The *k*-Bakery: Local-spin *k*-Exclusion Using Non-atomic Reads and Writes

Robert Danek*
Department of Computer Science
University of Toronto
Toronto, Canada
rdanek@cs.toronto.edu

ABSTRACT

Mutual exclusion is used to coordinate access to shared resources by concurrent processes. k-Exclusion is a variant of mutual exclusion in which up to k processes can simultaneously access the shared resource. We present the first known shared-memory k-exclusion algorithms that use only atomic reads and writes, have bounded remote memory reference (RMR) complexity, and tolerate crash failures. Our algorithms have RMR complexity $O(N)$ in both the cache-coherent and distributed shared-memory models, where N is the number of processes in the system. Additionally, we present a k-exclusion algorithm that satisfies the First-In-First-Enabled (FIFE) fairness property. FIFE requires that processes become "enabled" to enter the CS roughly in the order that they request access to the shared resource. Finally, we present a modification to the FIFE k-exclusion algorithm that works with non-atomic reads and writes. The high-level structure of all our algorithms is inspired by Lamport's famous Bakery algorithm.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Algorithms, Performance, Theory

Keywords

Mutual exclusion, k-exclusion, safe registers, remote memory references, shared memory

*Supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

PODC'10, July 25–28, 2010, Zurich, Switzerland.

1. INTRODUCTION

Mutual exclusion [9] is used to resolve conflicts between multiple processes trying to access a shared resource in a multiprocessor system. A mutual exclusion algorithm consists of two methods, the *trying* and *exit protocols*, that protect the resource (also called the *critical section* (CS)) so that at most one process is allowed to use the resource at a given time. A process executes the trying protocol whenever it wants to access the CS, and does not enter the CS until the trying protocol terminates. After a process leaves the CS, it executes the exit protocol. If a process is not in the trying protocol, exit protocol, or CS, it is said to be in the non-critical section (NCS).

To coordinate access to the CS, processes communicate with each other by reading and writing variables in shared memory. In particular, the algorithms that we consider in this paper use *atomic* and *non-atomic* reads and writes. Each time a process reads or writes a single shared variable, we say that the process has taken a *step*. Intuitively, an atomic read or write takes effect "instantaneously" in one indivisible step, whereas non-atomic reads and writes can interfere with each other in the following way: if a non-atomic read and write access the same variable concurrently, then the read returns an arbitrary value. We do not worry about the case in which multiple processes concurrently write the same variable, as this situation does not arise in our algorithms. More precisely, atomic (non-atomic) reads and writes of variables in this paper behave the same as reads and writes of multireader single-writer atomic (safe) registers as defined by Lamport [19].

k-Exclusion [12] is a generalization of mutual exclusion that allows up to k processes to be in the critical section concurrently, and can tolerate up to $k-1$ process *crashes* in a manner described more precisely in the starvation freedom property below. We say that a process crashes if it stops taking steps while outside the NCS. If a process crashes we say that it is *faulty*; otherwise we say that it is *non-faulty*. Note that, for our purposes, a process that stops taking steps in the NCS, i.e., a process that never attempts to enter the CS after some point, is not considered faulty.

The correctness properties that a k-exclusion algorithm must satisfy are summarized as follows:

k-Exclusion: At most k processes are in the critical section (CS) at the same time.

Starvation Freedom: If a non-faulty process p is in the trying protocol and at most $k-1$ other processes crash, then p eventually enters the CS.

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

We assume that the CS is finite to prevent situations in which processes can take an infinite number of steps inside the CS. That is, a process that enters the CS and does not crash will eventually enter the exit protocol.

In ordinary mutual exclusion (i.e., $k = 1$), it is sometimes desirable for processes to execute through the CS roughly in the order in which they leave the NCS. This order is more "fair" than if processes were allowed to execute through the CS in an arbitrary order, as it gives priority to processes that try accessing the CS earlier than others. To make this idea of fair ordering more precise, we split the trying protocol into two parts: the first part is a bounded piece of code called the *doorway*, and the second part is known as the *waiting room*. Fair ordering is captured by the *First-Come-First-Served (FCFS)* property [17]:

FCFS: If a process p finishes the doorway before a process q starts the doorway, then q does not enter the CS before p enters the CS.

In the context of k-exclusion, when $k > 1$, requiring that an algorithm satisfies FCFS does not make sense, since it conflicts with the starvation freedom property: If a process p completes the doorway and crashes before a non-faulty process q enters the doorway, then a k-exclusion algorithm that satisfies FCFS must prevent q from ever entering the CS, thereby violating starvation freedom. However, we can weaken the FCFS property so that it does not conflict with starvation freedom and still provides some modicum of fairness. We define a new property in this paper called *k-FCFS*:

k-FCFS: For any set of processes Y such that $|Y| = k$, if all processes in Y finish the doorway before a process p starts the doorway, then p does not enter the CS before at least one process in Y enters the CS.

A nice feature of k-FCFS is that it makes sense for all $k \geq 1$. In particular, for $k = 1$, it is simply the FCFS property for ordinary mutual exclusion.

Another fairness property for the k-exclusion problem, known as *First-In-First-Enabled (FIFE)*, was originally defined by Fischer et al. [12]. Intuitively, FIFE requires that processes become *enabled* to enter the CS in the order in which they execute through the doorway. A process p is enabled to enter the CS if it can enter the CS in a bounded number of its own steps. More precisely:

FIFE: If a process p finishes the doorway before a process q starts the doorway, and q enters the CS before p, then p can enter the CS in a bounded number of its own steps.

