An Improved k-Exclusion Algorithm

Meirav Zehavi
Department of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel
Email: meizeh@cs.technion.ac.il

***Abstract*— k-Exclusion is a generalization of Mutual Exclusion that allows up to k processes to be in the critical section concurrently. *Starvation Freedom* and *First-In-First-Enabled (FIFE)* are two desirable progress and fairness properties of k-Exclusion algorithms. We present the first known bounded-space k-Exclusion algorithm that uses only atomic reads and writes, satisfies Starvation Freedom, and has a bounded *Remote Memory Reference (RMR)* complexity. Our algorithm also satisfies FIFE, and has an RMR complexity of $O(n)$ in both the cache-coherent and distributed shared memory models.**

***Index Terms*— k-exclusion, shared memory, remote memory reference**

I. Introduction

We consider an interleaving model of concurrency, where executions are modeled as sequences of *steps*. Each step, performed by a single process, is a read or write to a shared variable, or a local computation. The code of each process is divided into four sections: (1) *Remainder*, (2) *Entry*, (3) *Critical*, and (4) *Exit*, that are executed cyclically in this order. *k-Exclusion* [1] allows up to k processes to be in the critical section (CS) concurrently; and *Bounded Exit* states that any process in the exit section finishes this section in a bounded number of its steps. A k-Exclusion algorithm is an algorithm defined by the entry and exit sections that satisfies these properties.

k-Exclusion is used to solve conflicts between multiple processes trying to access a shared resource in a multiprocessor system. It generalizes the well-studied Mutual Exclusion problem [2], which allows up to one process to be in the CS concurrently. Afek et al. [3] illustrate k-Exclusion by using the following example. Suppose each process controls some device which from time to time needs to enter a mode of high electrical power consumption. The main circuit breaker can withstand at most k devices at high electrical power consumption. By allowing each process to switch its device on only when it is in the CS, a k-Exclusion algorithm will protect the circuit breaker from burning out.

A *doorway* is a bounded piece of the entry code; and a process is *enabled in t steps* if it enters the CS in at most t of its steps. We next present two desirable progress and fairness properties of k-Exclusion algorithms [4].

- *Starvation Freedom*: If a non-faulty process p is in the entry section and at most $k-1$ other processes crash, then p eventually enters the CS. Thus, the algorithm can tolerate up to $k-1$ process crashes.
- *First-In-First-Enabled (FIFE)* (resp. *FIFE in t steps*): If a process p finishes the doorway before a process q starts the doorway, and q enters the CS before p, then p is enabled in $O(1)$ (resp. t) steps. FIFE is an adaptation of *First-Come-First-Served (FCFS)* (of Mutual Exclusion) to k-Exclusion (see [4]).

We consider two models for shared memory architectures: *Distributed Shared Memory (DSM)* and *Cache-Coherent (CC)*. In the DSM model, each process has a memory module that it accesses locally and the other processes access remotely. A *Remote Memory Reference (RMR)* occurs when a process accesses a shared variable that is located in the memory module of another process.

As in [4], we describe the write-through/write-invalidate CC model, although our results apply equally to the similar write-back/write-invalidate model. In this model, each process has a local cache, and there is a global memory store that all the processes access remotely. When a process reads a shared variable v that is not located in its local cache, it makes an RMR and caches a local copy of v. When a process writes to a shared variable v, it makes an RMR, writing to the global memory store, and invalidates the copies of v that are cached by other processes.

A *passage* is the time between when a process starts the entry section to when it next finishes the exit section.[1] The *RMR complexity* of an algorithm is the maximum number of RMRs a process makes in a passage. An algorithm is *local-spin* if its RMR complexity is bounded.

II. Prior Work and Our Contribution

Table I presents a summary of known local-spin k-Exclusion algorithms. The RMR Complexity column refers to the CC and DSM models, where c denotes the maximum number of processes simultaneously outside the remainder section. R denotes Read, W denotes Write, T&S denotes Test&Set, F&I denotes Fetch&Increment, and C&S denotes Compare&Swap. Note that an algorithm uses bounded space if it uses a bounded number of bits.

Peterson [5] gave an $O(n^3)$ RMR complexity bounded-space k-Exclusion algorithm for the CC model, using only Read and Write, which satisfies neither Starvation Freedom nor FIFE. Anderson et al. [6] gave $O(k\log(n/k))$ and $O(c)$ RMR complexity bounded-space k-Exclusion algorithms for both the CC and DSM models. These algorithms satisfy Starvation Freedom, using Read, Write, T&S and F&I. Danek et al. [7] gave an $O(k\log(n))$ RMR complexity bounded-space k-Exclusion algorithm for both the CC and DSM models, using only Read

[1]Time refers to positions of steps in an execution.

Ref.	RMR Complexity	Instruct.	Starv. Free.	FIFE	Bounded Space
[5]	CC: $O(n^3)$ DSM: ∞	R, W	No	No	Yes
[6]	$O(k \log \frac{n}{k})$	R, W, T&S, F&I	Yes	No	Yes
[6]	$O(c)$	R, W, T&S, F&I	Yes	No	Yes
[7]	$O(k \log n)$	R, W	No	No	Yes
[4]	$O(n)$	R, W	Yes	Yes	No
[8]	CC: $O(\log k)$ DSM: ∞	R, W, F&I, C&S	Yes	Yes	No
This paper	$O(n)$	R, W	Yes	Yes	Yes

TABLE I.
KNOWN LOCAL-SPIN k-EXCLUSION ALGORITHMS.

and Write, which satisfies neither Starvation Freedom nor FIFE.

