *The algorithm satisfies bounded exit.*

Proof. The Kim and Anderson [13] algorithm, which we use for the auxiliary lock, satisfies bounded exit. Consequently, it follows from the structure of our algorithm that it too satisfies bounded exit.

Lemma 4. *The algorithm satisfies FCFS.*

Proof sketch. Assume that some process p finishes the doorway before some process q starts the doorway, and suppose, by way of contradiction, that q enters the CS before p in the corresponding passages. Immediately before q does so, p and q hold their tickets simultaneously. Since p finished the doorway before q started it, p's call to ObtainTicket() finished executing before q's call to ObtainTicket() started. This and the ticket dispenser specification imply that p's ticket is smaller than q's. If p adds itself to Q at line 11 before q, then q has no hope of entering the CS before p since p will be in front of q in Q. So it must be the case that q adds itself to Q before p by executing the locked segment of code at lines 6–13 before p. In this case, however, q's call to RemoveSelf() at line 7 returns $tmp\_id \neq \perp$ (possibly $tmp\_id = p$), since Set contains p. This means that at line 9, q adds some process to Q with a dummy ticket. Process q cannot be signalled to enter the CS while there is a dummy ticket in Q, and it turns out the latter condition holds at least until p adds itself with its proper ticket to Q. When p does add itself to Q, it will be in front of q, since p has a smaller ticket than q. This implies that p will enter the CS before q, which contradicts the assumption that q enters before p.

Lemma 5. *The algorithm satisfies deadlock freedom.*

Proof sketch. Suppose, by way of contradiction, that deadlock freedom does not hold. That is, some process p loops forever in the trying protocol, and after some point in the execution, no process enters the CS. It turns out that the only place where p may be looping is at line 15, while waiting to be signalled to enter the CS. Furthermore, since there is a point after which no process enters the CS, there must be a last call to Q.FindMin() (line 12 or 20). The contradiction that we derive is to show that after the last call to Q.FindMin(), there must be another call to Q.FindMin().

When the last call to Q.FindMin() occurs, it cannot return $\perp$. If it did return $\perp$, this would mean Q is empty. But then p's final execution of the locked segment of code at lines 6–13 must occur after the last call to Q.FindMin(), otherwise p would already be in the queue and at (or about to execute) line 15 when the latter call occurs. This implies that p executes Q.FindMin() after the last call to Q.FindMin().

It also follows that when the last call to Q.FindMin() occurs, it returns the ID of a process q that is not associated with a dummy ticket. If q were associated with a dummy ticket, then q must be in the trying protocol before the locked segment of code. This means that q eventually executes Q.FindMin() after the last call to Q.FindMin().

Thus, one of the two following cases must hold: (i) the last call to Q.FindMin() is at line 12 and returns the ID of the caller, a process q; or (ii) the last call to Q.FindMin() is at line 20 and returns the ID of a process q that is at lines 14–16 at the time. In both cases, q will eventually be signalled to enter the CS, and so q will eventually call Q.FindMin() at line 20, after the last call to Q.FindMin().

Lemma 6. *The algorithm satisfies lockout freedom.*

Proof. Lockout freedom follows directly from FCFS (Lemma 4) and deadlock freedom (Lemma 5).

Lemma 7. *The algorithm has RMR complexity* $O(\min(k, \log N))$ *in both the DSM and CC models.*

Proof sketch. The ticket dispenser operations and *SpecialSet* operations have step complexity $O(\min(k, \log N))$. For the auxiliary lock at lines 5, 14, 16, 23, we use the adaptive mutual exclusion algorithm of Kim and Anderson [13], which has RMR complexity $O(\min(k, \log N))$. For the priority queue, we use a min-heap implementation, which has step complexity $O(\log k)$. The busy-wait loop at line 15 incurs $O(1)$ RMRs in the CC model, and no RMRs in the DSM model. Every other line of the algorithm causes at most $O(1)$ RMRs per passage.

The preceding lemmas culminate in the following theorem:

Theorem 1. *The algorithm defined by Figure 1 is a correct FCFS mutual exclusion algorithm, and it has RMR complexity* $O(\min(k, \log N))$ *in both the DSM and CC models, where* k *is the point contention and* N *is the number of processes in the system.*

Acknowledgements: We are grateful to Vassos Hadzilacos for his insightful feedback during the writing of this paper. We also thank the anonymous referees for their helpful comments.

References

1. Dijkstra, E.: Solution of a problem in concurrent programming control. Communications of the ACM **8**(9) (September 1965) 569
2. Lamport, L.: A new solution to Dijkstra's concurrent programming problem. Communications of the ACM **17**(8) (August 1974) 453–455
3. Mellor-Crummey, J., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9**(1) (February 1991) 21–65
4. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems **1**(1) (January 1990) 6–16
5. Kim, Y.J., Anderson, J.: A time complexity bound for adaptive mutual exclusion. In: Proc. DISC '01. (2008) 1–15
6. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: Proc. STOC'08. (2008) 217–226
7. Golab, W., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-RMR implementations of cas and other synchronization primitives using read and write operations. In: Proc. PODC '07, New York, NY, USA, ACM (2007) 3–12
8. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. **5**(1) (1987) 1–11
9. Styer, E.: Improving fast mutual exclusion. In: PODC '92: Proceedings of the eleventh annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1992) 159–168
10. Choy, M., Singh, A.K.: Adaptive solutions to the mutual exclusion problem. Distrib. Comput. **8**(1) (1994) 1–17
11. Attiya, H., Bortnikov, V.: Adaptive and efficient mutual exclusion. Distrib. Comput. **15**(3) (2002) 177–189
12. Yang, J.H., Anderson, J.H.: A fast, scalable mutual exclusion algorithm. Distributed Computing **9**(1) (1995) 51–60
13. Kim, Y.J., Anderson, J.: Adaptive mutual exclusion with local spinning. Dist. Computing **19**(3) (2007) 197–236
14. Afek, Y., Stupp, G., Touitou, D.: Long lived adaptive splitter and applications. Distrib. Comput. **15**(2) (2002) 67–86
15. Cypher, R.: The communication requirements of mutual exclusion. In: SPAA '95: Proc. of the 7th annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM Press (1995) 147–156
16. Anderson, J., Kim, Y.J.: An improved lower bound for the time complexity of mutual exclusion. Distributed Computing **15**(4) (December 2002) 221–253
17. Fan, R., Lynch, N.: An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In: PODC '06: Proc. of the 25th annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM Press (2006) 275–284
18. Jayanti, P.: f-arrays: Implementation and applications. In: PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing, New York, NY, USA, ACM (2002) 270–279
19. Taubenfeld, G.: The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. In: DISC. (2004) 56–70