FIFE is a stronger property than k-FCFS in that any k-exclusion algorithm that satisfies FIFE also satisfies k-FCFS. To see why this is case, suppose, by way of contradiction, that there is a k-exclusion algorithm that satisfies FIFE but not k-FCFS. There exists an execution of this algorithm in which some set of processes Y, where $|Y| = k$, all finish the doorway before a process p starts the doorway, and p enters the CS before any process in Y enters the CS. By FIFE, each process in Y is able to enter the CS in a bounded number of its own steps. Thus, the execution can proceed in such a way that process p remains in the CS until all processes in Y enter the CS. We now have a situation in which $k + 1$ processes are in the CS, contradicting that the algorithm satisfies k-exclusion.

The time complexity of an algorithm is typically measured by counting the number of reads or writes that can occur during its execution. This measure is useless for k-exclusion (and ordinary mutual exclusion) algorithms, however, for the following reason. When a process executes the trying protocol of a k-exclusion algorithm, it may be forced to wait to enter the CS so as not to violate the k-exclusion property. During this waiting period, a process can read or write a single variable an unbounded number of times. Even if no processes are already in the CS, Alur and Taubenfeld [2] proved that for any mutual exclusion algorithm that uses only reads and writes and involves two or more processes, even the first process into the CS may need to make an unbounded number of accesses to shared variables. This leads to an unbounded time complexity measure.

A more practical measure of time complexity that avoids the preceding problem is obtained by counting the number of *remote* memory references (RMRs) that a process makes in a *passage*. A passage is the time between when a process leaves the NCS to when it next returns to it, and an RMR occurs whenever a process accesses a variable that requires it to traverse the processor-to-memory interconnect, which can be a bottleneck. As a result of having to traverse the interconnect, RMRs tend to be an order of magnitude slower than local memory references, and the performance of many shared-memory algorithms degrades as the number of RMRs they make increases [20]. This observation suggests that to maximize the performance of such algorithms, it is important to minimize the number of RMRs. The worst-case number of RMRs that a process can make in a passage is referred to as the algorithm's *RMR complexity*, and an algorithm having bounded RMR complexity is referred to as a *local-spin* algorithm.

To make the idea of a remote versus local memory reference more precise, we consider two models for shared memory architectures: the Distributed Shared Memory (DSM) model, and Cache-Coherent (CC) model [3]. In the DSM model, each process has a memory module that it can access locally and all other processes can access remotely. Each shared variable is assigned to a processor's memory module prior to the start of execution, and remains there for the duration of the execution.

There are a number of different flavours of CC models. We describe the write-through/write-invalidate model here, although our results apply equally to the write-back/write-invalidate model, which is similar and is also commonly used in practice. In the write-through/write-invalidate CC model, each process has a local cache, and there is a global memory store that all processes can access remotely. To read a variable, a process first tries reading it locally in its cache. If the variable is not there, the process then accesses the variable remotely in the global store and caches it locally. A cached copy of the variable will remain in a process's cache until the copy is invalidated. To write a variable, a process writes it remotely to the global memory store and invalidates any copies cached by other processes.

Our main contribution in this paper is the first known

Reference	RMR Complexity	Instructions Used	Starvation Free
Fischer et al. [12]	∞	Atomic Actions[1]	Yes
Fischer et al. [13]	∞	Atomic Actions	Yes
Dolev et al. [10]	∞	Non-Atomic Read and Write	Yes
Afek et al. [1]	∞	Atomic Read and Write	Yes
Peterson (CC) [21]	$\Theta(N^3 - Nk^2)$	Atomic Read and Write	No
Peterson (DSM) [21]	∞	Atomic Read and Write	No
Burns and Peterson [6]	∞	Atomic Read and Write	Yes
Gottlieb et al. [14]	∞	Fetch&Add	No
Anderson and Moir (CC/DSM) [4]	$\Theta(k \log (N/k))$	Fetch&Add, Test&Set, Compare&Swap	Yes
Anderson and Moir (CC/DSM) [4]	$\Theta(c)$	Fetch&Add, Test&Set, Compare&Swap	Yes
Danek and Lee (CC/DSM) [8]	$\Theta(k \log N)$	Atomic Read and Write	No
CC/DSM algorithm (Figure 2 - unshaded portion)	$\Theta(N)$	Atomic Read and Write	Yes
FIFE algorithm (Figure 2)	$\Theta(N)$	Atomic Read and Write	Yes
Non-Atomic FIFE algorithm (Figure 3)	$\Theta(N)$	Non-Atomic Read and Write	Yes

Table 1: Known k-exclusion algorithms.

local-spin k-exclusion algorithms that use only atomic reads and writes. Our algorithms have $O(N)$ RMR complexity in both the CC and DSM models, where N is the number of processes in the system, and satisfy the k-FCFS property. One variant of the algorithms additionally satisfies the FIFE property. Also, we present a version of the FIFE k-exclusion algorithm that works with non-atomic reads and writes. The high-level structure of all our algorithms is inspired by Lamport's famous Bakery algorithm.