Danek [4] gave an $O(n)$ RMR complexity k-Exclusion algorithm for both the CC and DSM models, using only Read and Write that are not necessarily atomic. Finally, Choi [8] gave an $O(\log k)$ RMR complexity k-Exclusion algorithm for the CC model, using Read, Write, F&I and C&S. These two algorithms satisfy both Starvation Freedom and FIFE, but use unbounded space.

There are several other known k-Exclusion algorithms [1], [3], [9]–[12], which are not local-spin in either the CC or DSM models.

Danek et al. [13] gave an $O(\log n)$ RMR complexity bounded-space Mutual Exclusion algorithm, using only Read and Write, which satisfies Starvation Freedom and FCFS.[2] Any Mutual Exclusion algorithm, using only Read, Write and stronger primitives such as C&S, which satisfies Starvation Freedom, has an RMR complexity of $\Omega(\log n)$ [15], [16]. Thus, the $O(\log n)$ upper bound is tight. For the *amortized* RMR complexity of such algorithms, this is not true (see [17], [18]).

We present the first known bounded-space k-Exclusion algorithm that uses only Read and Write, satisfies Starvation Freedom, and has a bounded RMR complexity. Our algorithm also satisfies FIFE, and has an RMR complexity of $O(n)$ in both the CC and DSM models. It is based on a *Concurrent Timestamp System (CTS)* [19], the *Bakery algorithm* [20], and a mechanism that allows waiting processes to rely on other processes to update them about their waiting statuses.

The paper is organized as follows. We first provide a brief overview of the Bakery algorithm and CTSs. We then present our k-Exclusion algorithm, and prove its correctness. Finally, we state directions for further research.

III. PRELIMINARIES

Our algorithm is based on the Bakery algorithm—a Mutual Exclusion algorithm that satisfies Starvation Freedom and FCFS, but is not local-spin (see the pseudocode below). When a process p is in the doorway, it obtains a ticket bigger than those of all the other processes that have already finished the doorway, but not the CS. Then, p waits until all the processes having smaller nonzero tickets finish the exit section.

[2]A more space-efficient algorithm was recently given in [14].

The Bakery algorithm: (for p)

```
Shared Variables:
1) Num[0...$n-1$] - init all 0
2) Choosing[0...$n-1$] - init all False
Entry: (doorway: lines 1–3)
1) Choosing[$p$] $\leftarrow$ True
2) Num[$p$] $\leftarrow$ 1 + max{Num[0], ..., Num[$n-1$]}
3) Choosing[$p$] $\leftarrow$ False
4) Foreach $i \in \{0,\dots,n-1\} \setminus \{p\}$:
    a) wait until Choosing[$i$] = False
    b) wait until (Num[$i$], $i$) > (Num[$p$], $p$) $\vee$
         Num[$i$] = 0
Exit:
5) Num[$p$] $\leftarrow$ 0
```

The Bakery algorithm does not bound the values of the tickets. Taubenfeld [21] gave a bounded-space version of this algorithm, relying on the fact that there is at most one process in the CS at a time, which is not true for k-Exclusion. We use a CTS to order the processes like the Bakery algorithm, while using bounded space.

For each process p, a CTS provides two procedures: Label$_p$ and Scan$_p$. The CTS stores an array of timestamps, ordered according to a serialization of the Label procedures. Each Scan returns the current permutation of the n identifiers of the processes.

Let P be a set of processes outside any Label procedure during some time interval. Then, the permutations returned by all the Scans performed entirely in this interval are consistent with one another with regards to the order they impose on P. Now, let p be a process outside Label$_p$ during an interval $[T_1^p, T_2^p]$, let q be a process performing Scan$_q$ during an interval $[T_1^q, T_2^q]$, such that $T_1^p \le T_1^q$, and, for any process i, let T_2^i be a time when any Label$_i$ starting before T_2^q is finished. If the Scan$_q$ that started at T_1^q returned a permutation satisfying $p < i$, then any Scan performed entirely in $[T_2^i, T_2^p]$, returns a permutation satisfying $p < i$.

Consider the following two properties of a CTS [3].

1) If a process p begins Label$_p$ after a process q finishes Label$_q$, then any Scan performed entirely after both labeling procedures and before any subsequent labeling procedures by q, returns a permutation in which $q < p$.
2) Let P be a set of processes, such that in some interval each $p \in P$ executes Label$_p$, and then Scan$_p$ which is not followed by Label$_p$. Then, there is $p \in P$ whose last Scan$_p$ in this interval returns a permutation that orders all the other processes in P before it, and no Label$_q$, for any $q \in P$, starts after the last Label$_p$ in this interval finishes.

There is a bounded-space CTS that satisfies these properties, using $O(n)$ reads and writes per Label or Scan execution [19]. In our algorithm, we use such a CTS.

IV. The Algorithm

Recall that we use a CTS to order processes like the Bakery algorithm, while using bounded space. Each process invokes Label (in the doorway) to obtain a place among the waiting processes. Afek et al. [3], who also gave a k-Exclusion algorithm that is based on a CTS and the Bakery algorithm, then handle each waiting process by simply allowing it to repeatedly invoke Scan until it discovers that there are less than k processes ahead of it. Thus, in a single passage, a process can invoke Scan an unbounded number of times. Since Scan makes RMRs, this approach results in an algorithm that is not local-spin.

We use a different approach. Our waiting processes do not invoke Scan, but rely on other processes to update them about their waiting statuses. We next describe the mechanism we have developed in applying this approach.

After a process p invokes Label$_p$, it invokes Scan$_p$ (in the doorway). Then, using an array called *Order*<, p informs each process i what it read about the order between them. Using arrays called *Reveal* and *Discover*, i relies on this information only if it is newer than the information i has from its last Scan$_i$.

When a process is in the doorway, it uses Reveal to reveal itself to the other processes; and when it is in the exit section, it uses Reveal to disguise itself from them. The processes in the doorway check which processes are revealed, and use Discover to inform them about their discoveries. If a process reads information from a process that has discovered it, it considers this information relevant. When a process is in the exit section, it deletes its traces from Discover.

Consider the following scenario. A process p discovers a process q, q disguises itself and deletes its traces from Discover, and only then p writes to Discover. Thus, q can next rely on irrelevant information. To avoid this problem, Discover has two cells for each possible discovery, and a discovery is relevant only if it is written in both cells.

Another problem arises when a process p executes only a part of the doorway, and then a process q uses new information written by p to deduce that old information (which p updates when it next executes the rest of the doorway) is relevant. Using an array called *Ignore*, we avoid this problem. A process turns on some flags in Ignore shortly after entering the doorway, and turns them off before finishing the doorway. A process does not rely on information in Order< which is written by a process that has announced it should currently be ignored.

Using an array called *Compete*, processes do not wait on processes ordered before them that are not competing with them on the CS. A process simply turns on some flags in Compete when it enters the doorway, and turns them off before finishing the exit section.

Finally, using an array called *Allow*, we achieve FIFE. Once a process is enabled, it informs the processes preceding it that they can enter the CS.[3] The necessity of Allow follows from the fact that in our algorithm, a process does not make its announcements to all the other processes at once (otherwise the algorithm will not be local-spin in the DSM model). Thus, a process p ordered *after* a process q can discover that it is enabled, while, without using Allow, q never discovers that it is enabled.

[3]Several Mutual Exclusion algorithms use an array similar to Allow (e.g., the promote array of Hendler et al. [17]).

We next present a high-level description of our algorithm (see the pseudocode below).

When a process p enters the doorway, it first informs the other processes that it is competing on the CS (line 1), and that they should currently ignore its information (line 2). Then, p obtains a place among the waiting processes (line 3). Afterwards, p checks which new processes are revealed, cancels the permission it might have given them to enter the CS, and informs them about its discoveries (line 4). The process p then reveals itself (line 5). Then, p invokes Scan$_p$, and informs the other processes what it read about the order between them (lines 6–7). Afterwards, p informs them that they should not ignore its information (line 8). It initializes a set called *precede* to hold all the processes preceding it (lines 9-10). Then, p waits until less than k competing processes in its precede set do not have relevant information that the order between them has changed (line 11(a)), or until some process has relevant information that p can enter the CS (line 11(b)). Finally, p allows each process preceding it to enter the CS (line 12).

In the exit section, p first disguises itself (line 13). Then, p deletes its traces from the discoveries of the other processes (line 14), and informs them that it is not competing on the CS (line 15).

Algorithm 1: (for p)

Shared Variables:

```
1) Compete[$0 \ldots n-1$][$0 \ldots n-1$] - init all False.
2) Ignore[$0 \ldots n-1$][$0 \ldots n-1$] - init all False.
3) Order< [$0 \ldots n-1$][$0 \ldots n-1$] - init immaterial.
4) Reveal[$0 \ldots n-1$] - init all False.
5) Allow[$0 \ldots n-1$][$0 \ldots n-1$] - init immaterial.
6) Discover[$0 \ldots n-1$][$0 \ldots n-1$][$0, 1$] - init all False.
```

DSM model: Compete[i][p], Ignore[i][p], Order< [i][p], Allow[i][p], Reveal[p], Discover[i][p][0], and Discover[i][p][1] are local to process p for all i.

Entry: (doorway: lines 1–10)

```
1) Foreach $i \in \{0, \ldots, n-1\}$: Compete[$p$][$i$] $\leftarrow$ True
2) Foreach $i \in \{0, \ldots, n-1\}$: Ignore[$p$][$i$] $\leftarrow$ True
3) Label$_p$
4) Foreach $i \in \{0, \ldots, n-1\}$ s.t. !Discover[$p$][$i$][0] $\vee$ !Discover[$p$][$i$][1]:
     a) Discover[$p$][$i$][0] $\leftarrow$ False
     b) Discover[$p$][$i$][1] $\leftarrow$ False
     c) Allow[$p$][$i$] $\leftarrow$ False
     d) If Reveal[$i$]: Discover[$p$][$i$][0] $\leftarrow$ True
     e) If Reveal[$i$]: Discover[$p$][$i$][1] $\leftarrow$ True
     f) If (!Discover[$p$][$i$][0] $\vee$ !Discover[$p$][$i$][1]):
          i) Discover[$p$][$i$][0] $\leftarrow$ False
          ii) Discover[$p$][$i$][1] $\leftarrow$ False
5) Reveal[$p$] $\leftarrow$ True
6) order $\leftarrow$ Scan$_p$
```

```
7) Foreach $i \in \{0,\dots,n-1\}$: Order< $[p][i] \leftarrow$ (order.$p <$ order.$i$)
8) Foreach $i \in \{0,\dots,n-1\}$: Ignore$[p][i] \leftarrow$ False
9) precede $\leftarrow \emptyset$
10) Foreach $i \in \{0,\dots,n-1\}$ s.t. (order.$i <$ order.$p$): precede.add($i$)
11) While |precede| $\geq k$:
    a) Foreach $i \in$ precede:
        i) If !Compete$[i][p]$: precede.delete($i$)
        ii) If (Discover$[i][p][0] \wedge$ Discover$[i][p][1] \wedge$ !Ignore$[i][p] \wedge$ !Order< $[i][p]$):
            precede.delete($i$)
    b) Foreach $i \in \{0,\dots,n-1\}$ s.t. Discover$[i][p][0] \wedge$ Discover$[i][p][1]$:
        i) If Allow$[i][p]$: precede $\leftarrow \emptyset$
12) Foreach $i \in \{0,\dots,n-1\}$ s.t. (order.$i <$ order.$p$): Allow$[p][i] \leftarrow$ True

Exit:
13) Reveal$[p] \leftarrow$ False
14) Foreach $i \in \{0,\dots,n-1\}$:
    a) Discover$[i][p][0] \leftarrow$ False
    b) Discover$[i][p][1] \leftarrow$ False
15) Foreach $i \in \{0,\dots,n-1\}$: Compete$[p][i] \leftarrow$ False
```

V. Correctness

We first state three simple observations which are used throughout this section.