The remainder of the paper is organized as follows. In the next section we present a comparison of our results to known k-exclusion algorithms and discuss the importance of our results in the context of what is currently known about mutual exclusion and k-exclusion. We then provide a brief overview of Lamport's Bakery algorithm, after which we present our results. The first algorithm is a k-exclusion algorithm satisfying k-FCFS. We then modify the algorithm so that it satisfies FIFE, and finally present a version of the algorithm that uses only non-atomic reads and writes. We provide a full proof of correctness for the first algorithm, but omit the proofs for the other algorithms due to space limitations.

2. KNOWN RESULTS AND OPEN PROBLEMS

In Table 1 we present a summary of known k-exclusion algorithms and their properties[2]. Several algorithms [21, 14, 8] do not satisfy starvation freedom as defined above. Rather, they satisfy weaker progress properties that depend on processes not crashing. Excluding these exceptions, our discussion of k-exclusion algorithms in this paper is restricted to algorithms satisfying all of k-exclusion, starvation freedom, and bounded exit.

The algorithms in this paper have a very similar structure to the k-exclusion algorithm by Afek et al. [1], which is also inspired by Lamport's Bakery algorithm. The main differences between the present work and the algorithm in [1] are that our algorithms are local-spin, and we also present a k-exclusion algorithm that works with non-atomic reads and writes.

The only previously known local-spin k-exclusion algorithms satisfying starvation freedom are by Anderson and Moir [4]. Their algorithms have RMR complexity $\Theta(k \log (N/k))$ and $\Theta(c)$, where c is *point contention*, i.e., the maximum number of processes simultaneously outside of the NCS during a passage. Unlike the present work, Anderson and Moir's algorithms use strong synchronization primitives such as Fetch&Add, Test&Set, and Compare&Swap in addition to atomic reads and writes. Interestingly, the worst-case RMR complexity of their algorithms matches asymptotically the worst-case RMR complexity of the algorithms in this paper, for any k that is a constant fraction of N (e.g., for $k = N/2$). This is significant because it leads to a state of affairs in which the only known local-spin k-exclusion algorithms that use strong synchronization primitives have the same worst-case RMR complexity (within constant factors) as the only known local-spin k-exclusion algorithms that use only atomic reads and writes. This is in contrast to what is known about the case in which $k = 1$ (i.e., ordinary mutual exclusion): Using stronger synchronization primitives like Fetch&Add there exist $O(1)$ RMR complexity mutual exclusion algorithms [3], whereas the class of mutual exclusion algorithms that use only atomic reads and writes (and comparison primitives) have $\Omega(\log N)$ RMR complexity [5].

This paper solves the open problem of whether there exists local-spin k-exclusion algorithms that use only reads and writes. In light of the preceding paragraph, our results also open some interesting questions. For a given set of synchronization primitives, what exactly is the RMR complexity of k-exclusion as k varies? Is the RMR complexity of k-exclusion (asymptotically) the same as that of mutual exclusion for all values of k, or are there values of k for which k-exclusion is (asymptotically) harder in terms of RMRs than

[1]Atomic actions refer to parts of the algorithm that are assumed to execute atomically. Atomic actions are more complex than Fetch&Add or Compare&Swap operations, and consist of several reads, writes, and comparisons executed together.

[2]This table is adapted from one by Anderson and Moir [4].

```
shared variables:
    Doorway: array[1..N] of boolean init all false
    Ticket: array[1..N] of ℕ init all 0

1  loop
2  |  NCS
3  |  Doorway[p] := true
4  |  Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
5  |  Doorway[p] := false
6  |  for i := 1 to N do
7  |  |  await Doorway[i] = false
8  |  |  await Ticket[i] = 0 ∨ (Ticket[i], i) ≥ (Ticket[p], p)
9  |  CS
10 |  Ticket[p] := 0
11 end loop
```

Figure 1: Lamport's Bakery algorithm for ordinary mutual exclusion, for process $p \in \{1, ..., N\}$

mutual exclusion? This question can be asked with respect to algorithms that use only reads and writes, as well as algorithms that also use stronger synchronization primitives such as FETCH&ADD. In the case of mutual exclusion it is known that the choice of primitives affects the RMR complexity. As pointed out above, however, this not (yet) known in the case of k-exclusion.

3. LAMPORT'S BAKERY ALGORITHM

Lamport's Bakery algorithm [17] is a non-local-spin FCFS mutual exclusion algorithm that is correct even when reads and writes are non-atomic. It is given in Figure 1. The name of the algorithm arises from the fact that processes behave like people waiting in line at a bakery. Each process obtains a ticket and then waits for its turn to be served by the CS.

The algorithm uses two shared arrays: *Doorway*, and *Ticket*. A process p sets $Doorway[p]$ be true at line 3 to indicate to other processes that it has started the doorway, and then sets it to false at line 5 when it finishes the doorway. At line 4 of the doorway, process p obtains a ticket used to indicate its order of priority to enter the CS. In the waiting room, process p waits for each other process q to finish the doorway (line 7), and then waits until it has priority over q to enter the CS (line 8). Process p has priority over q to enter the CS if either q is not requesting entry into the CS (indicated by $Ticket[q] = 0$) or q's ticket is larger than p's ticket (using process id's to break ties between equal tickets).

4. LOCAL-SPIN k-EXCLUSION

In Figure 2 we present a k-exclusion algorithm that has $O(N)$ RMR complexity in the CC and DSM models. We ignore in this section the shaded parts of the algorithm, which are necessary to ensure FIFE holds. We explain the shaded parts in the next section.

The high-level structure of the algorithm is similar to Lamport's bakery algorithm: processes choose a ticket in the first part of the trying protocol, and then wait for processes with lower-numbered tickets to execute through the CS. There are two main differences between our algorithm and Lamport's algorithm. Firstly, processes announce their ticket at the start of the trying protocol differently. Secondly, in our algorithm a process does not have to wait for every process with a lower-numbered ticket to finish the CS, but rather only waits until there are fewer than k processes with lower-numbered tickets (line 19). We now explain the algorithm in more detail.

Processes use two shared arrays to communicate with each other: the *Want* array, and the *Ticket* array. The *Want* array is used by processes to announce their wish to enter the CS. In particular, the value stored in entry $Want[p][q]$ is a "ticket value", and it indicates to process q whether p is trying to enter the CS. When a process p is in the NCS, $Want[p][q] = \infty$, indicating that p does not want to enter the CS. When $Want[p][q] = v$ for some value $v \neq \infty$, then process p is outside of the NCS and is trying to enter the CS with the ticket value v. Ticket values provide a rough guideline for the order in which processes are admitted to the CS. Ticket values are stored and generated using the shared array *Ticket*. We now explain how these shared arrays are employed in the trying and exit protocols of the algorithm.

The trying protocol (lines 14..21) consists of two parts: the doorway, and the waiting room. The doorway consists of lines 14..15. In this part of the trying protocol, a process first announces itself at line 14 to all other processes with the ticket it obtained in its previous passage (0 if it executed no previous passage). A process then obtains a new ticket at line 15. Once a process is done the doorway, it announces the ticket it just obtained to all other processes at line 16.

One may wonder why it is not sufficient to simply start the trying protocol by obtaining a new ticket (line 15) and then announce it. This is due to a race condition that arises otherwise, but we postpone discussion of this until after we explain the next part of the algorithm.

After announcing its ticket, a process p initializes *predecessor_set* to be the set of all other processes in the system (line 18). The predecessor set is intended to approximate the set of processes that are trying to enter the CS concurrently with p and that have priority over p to enter the CS. We say that a process q has priority over p if q and p choose tickets t_q and t_p and $(t_q, q) < (t_p, p)$.

The purpose of lines 19..21 is to prevent a process p from entering the CS until enough processes are eliminated from p's predecessor set. To this end, p repeatedly checks the state of all other processes still in its predecessor set in the

loop at line 20. Once p detects that a process q no longer has priority over it, p removes q from its predecessor set at line 21.

When the size of p's predecessor set goes below k, p enters the CS.

In the exit protocol (line 25), a process simply announces to all other processes that it is returning to the NCS by setting the appropriate values in the *Want* array to ∞.

We now explain the race condition that arises if line 14 is removed from the algorithm. If two processes p and q execute line 15 concurrently, the following may occur: q obtains a smaller ticket than p, but does not yet announce the ticket at line 16. Process p then races ahead of q into the waiting room. As q has not yet announced its ticket to p, p will see that $Want[q][p] = \infty$, and so p will remove q from its predecessor set. This is premature on p's part, since q actually has priority over p. When q executes the waiting room, it also removes p from its predecessor set, since p has a larger ticket than q. It is easy to see how this can lead to a violation of k-exclusion for $k = 1$. More elaborate execution scenarios can be constructed in which k-exclusion is violated for $k > 1$.

The preceding scenario is avoided if q first announces the ticket it obtained in its previous passage. Since q, in its current passage, obtains a ticket smaller than p, the ticket q announces at line 14 is also guaranteed to be smaller than p's ticket. Moreover, by the time p starts the waiting room, q will have finished line 14, otherwise q will obtain a ticket larger than p. Thus, when p executes the waiting room, it will not remove q from its predecessor set prematurely.

4.1 Notation

Below we prove that the k-exclusion algorithm in Figure 2 is correct. Our proof of correctness uses notation similar to that used by Lamport [18, 19] for describing the order in which processes execute operations. In particular, if A and B are operations (not necessarily atomic), the notation $A \rightarrow B$ denotes that A ends before B starts, and $A \dashrightarrow B$ denotes that B does not end before A starts.

In our proofs we make use of two simple facts about the $\rightarrow$ and $\dashrightarrow$ relations that follow from the preceding definitions: (a) $A \dashrightarrow B \Leftrightarrow B \not\rightarrow A$, and (b) $A \rightarrow B \dashrightarrow C \rightarrow D \Rightarrow A \rightarrow D$.

Processes can execute multiple passages of the algorithm, and so to be strictly accurate in our proofs, whenever we refer to a process executing an operation, we should also specify the passage being executed. However, to avoid unnecessary complexity in our notation, if we do not have to reason across multiple passages, we omit specifying the passage and assume that the passage being executed by each process is the "current" (i.e., latest) one with respect to some point in the execution.

4.2 Proof of Correctness

Let C_p denote the operation in which process p chooses its ticket at line 15, and let EP_p denote the operation in which p executes its exit protocol at line 25.