1) For all p and i, Compete$[p][i]$ is True iff p is in a passage in which it is after the execution of line 1 for i and before the execution of line 15 for i.
2) For all p and i, Ignore$[p][i]$ is True iff p is in a passage in which it is after the execution of line 2 for i and before the execution of line 8 for i.
3) For all p, Reveal$[p]$ is True iff p is in a passage in which it is after the execution of line 5 and before the execution of line 13.

Algorithm 1 uses a bounded-space CTS, $O(n^2)$ boolean variables, and $O(n)$ variables that require $O(n \log n)$ space. We thus have the following result.

Lemma 1: Algorithm 1 is bounded-space.

By the code, each process executes $O(n)$ steps in the exit section. We thus have the following result.

Lemma 2: Algorithm 1 satisfies Bounded Exit.

We next claim that if a waiting process p reads True from Discover$[q][p][0,1]$, then it had revealed itself before q discovered it (thus, updates which q wrote for p, such as Order< $[q][p]$, are relevant in the current passage of p). Moreover, Discover$[q][p][1]$ remains True as long as p is still waiting. This lemma will be used in the proofs of the k-Exclusion and Starvation Freedom properties.

Lemma 3: Let p and q be two processes, such that p executes the following instructions of line 11(a)ii or 11b in this order in a single passage.

1) p reads True from Discover$[q][p][0]$.
2) p reads True from Discover$[q][p][1]$.

Let T be the time when p executed the second instruction. Then, the last time before T when p executed line 5, had been before the last time before T when q checked the condition of line 4e for p. Also, after T and while p does not enter the exit section, Discover$[q][p][1]$ stays True.

Proof: Denote the time mentioned in the lemma when p reads Discover$[q][p][0]$ (resp. Discover$[q][p][1]$) by CD_0 (resp. CD_1). Next, consider the execution until and not including the point after CD_1 when p enters the exit section (if no such point exists, consider all the execution). Note that $CD_0 < CD_1$.

The only processes writing to Discover$[q][p][0]$ and Discover$[q][p][1]$, which are initially False, are q and p. Only q writes True to Discover$[q][p][0]$ (resp. Discover$[q][p][1]$), and only in line 4d (resp. 4e). The process q writes False to Discover$[q][p][0]$ (resp. Discover$[q][p][1]$) only in lines 4a and 4(f)i (resp. 4b and 4(f)ii). The process p writes False to Discover$[q][p][0]$ (resp. Discover$[q][p][1]$) only in line 14a (resp. 14b).

Since p reads True from Discover$[q][p][0]$ at CD_0, there is a point $AD_0 < CD_0$ when q executes the assignment of line 4d for p, and Discover$[q][p][0]$ stays True in $[AD_0, CD_0]$. Similarly, there is a point $AD_1 < CD_1$ when q executes the assignment of line 4e for p, and Discover$[q][p][1]$ stays True in $[AD_1, CD_1]$. Denote by CR_0 (resp. CR_1) the time when q executes its last step before AD_0 (resp. AD_1). At CR_0 and CR_1, q reads True from Reveal$[p]$. If $AD_1 < AD_0$, then q executes line 4b for p in (AD_1, AD_0). Thus, $AD_1 < CD_1 < AD_0 < CD_0$, which is a contradiction. We get that $CR_0 < AD_0 < CR_1 < AD_1$.

Denote by R the last time p executes line 5. From R on, if False is assigned to Discover$[q][p][0]$ or to Discover$[q][p][1]$, this assignment is done by q.

Since $AD_0 < CR_1 < AD_1$, q executes at least two more steps after AD_0, which we denote by S_1 and S_2.

Consider the following two cases.

1) $R < S_1$. By Observation 3, q reads True from Reveal$[p]$ at S_1, and executes the assignment of line 4e for p at S_2. At this point, both Discover$[q][p][0]$ and Discover$[q][p][1]$ are True, and according the code, they stay True.
2) $S_1 < R$. Then, $CR_0 < AD_0 < R$. By Observation 3, there are points R' and E' s.t. $R' < CR_0 \leq E' < R$, p executes line 5 at R', and during $[R', E']$, p is in the same passage as it is at R' and it has not yet executed line 13. Denote by F_0 the last time after E' when p executes line 14a for q . Note that $F_0 < R$, and p is in consecutive passages at F_0 and R. Since Discover$[q][p][0]$ is True during $[AD_0, CD_0]$ and $F_0 < R < CD_0$, we get that $CR_0 < F_0 < AD_0$.
By Observation 3 and since $F_0 < S_1 < R$, Reveal$[p]$ is False at S_1. Thus, S_2 is the first part of the condition of line 4f. Then, q executes the second part, and reads Discover$[q][p][1]$ as False. Afterwards, at a time we denote by T^*, q assigns False to Discover$[p][q][0]$. $CD_0 < T^*$, and thus $R < T^* < CR_1 < AD_1$. By Observation 3

and the code, after T^*, q performs a passage in which it executes the assignments of lines 4d and 4e for p. During the time interval between T^* and the second assignment, Discover$[q][p][1]$ is False, and afterwards it stays True. Thus, after AD_1, Discover$[q][p][1]$ stays True.