LEMMA 1. *Let p and q be distinct processes. If $C_p \rightarrow C_q$, then $t_p < t_q$, where t_p and t_q are the tickets chosen by p and q, respectively.*

PROOF. Process p finishes line 15 before q starts it, and so the value that q reads from $Ticket[p]$ at line 15 is no smaller than t_p. This and inspection of line 15 imply that the ticket t_q that q chooses satisfies $t_q > t_p$. □

As a corollary to the preceding lemma, we state its contrapositive:

COROLLARY 2. *Let p and q be distinct processes, and suppose p and q choose tickets t_p and t_q. If $t_q \leq t_p$, then $C_q \dashrightarrow C_p$.*

LEMMA 3. *Let p and q be distinct processes, and suppose p and q obtain tickets t_p and t_q, respectively. If $(t_q, q) < (t_p, p)$, then p does not remove q from $p.predecessor\_set$ before EP_q starts.*

PROOF. Suppose, by way of contradiction, that p removes q from $p.predecessor\_set$ at line 21 before EP_q starts. At line 21, p reads a value w_q in $Want[q][p]$ such that $(t_p, p) < (w_q, q)$ (denote this read operation R) before EP_q starts. Process q writes into $Want[q][p]$ at most three times in its passage: at line 14 it writes the ticket $t'_q < t_q$ that it chose in its preceding passage (denote this write operation W_1); at line 16 it writes t_q (denote this write operation W_2); and finally, at line 25, it writes ∞ (denote this write operation W_3).

CLAIM 3.1. $R \rightarrow W_3$.

PROOF. The claim follows from the following two facts: (i) R happens before EP_q starts, and (ii) W_3 occurs entirely as part of EP_q. (Claim 3.1) □

CLAIM 3.2. $W_1 \rightarrow R$.

PROOF. By the assumption that $(t_q, q) < (t_p, p)$, $t_q \leq t_p$. This and Corollary 2 imply that $C_q \dashrightarrow C_p$. By inspection of the algorithm, $W_1 \rightarrow C_q$ and $C_p \rightarrow R$. Thus, $W_1 \rightarrow C_q \dashrightarrow C_p \rightarrow R$, and so $W_1 \rightarrow R$. (Claim 3.2) □

Read and write operations are atomic, which implies that either $R \rightarrow W_2$, or $W_2 \rightarrow R$. We consider these cases separately. In the first case, by Claim 3.2, $W_1 \rightarrow R \rightarrow W_2$. W_1 and W_2 are successive writes into $Want[q][p]$, and so R must read the value written by W_1, namely the ticket $t'_q < t_q$ that q chose in its preceding passage. By assumption, $(t_q, q) < (t_p, p)$, which implies that $t_q \leq t_p$; since $t'_q < t_q$, we have $t'_q < t_p$. That is, R must read a value t'_q such that $t'_q < t_p$. This contradicts that R reads a value w_q such that $(t_p, p) < (w_q, q)$, i.e., $w_q \geq t_p$.

In the second case, $W_2 \rightarrow R$. This and Claim 3.1 imply that $W_2 \rightarrow R \rightarrow W_3$. W_2 and W_3 are successive writes into $Want[q][p]$ and so the value w_q that R reads must be the value t_q written by W_2, i.e., $w_q = t_q$. By assumption, $(t_q, q) < (t_p, p)$, which contradicts that $(t_p, p) < (w_q, q)$. □

LEMMA 4. *The algorithm in Figure 2 satisfies k-exclusion.*

PROOF. Suppose, by way of contradiction, that $k+1$ processes are in the CS concurrently. Let Y be this set of processes, and let p be the process in Y with the largest $(ticket, process\_id)$ pair. By Lemma 3, when p executes the trying protocol, p does not remove any process in Y from $p.predecessor\_set$ before some process in Y starts the exit protocol. This implies that the size of p's predecessor set is at least k until after some process in Y leaves the CS. Thus p cannot enter the CS until after some process in Y leaves the CS, which contradicts that p is in the CS concurrently with all processes in Y. □

(Shaded parts of this algorithm are necessary to satisfy FIFE.)

shared variables:

Want: **array**$[1..N][1..N]$ **of** $\mathbb{N} \cup \{\infty\}$ **init all** ∞
Ticket: **array**$[1..N]$ **of** $\mathbb{N}$ **init all 0**
Capture: **array**$[1..N][1..N]$ **of** $\mathbb{N}$ **init all 0**

(DSM model: $Ticket[p]$, $Want[i][p]$, $Capture[i][p]$ are local to process p for all i)

private variables:

predecessor_set: **Set of** $\mathbb{N}$
captured: **boolean**

```
12 loop
13   NCS
14   foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
15   Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
16   foreach i ∈ {1..N} \ {p} do  Want[p][i] := Ticket[p]
17   captured := false
18   predecessor_set := {1..N} \ {p}
19   while |predecessor_set| ≥ k ∧¬captured do
20     foreach i ∈ predecessor_set do
21       if (Ticket[p], p) < (Want[i][p], i) then  predecessor_set := predecessor_set \ {i}
22     foreach i ∈ {1..N} do  if Ticket[p] < Capture[i][p] then  captured := true
23   foreach i ∈ {1..N} do Capture[p][i] := Ticket[p]
24   CS
25   foreach i ∈ {1..N} do  Want[p][i] := ∞
26 end loop
```

Figure 2: (Atomic Reads/Writes) k-exclusion algorithms for process $p \in \{1, ..., N\}$

LEMMA 5. *The algorithm in Figure 2 satisfies starvation freedom.*

PROOF. Suppose, by way of contradiction, that starvation freedom does not hold. Let Y be the set of non-faulty processes that are stuck forever in the TP, and let p be the process in Y with the smallest $(ticket, process\_id)$ pair.