■

Lemma 4: Algorithm 1 satisfies k-Exclusion.

Proof: Suppose, by way of contradiction, that there is an execution in which there is a point T when a process enters the CS although there are k processes in the CS. In the rest of the proof, consider this execution until T.

Let P be the set of $k+1$ processes in the CS at T. By Property 2 of a CTS, there is a process p in P whose last Scan$_p$ returned a permutation that ordered all of the other processes in P before it, and no Label$_q$, for any $q \in P$, started after the last Label$_p$ had finished. Denote the set of such processes by C. Each $p \in C$, in its last execution of line 10, initialized precede to include all the k processes in $P \setminus \{p\}$. Let p be the process in C s.t. each $q \in C \setminus \{p\}$ had started its last Scan$_q$ before p started its last Scan$_p$. Denote $S = \{i \in P :$ the last Scan$_i$ returned a permutation in which $p < i$, and the last Label$_p$ had finished before the last Scan$_i$ started$\}$.

Claim 1. $S = \emptyset$.

Proof: Suppose, by way of contradiction, that there is a process $i \in S$. Since the last Scan$_p$ and Scan$_i$ returned permutations that are not consistent with regards to the order they impose on $\{p, i\}$, the last Scan$_p$ had started before the last Label$_i$ finished. Thus, the last Scan$_p$ had started before the last Scan$_i$ started. By our choice of p, we get that $i \in P \setminus C$.

Suppose that there is a process $j \in P$ that started a Label$_j$ after i had finished its last Label$_i$. Since p finished its last Label$_p$ after j had started its last Label$_j$, p finished its last Label$_p$ after i had finished its last Label$_i$. Thus, the last Scan$_p$ started after i had finished its last Label$_i$, which is a contradiction.

Since $i \in P \setminus C$, there is a process $j \in P$ s.t. the last Scan$_i$ returned a permutation in which $i < j$. Thus, the last Scan$_i$ returned a permutation in which $p < i < j$. By Property 2 of a CTS, the last Scan$_j$ returned a permutation in which $i < j$. Since the last Scan$_p$ and Scan$_i$ returned permutations that are not consistent with regards to the order they impose on $\{p, j\}$, the last Scan$_p$ had started before the last Label$_j$ finished. Thus, the last Label$_p$ had finished before the last Scan$_j$ started.

If the last Scan$_i$ had started before the last Scan$_j$ started, then the last Scan$_i$ and Scan$_j$ returned permutations that are consistent with regards to the order they impose on $\{p, i\}$. Else, the last Scan$_i$ and Scan$_j$ returned permutations that are consistent with regards to the order they impose on $\{p, j\}$. In the first case, the permutation returned by the last Scan$_j$ satisfies $p < i < j$, and in the second, it satisfies $p < j$. We get that $j \in S$.

Since we chose $i \in S$ arbitrarily, there is a function $f : S \to S$ s.t. for each $s \in S$, the last Scan$_s$ returned a permutation in which $s < f(s)$. By Property 2 of a CTS, there is a process $s^* \in S$ whose last Scan$_{s^*}$ returned a permutation that ordered all the other processes in S before it, and thus we have a contradiction. ■

Let A be the set of every process i that executed line 5 after the last Label$_p$ had finished, entered line 12 afterwards, and either i is p or the previously mentioned execution of line 5 was followed by a Scan$_i$ that returned a permutation in which $p < i$. Clearly, $A \neq \emptyset$. For any $i \in A$, we use the following notation. If $i = p$, then Scan$^*{}_i$ denotes the last Scan$_p$; else, it denotes the last Scan$_i$ that started after the last Label$_p$, returned a permutation in which $p < i$, and afterwards i entered line 12. Let Label$^*{}_i$ denote the Label$_i$ that precedes Scan$^*{}_i$.

Let q be the process in A whose last execution of line 5 before Scan$^*{}_q$ was after the last execution of line 5 before Scan$^*{}_i$ of any other $i \in A$. Denote the time after Scan$^*{}_q$ when q enters line 12 by T^*. There is a process i s.t. one of the following conditions holds.

1) $i \in P \setminus \{q\}$ and Compete$[i][q]$ was False at some point during the last time before T^* when q was in the while-loop. By Observation 1 and since i is in the CS at T, there was a point during the last time before T^* when q was in the while-loop in which i was before its last execution of line 1 for q. Note that the last Label$_i$ had started before the last Label$_p$ finished, and the last Label$_p$ had finished before Scan$^*{}_q$ started. Thus, we have a contradiction.
2) $i \in P \setminus \{q\}$ and q did not add i to its precede set during the last time before T^* when it executed line 10. Thus, Scan$^*{}_q$ returned a permutation in which $q < i$. We get that $q \neq p$, and Scan$^*{}_q$ returned a permutation in which $p < q < i$ (thus, $p \neq i$).
Since Scan$_p$ and Scan$^*{}_q$ returned permutations that are not consistent with regards to the order they impose on $\{p, i\}$, the last Scan$_i$ started after the last Label$_p$ had finished. Thus, the last Scan$_i$ returned a permutation in which $p < i$ (in order to be consistent with Scan$^*{}_q$), and we get that $i \in S$. By Claim 1, we have a contradiction.
3) $i \in P \setminus \{q\}$ and q calculated (Discover$[i][q][0] \wedge$ Discover$[i][q][1] \wedge$!Ignore$[i][q] \wedge$!Order< $[i][q]$) as True during the last time before T^* when q was in the while-loop. By Lemma 3, the last time before T^* when q executed line 5 preceded the last time when i checked the condition of line 4e for q. Thus, the last Scan$_i$ started after the last time before T^* when q executed line 5, which is also after the last Label$_p$ and Label$^*{}_q$ had finished. By Claim 1, $i = p$ or the last Scan$_i$ returned a permutation in which $i < p$.
By Observation 2 and since the last Label$_i$ had started before the last Label$_p$ finished, q read Ignore$[i][q]$ as False when i was after its last execution of line 8 for q. Thus, q next calculated !Order< $[i][q]$ as True after the last time when i executed line 7 for q; therefore the last Scan$_i$ returned a permutation in which $q < i$. Thus, $q \neq p$, the last Scan$_i$ returned a permutation in which $q <$