By inspection of the code, we see that p removes from its predecessor set any process that it sees as having announced a larger $(ticket, process\_id)$ pair or as having returned to the NCS. This, the fact that process' tickets increase monotonically, and the assumption that p has the smallest $(ticket, process\_id)$ pair of all the non-faulty processes that get stuck forever, imply that eventually the only processes in p's predecessor set are crashed processes. However, there are at most $k-1$ crashed processes, and so p eventually enters the CS, contradicting that p never enters the CS. □

LEMMA 6. *The algorithm in Figure 2 satisfies k-FCFS.*

PROOF. Let Y be a set of processes such that $|Y| = k$, and assume all processes in Y finish the doorway before a process p starts the doorway. Thus $\forall q \in Y$, $C_q \rightarrow C_p$. This and Lemma 1 (with the roles of p and q interchanged) imply that each process in Y obtains a ticket strictly smaller than the ticket that p obtains. By Lemma 3, during p's execution of the trying protocol, p does not remove any process in Y from $p.predecessor\_set$ until at least one process in Y completes the CS. This implies that the size of $p.predecessor\_set$ will be at least k until some process in Y executes through the CS. Process p does not enter the CS until the size of its predecessor set is less than k, and so p does not enter the CS until some process in Y enters the CS. □

LEMMA 7. *A process makes $\Theta(N)$ RMRs in a passage of the algorithm in Figure 2 in the DSM and CC models.*

PROOF. We do the proof for the DSM model here and omit the proof for the CC model due to space limitations.

A process p announces itself twice to every other process in the system: once when it first leaves the NCS (line 14), and a second time after it obtains its ticket (line 16). Each of these announcements incurs $\Theta(N)$ RMRs. Obtaining a ticket also incurs $\Theta(N)$ RMRs. No RMRs are made while waiting in the loop at line 19. Finally, in the exit protocol, p withdraws its announcement to enter the CS from every other process exactly once. This also incurs $\Theta(N)$ RMRs. Thus the RMR complexity is $\Theta(N)$. □

THEOREM 8. *The unshaded portion of the algorithm in Figure 2 satisfies k-exclusion, starvation freedom, and k-FCFS. Moreover, it has RMR complexity $\Theta(N)$ in the CC and DSM models.*

PROOF. The result follows from Lemmas 4, 5, 6, and 7. □

5. FIFE *k*-EXCLUSION

The algorithm presented in the preceding section does not satisfy the FIFE property. To see why, we illustrate a scenario in which FIFE is violated by the unshaded portion of the algorithm in Figure 2.

Assume that $k = 2$, and consider the following execution of processes p_1, p_2, p_3, and p_4: Process p_1 executes its doorway entirely, choosing a ticket $t_{p_1} > 0$. After this, process p_2 executes its doorway and advances into the CS. Process p_3 and process p_4 then execute their doorway, choosing tickets t_{p_3} and t_{p_4}, both of which are larger than t_{p_1}. Process p_3 and p_4 then stop taking steps temporarily. We assume this is the first time processes p_3 and p_4 have left the NCS, and so $Want[p_3][p_1] = 0$ and $Want[p_4][p_1] = 0$. This implies that when p_1 executes the loop at line 20, p_1 does not remove p_3 or p_4 from $p_1.predecessor\_set$, and hence does not advance into the CS, until either p_3 writes $Want[p_3][p_1] = t_{p_3}$ or p_4 writes $Want[p_4][p_1] = t_{p_4}$ (line 16). This violates FIFE, which requires that p_1 enters the CS in a bounded number of its own steps after p_2 enters the CS.

In this section we present a k-exclusion algorithm that satisfies FIFE, which turns out to be a simple modification of the preceding algorithm. The necessary modifications are shown shaded in gray in Figure 2. As before, the doorway consists of lines 14..15.

FIFE says that if a process p finishes the doorway before a process q starts the doorway, and q enters the CS before p, then p enters the CS in a bounded number of its own steps. To ensure this, just before the process q enters the CS, q *captures* all processes that have a smaller ticket (line 23). In particular, since process p finishes the doorway before q starts the doorway, process p's ticket will be strictly smaller than q's ticket. Process q captures p by setting the value of $Capture[q][p]$ to q's current ticket. This and the fact that p has a smaller ticket than q, means that after q enters the CS, it will be the case that $Ticket[p] < Capture[q][p]$. The next time process p evaluates the condition $Ticket[p] < Capture[q][p]$ at line 22, p sets $captured = \mathsf{true}$, allowing p to terminate the loop at line 19 and enter the CS in a bounded number of its own steps, as required to satisfy FIFE.

With this modification we open an additional path for a process to enter the CS – by finding that it has been captured. Starvation freedom is clearly not affected by this, but there is the potential for k-exclusion to be violated since it is now "easier" for a process to enter the CS. This does not happen, although the details behind why are somewhat subtle. Intuitively, the algorithm protects against this possibility by allowing processes to only capture other processes with smaller tickets.

THEOREM 9. *The algorithm in Figure 2 satisfies k-exclusion, starvation freedom, and FIFE. Moreover, it has RMR complexity $\Theta(N)$ in the DSM and CC models.*

6. *k*-EXCLUSION USING NON-ATOMIC READS AND WRITES

The algorithms presented above are correct when atomic reads and writes are used. A simple way to transform these algorithms to work with non-atomic reads and writes is to use Lamport's register constructions [19]. Recall, atomic (non-atomic) reads and writes in this paper have the same semantics as defined by Lamport for multireader single-writer atomic (safe) registers. Using Lamport's constructions, we can construct atomic registers from safe registers, i.e., simulate atomic reads and writes by using only non-atomic reads and writes.

The problem with using register constructions is that they are inefficient, since they increase the RMR and space complexity of an algorithm by at least a factor of $\Omega(N)$. It turns out that we can do better than this. In this section we modify the FIFE k-exclusion algorithm in Figure 2, without affecting either its asymptotic RMR or space complexity, so that it works when reads and writes are non-atomic. The new algorithm is given in Figure 3. The trying protocol consists of lines 29..45, and the doorway consists of lines 29..31.

To understand the new algorithm, we first explain why the algorithm in Figure 2 does not satisfy k-exclusion, starvation freedom, or FIFE if reads and writes are non-atomic. The problem is that there are no guarantees on the value a process reads from a variable if some other process is concurrently writing the variable. Consider, for example the case when $k = 1$. Suppose processes p and q concurrently execute line 15, and that p and q choose tickets t_p and t_q, respectively. Further suppose that $t_q < t_p$ and that after p and q are both done choosing tickets (line 15), q temporarily stops taking steps. Process p races ahead of q into the loop at line 19, and starts reading $Want[q][p]$ at line 21. At the same time, q starts writing $Want[q][p]$ at line 16. Since the reads and writes are not atomic, p can read any value from $Want[q][p]$. In particular, suppose that p reads a value from $Want[q][p]$ such that $t_p < Want[q][p]$. In this case, p removes q from its predecessor set. All other processes are in the NCS, and so p removes them from its predecessor set as well and advances into the CS. When q reaches line 21, it evaluates $(t_q, q) < (Want[p][q], p)$ to be true, since the ticket t_p that p last wrote to $Want[p][q]$ is larger than the ticket t_q that q chose. Thus q will remove p from its predecessor set, along with all the other processes, which are in the NCS. After this, q advances into the CS, and k-exclusion is violated for $k = 1$. A more complex scenario can be constructed for $k = 2$ to illustrate that starvation freedom and FIFE also do not hold.

To solve this problem, we employ a technique similar to one used in Lamport's register constructions. The technique involves reading variables in the opposite order of which they are written. We "duplicate" each $Want[i][j]$ and $Capture[i][j]$ variable in our algorithm. That is, for each i, j, we create $Want[i][j][1]$, $Want[i][j][2]$, $Capture[i][j][1]$, and $Capture[i][j][2]$. Wherever a write of value v to $Want[i][j]$ occurs in the algorithm in Figure 2, in the new algorithm we write v to $Want[i][j][1]$ and $Want[i][j][2]$, *in that order*. Wherever a read of variable $Want[i][j]$ occurs in the algorithm in Figure 2, in the new algorithm we read $Want[i][j][2]$ and $Want[i][j][1]$, *in that order*. Reading and writing $Capture[i][j][1]$ and $Capture[i][j][2]$ in the new algorithm is done analogously. Note that in the algorithm we present the duplicate writes on the same line (e.g., line 30). We use this notation for compactness; the writes are still distinct (non-atomic) operations and occur in the order that they appear on the line.

There is also a new condition checked at line 44 that was not in the version of the algorithm with atomic reads and writes. In that version, there was no harm in p capturing i repeatedly in successive passages. Here, however, if p continues capturing i in each passage that p executes, it could be that every time that i checks to see if it is captured by

```
shared variables:
    Want: array[1..N][1..N][1..2] of ℕ ∪ {∞} init all ∞
    Ticket: array[1..N] of ℕ init all 0
    Capture: array[1..N][1..N][1..2] of ℕ init all 0

    (DSM model: Ticket[p], Want[i][p][j], and Capture[i][p][j] are local to process p for all i, j)

private variables:
    predecessor_set: Set of ℕ
    captured: boolean

27 loop
28     NCS
29     foreach i ∈ {1..N} \ {p} do
30         Want[p][i][1] := Ticket[p]; Want[p][i][2] := Ticket[p]
31     Ticket[p] := 1 + max(Ticket[1], Ticket[2], ..., Ticket[N])
32     foreach i ∈ {1..N} \ {p} do
33         Want[p][i][1] := Ticket[p]; Want[p][i][2] := Ticket[p]
34     predecessor_set := {1..N} \ {p}
35     captured := false
36     while |predecessor_set| ≥ k ∧ ¬captured do
37         foreach i ∈ predecessor_set do
38             if (Ticket[p], p) < (Want[i][p][2], i) then
39                 if (Ticket[p], p) < (Want[i][p][1], i) then predecessor_set := predecessor_set \ {i}
40         foreach i ∈ {1..N} do
41             if Ticket[p] < Capture[i][p][2] then
42                 if Ticket[p] < Capture[i][p][1] then captured := true
43     foreach i ∈ {1..N} do
44         if Capture[p][i][1] ≤ Ticket[i] then
45             Capture[p][i][1] := Ticket[p]; Capture[p][i][2] := Ticket[p]
46     CS
47     foreach i ∈ {1..N} do
48         Want[p][i][1] := ∞; Want[p][i][2] := ∞
49 end loop
```

Figure 3: (Non-atomic Reads/Writes) FIFE k-exclusion algorithm for process $p \in \{1, ..., N\}$

p (on lines 41..42), it happens to read $Capture[p][i][1]$ and $Capture[p][i][2]$ as p is writing into them (on line 45) and so it does not find that it is captured. This neutralizes the capturing mechanism and can result in a violation of FIFE. To avoid the problem p does not attempt to capture any process it previously captured, by performing the test on line 44.