$i \leq p$, and Scan$^*{}_q$ returned a permutation in which $p < q$. Since the last Scan$_i$ and Scan$^*{}_q$ started after the last Label$_p$ and Label$^*{}_q$ had finished, we get a contradiction.

4) q calculated (Discover$[i][q][0] \wedge$ Discover$[i][q][1] \wedge$ Allow$[i][q]$) as True during the last time before T^* when q was in the while-loop. Denote by Q_1 and Q_2 the times when q read True from Discover$[i][q][1]$ and Allow$[i][q]$, respectively, when it executed this calculation. Note that $Q_1 < Q_2$.
Note that i is the only process that writes True to Discover$[i][q][1]$ (only in line 4e) and Allow$[i][q]$ (only in line 12), which are initially False. Thus, there are points I_1 and I_2 when i executed the assignments of lines 4e and 12 for q, respectively, s.t. Discover$[i][q][1]$ was always True in $[I_1, Q_1]$ and Allow$[i][p]$ was always True in $[I_2, Q_2]$. By Lemma 3, Discover$[i][p][1]$ stayed True from Q_1 until the time E when q next entered the exit section, and the last time T_r before Q_1 when q executed line 5 preceded the last time T_c before Q_1 when i checked the condition of line 4e for q.
Suppose that $I_2 < T_c$. Then, there is a point in (I_2, T_c) when i executed line 4b for q. Since Discover$[i][q][1]$ is always True in $[I_1, E]$, $I_2 < I_1$. Allow$[i][q]$ was False at I_1, since i executed line 4c for q before I_1 s.t. its following execution of line 12 for q was after I_1. Thus, $I_2 < Q_2 < I_1 < Q_1$. Since $Q_1 < Q_2$, we have a contradiction.
Since $T_c < I_2$, i executed line 5 and Scan$_i$ in (T_c, I_2). Thus, we can denote by T'_r the last time before I_2 when i executed line 5, and by T_s the last time before I_2 when i started Scan$_i$. We get that $T_r < T_c < T'_r < T_s < I_2$. Thus, the last Label$_p$ had finished before T'_r. Moreover, the Scan$_i$ that started at T_s returned a permutation in which $q < i$ (otherwise the assignment at I_2 would not have happened). This implies that $i \neq q$.
If $q = p$, then clearly $i \in A$. Else, since the last Label$_p$ and Label$^*{}_q$ had finished before T_s, and Scan$^*{}_q$ returned a permutation in which $p < q$, we get that the Scan$_i$ that started at T_s returned a permutation in which $p < q < i$. Again, we get that $i \in A$. This is a contradiction to our choice of q, since we have proved that $T_r < T'_r < T_s$, and that i is in line 12 after T_s.

■

Lemma 5: Algorithm 1 satisfies FIFE in $O(n)$ steps.

Proof: Let p and q be two processes such that p finishes the doorway at T_1, q enters the doorway at $T_2 > T_1$, q enters the CS at $T_3 > T_2$, and

- p is in the entry section during $[T_1, T_3]$.
- q is in the entry section during $[T_2, T_3)$.

When we refer to an instruction executed by p (resp. q), consider its last execution by p (resp. q) before T_3.

By Observation 3, Reveal$[p]$ is True during $[T_1, T_3]$. Thus, if q reads Discover$[q][p][0]$ or Discover$[q][p][1]$ as False when executing line 4 for p, then it next assigns True to both of them. The only other process that can write False to these variables is p, and only in the exit section. Since p is in the entry section during $[T_1, T_3]$, when q finishes executing line 4, Discover$[q][p][0]$ and Discover$[q][p][1]$ are True. By the code, while p does not enter the CS, they stay True.

Note that Label$_p$ finishes before Label$_q$ starts. By Property 1 of a CTS, the next Scan$_q$ returns a permutation in which $p < q$. Thus, when q next executes line 12, it writes True to Allow$[q][p]$. Since q is the only process that writes False to Allow$[q][p]$, and only in line 4c after reading Discover$[q][p][0]$ or Discover$[q][p][1]$ as False, there is a time T s.t. $T_2 < T \leq T_3$, after which Discover$[q][p][0]$, Discover$[q][p][1]$ and Allow$[q][p]$ are True as long as p does not enter the CS. Thus, from T on, p makes $O(n)$ steps after which it executes precede $\leftarrow \emptyset$ in line 11(b)i. It then executes $O(n)$ additional steps, after which it enters the CS. ■

Lemma 6: Algorithm 1 satisfies Starvation Freedom.

Proof: Suppose, by way of contradiction, that Algorithm 1 does not satisfy Starvation Freedom. Thus, by Lemmas 2 and 5, there is an execution in which there are less than k faulty processes, and there is a point T s.t. from T on, all non-faulty processes are in the remainder section or starved in the while-loop. Denote this set of starved processes by P. Observation 2 implies that from T on, for any non-faulty process p and any process i, !Ignore$[p][i]$ is True. Note that from T on, the values of all the shared variables do not change.

Let p be a process in P. Since p is stuck in the while-loop, there is a set W of at least k processes which were ordered before p in the permutation returned by the last Scan$_p$, and p cannot remove any of them from its precede set. Since there are less than k faulty processes, there is a non-faulty process $w \in W$. Note that from T on, (Discover$[w][p][0] \wedge$ Discover$[w][p][1] \wedge$!Order$<[w][p]$) is False. Moreover, from T on, !Compete$[w][p]$ is False, and thus, by Observation 1, $w \in P$.

Since we chose $p \in P$ arbitrarily, there is a function $f : P \rightarrow P$ s.t. for all $p \in P$, the last Scan$_p$ returned a permutation in which $f(p) < p$, and from T on, (Discover$[f(p)][p][0] \wedge$ Discover$[f(p)][p][1] \wedge$!Order$<[f(p)][p]$) is False. Note that from T on, !Order$<[p][f(p)]$ is True. Also, there is a nonempty set $\{s_0, s_1, \ldots, s_{m-1}\} = S \subseteq P$ s.t. for $0 \leq i \leq m-1$, $f(s_i) = s_{(i+1) \bmod m}$. Let (f, S) be such a pair s.t. S is of minimum size. Note that $m \geq 2$.

By Property 2 of a CTS, we can assume w.l.o.g that the last Scan$_{s_1}$ returned a permutation that ordered all the processes in $S \setminus \{s_1\}$ before s_1, and no Label$_s$, for any $s \in S$, started after the last Label$_{s_1}$ had finished. From T on, !Order$<[s_1][s]$ is True for all $s \in S$. Thus, from T on, (Discover$[s_1][s_0][0] \wedge$ Discover$[s_1][s_0][1]$) is False.

Claim 2. Let $i, j \in \{0, 1, \ldots, m-1\}$ s.t. $i \neq j$, and the last Scan$_{s_i}$ had started before the last Label$_{s_j}$ finished. Then, from T on, (Discover$[s_j][s_i][0] \wedge$ Discover$[s_j][s_i][1]$) is True.

Proof: The last execution of line 5 by s_i occurred before the last check of the condition of line 4 for s_i by s_j. By Observation 3, we get that after the last execution of line 5 by s_i, Reveal$[s_i]$ forever stays True. Since $s_i \in P$, it does not next execute lines 14a and 14b.

If when s_j last checked the condition of line 4 for s_i, (Discover$[s_j][s_i][0]$ $\wedge$ Discover$[s_j][s_i][1]$) was True, then, by the code, it forever stays True. Else, after s_j last checked this condition, it executed the assignments of lines 4d and 4e. Then, (Discover$[s_j][s_i][0]$ $\wedge$ Discover$[s_j][s_i][1]$) was True, and by the code, it forever stays True. Thus, from T on, (Discover$[s_j][s_i][0]$ $\wedge$ Discover$[s_j][s_i][1]$) is True. ■

By Claim 2, the last Scan$_{s_0}$ started after the last Label$_{s_1}$ had finished. Since the last Scan$_{s_0}$ and Scan$_{s_1}$ returned permutations that are not consistent with regards to the order impose on $\{s_0, s_1\}$, the last Scan$_{s_1}$ had started before the last Label$_{s_0}$ finished. By Claim 2, from T on, (Discover$[s_0][s_1][0]$ $\wedge$ Discover$[s_0][s_1][1]$) is True, and thus (Discover$[s_0][s_1][0]$ $\wedge$ Discover$[s_0][s_1][1]\wedge$!Order< $[s_0][s_1]$) is True.

Denote $t_0 = r_0 = s_0$, $t_1 = s_2$ and $r_1 = s_1$. For all $i \in \{2, \ldots, m-2\}$, denote $t_i = r_i = s_{i+1}$. For all $i \in \{0, \ldots, m-2\}$, denote $g(t_i) = t_{(i+1) mod(m-1)}$ and $h(r_i) = r_{(i+1) mod(m-1)}$. Denote $S^* = \{t_0, \ldots, t_{m-2}\}$ and $S^{**} = \{r_0, \ldots, r_{m-2}\}$. Consider the following cases.