To gain some understanding about how the preceding changes fix the algorithm, consider again the problem scenario that we described above in which k-exclusion was violated. Process p and q leave the NCS for the first time, write 0 into all *Want* entries in the loop at line 29, and then concurrently execute line 31. Process p chooses a ticket t_p larger than the ticket t_q that q chooses, i.e., $t_q < t_p$, after which process q temporarily stops taking steps. Process p then races ahead of q into the loop at line 36. When p tests the condition at line 38, the only way p evaluates it to be true is if q writes $Want[q][p][2]$ at the same time that p reads $Want[q][p][2]$. For q to write $Want[q][p][2]$ (line 33), it must first finish writing $Want[q][p][1]$ (line 33). Thus, when p checks the condition at line 39, p will evaluate it to be false unless q starts another write of $Want[q][p][1]$ concurrently with p's read of $Want[q][p][1]$. However, for this to happen, q must start another passage of the algorithm, in which case it is safe for p to remove q from p's predecessor set.

The complete proof of this algorithm's correctness is somewhat delicate, as is common with algorithms in the model with non-atomic operations.

THEOREM 10. *The algorithm in Figure 3 satisfies k-exclusion, starvation freedom, and FIFE when reads and writes are non-atomic. Moreover, it has RMR complexity $\Theta(N)$ in the DSM and CC models.*

7. CONCLUSION

We have presented the first known k-exclusion algorithms that are local-spin in the CC and DSM models and that use only atomic reads and writes, along with a version of the algorithm that works even if the read and write operations are not atomic. These algorithms are structured with the same elegance of Lamport's Bakery algorithm.

One desirable feature of the Bakery algorithm that is not present in our k-exclusion algorithms is that each process resets its ticket to zero in the exit protocol. This means

that if there is a period of quiescence in which all processes are in the NCS, all tickets will be reset to zero. In contrast, the tickets in our algorithms necessarily grow without bound and are never reset. We have developed more complex algorithms in which tickets are reset to zero in the exit protocol, but omit them from this paper due to space limitations [7]. Like the Bakery algorithm, tickets can still grow without bound in these algorithms if at all times some process is outside its NCS.

There exist variations of Lamport's Bakery algorithm for mutual exclusion [16, 22] in which tickets are bounded and each shared variable requires only $\Theta(\log N)$ bits. The techniques used in these papers, unfortunately, are not easily adapted to our algorithms, since they rely on the fact that at most one process can execute through the CS at a time. Afek et al. [1] use a more generic scheme, known as a bounded concurrent timestamp system (BCTS) [11], to bound tickets in their Bakery-like (non-local-spin) k-exclusion algorithm. However, the BCTS implementation in [11] requires large shared variables each with size $\Omega(N)$ bits. In fact, all bounded timestamp systems require $\Omega(N)$ bits per timestamp [15], and thus tend to be impractical. Furthermore, a process executing the algorithm of Afek et al. can make an unbounded number of calls to the BCTS while waiting to enter the CS, and it is not clear how these calls can be made so as to incur only a bounded number of RMRs. Finding a variant of our algorithms that are local-spin and use bounded tickets, preferably requiring $O(\log N)$ bits per timestamp, remains an open problem.

We also leave open the problem of determining whether (and, if so, how) the RMR complexity of k-exclusion depends on k. This is not known both for algorithms that use only read and write operations, and for algorithms that can use more powerful synchronization primitives.

Acknowledgements. The author would like to thank Vassos Hadzilacos, Wojciech Golab, Hyonho Lee, and the anonymous referees for their helpful comments on preliminary versions of this paper.

8. REFERENCES

[1] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. A bounded first-in, first-enabled solution to the ℓ-exclusion problem. *ACM Transactions on Programming Languages and Systems*, 16(3):939–953, 1994.

[2] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proc. of 13th RTSS*, pages 12–21, 1992.

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

[4] J. H. Anderson and M. Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11(1):1–20, 1997.

[5] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. of 40th STOC*, pages 217–226, New York, NY, USA, 2008.

[6] J. E. Burns and G. L. Peterson. The ambiguity of choosing. In *Proc. of 8th PODC*, pages 145–157, New York, NY, USA, 1989.

[7] R. Danek. *Thesis (forthcoming)*. PhD thesis, University of Toronto, 2010.

[8] R. Danek and H. Lee. Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems. In *Proc. of 22nd DISC*, pages 512–513, 2008.

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

[10] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: ℓ-exclusion as a test case. In *Proc. of 20th STOC*, pages 78–92, New York, NY, USA, 1988.

[11] D. Dolev and N. Shavit. Bounded concurrrent time-stamp systems are constructible. In *Proc. of 21st STOC*, pages 454–466, New York, NY, USA, 1989.

[12] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proc. of 20th FOCS*, pages 234–254, 1979.

[13] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. *ACM Transactions on Programming Languages and Systems*, 11(1):90–114, 1989.

[14] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, 1983.

[15] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[16] P. Jayanti, K. Tan, G. Friedland, and A. Katz. Bounding Lamport's Bakery algorithm. In *Proc. of 28th SOFSEM*, pages 261–270, London, UK, 2001.

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

[18] L. Lamport. On interprocess communication. Part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[19] L. Lamport. On interprocess communication. Part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[20] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[21] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[22] G. Taubenfeld. The black-white Bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In *Proc. of 18th DISC*, pages 56–70, 2004.