1) The last Scan$_{s_1}$ started after the last Label$_{s_2}$ had finished. Thus, the last Scan$_{s_0}$ and Scan$_{s_1}$ returned permutations that are consistent with regards to the order they impose on $\{s_1, s_2\}$. Thus, the last Scan$_{s_0}$ returned a permutation in which $s_2 < s_1 < s_0$. Since (g, S^*) cannot contradict the choice of (f, S), (Discover$[s_2][s_0][0]$ $\wedge$ Discover$[s_2][s_0][1]$ $\wedge$!Order$[s_2][s_0]$) is True. By Lemma 3, the last Scan$_{s_0}$ and Scan$_{s_2}$ started after the last Label$_{s_0}$ and Label$_{s_2}$. Since their returned permutations are not consistent with regards to the order they impose on $\{s_0, s_2\}$, we have a contradiction.
2) The last Scan$_{s_1}$ had started before the last Label$_{s_2}$ finished. By Claim 2, from T on, (Discover$[s_2][s_1][0]$ $\wedge$ Discover$[s_2][s_1][1]$) is True. Since $f(s_1) = s_2$, from T on, !Order< $[s_2][s_1]$ is False. Thus, the last Scan$_{s_2}$ returned a permutation in which $f(s_2) < s_2 < s_1$. Since (h, S^{**}) cannot contradict the choice of (f, S), from T on, (Discover$[f(s_2)][s_1][0]$ $\wedge$ Discover$[f(s_2)][s_1][1]$ $\wedge$!Order< $[f(s_2)][s_1]$) is True. Thus, the last Scan$_{f(s_2)}$ returned a permutation in which $s_1 < f(s_2)$. By Lemma 3, the last time s_1 executed line 5 had been before the last time $f(s_2)$ executed the condition of line 4e for s_1. Thus, the last Label$_{s_1}$ had finished before the last Scan$_{f(s_2)}$ started. Consider the two following cases.
   a) The last Scan$_{s_2}$ started after the last Label$_{f(s_2)}$ had finished. Thus, the last Scan$_{s_2}$ and Scan$_{f(s_2)}$ started after the last Label$_{s_1}$ and Label$_{f(s_2)}$ had finished. Since they returned permutations that are consistent with regards to the order they impose on $\{s_1, f(s_2)\}$, we have a contradiction.
   b) The last Scan$_{s_2}$ had started before the last Label$_{f(s_2)}$ finished. By Claim 2, from T on, (Discover$[f(s_2)][s_2][0]$ $\wedge$ Discover$[f(s_2)][s_2][1]$) is True. Thus, from T on, !Order< $[f(s_2)][s_2]$ is False, and the last Scan$_{f(s_2)}$ returned a permutation in which $f(s_2) < s_2$. The last Scan$_{s_2}$ and Scan$_{f(s_2)}$ started after the last Label$_{s_1}$ and Label$_{s_2}$ had finished, and thus returned permutations that are consistent with regards to the order they impose on $\{s_1, s_2\}$, which is a contradiction.

■

Lemma 7: Algorithm 1 has $O(n)$ RMR complexity in both the DSM and CC models.

Proof: Algorithm 1 uses a CTS in which each Label and Scan performs $O(n)$ reads and writes. Thus, lines 3 and 6 require $O(n)$ RMRs in both models. Each of the other lines of the algorithm, except those of the while-loop, requires $O(n)$ RMRs in both models.

In the DSM model, each process reads only local variables in the while-loop, which do not make RMRs. Thus, Algorithm 1 has $O(n)$ RMR complexity in this model.

Now consider the CC model. Let v be a shared variable accessed by a process p in the while-loop. There is only one process which is not p that can write to v. Denote this process by p_v.

We next prove that while p is in the while-loop, it performs $O(1)$ RMR reads of v, or $O(1)$ RMR reads of v after which it is enabled in $O(n)$ steps. Since there are $O(n)$ shared variables accessed by p in the while-loop, this implies that Algorithm 1 has $O(n)$ RMR complexity in the CC model. In each of the following cases, assume some minimum constant number of writes to v while p is in the while-loop (otherwise, our claim clearly holds).

1) $v =$ Compete$[i][p]$. Thus, $p_v = i$ and $i \in$ precede. i only writes to v in lines 1 and 15. Next we refer only to writes to v. After at most two writes, i enters the doorway, and then executes the write in line 1 and the write in line 15. Thus, while p is in the while-loop and before its fourth RMR read of v, i enters the doorway and then the CS. By Lemma 5, before the fourth RMR read of v, p is enabled in $O(n)$ steps.
2) $v =$ Ignore$[i][p]$. Thus, $p_v = i$ and $i \in$ precede. i only writes to v in lines 2 and 8. Next we refer only to writes to v. After at most two writes, i enters the doorway, and then executes the write in line 2, the write in line 8, and again the write in line 2. Thus, while p is in the while-loop and before its fifth RMR read of v, i enters the doorway and then the CS. By Lemma 5, before the fifth RMR read of v, p is enabled in $O(n)$ steps.
3) $v =$ Order< $[i][p]$. Thus, $p_v = i$ and $i \in$ precede. i only writes to v in line 7. Next we refer only to writes to v. After at most one write, i enters the doorway, and then executes the write in line 7. Its

next write to v is again in line 7. Thus, while p is in the while-loop and before its third RMR read of v, i enters the doorway and then the CS. By Lemma 5, before the third RMR read of v, p is enabled in $O(n)$ steps.

4) $v =$ Discover$[i][p]$. Thus, $p_v = i$. i only writes to v in lines 4a, 4b, 4d, 4e, 4(f)i and 4(f)ii. Next we refer only to writes to v. By Observation 3, while p is in the while-loop, Reveal$[p]$ is True. Thus, after at most five writes, i executes the writes in lines 4a, 4b, 4d and 4e. Then, both Discover$[i][p][0]$ and Discover$[i][p][1]$ are True. We get that while p stays in the while-loop, both Discover$[i][p][0]$ and Discover$[i][p][1]$ stay True, and the condition of line 4 stays False. Thus, there are no additional writes to v. We get that there are at most nine writes to v while p is in the while-loop.
5) $v =$ Allow$[i][p]$. Thus, $p_v = i$. i only writes to v in lines 4c and 12. Next we refer only to writes to v. After at most two writes, i enters the doorway, and then it may execute the writes in lines 4c and 12. Before the next write, i enters the CS. Thus, while p is in the while-loop and before its fifth RMR read of v, i enters the doorway and then the CS. By Lemma 5, before the fifth RMR read of v, p is enabled in $O(n)$ steps.

■

Finally, by Lemmas 1, 2, 4, 5, 6 and 7, we get the following result.

Theorem 1: Algorithm 1 is a bounded-space k-Exclusion algorithm that satisfies FIFE and Starvation Freedom. Moreover, it has an RMR complexity of $O(n)$ in both the DSM and CC models.

VI. Conclusion and Future Work

We have presented the first known bounded-space k-Exclusion algorithm that uses only Read and Write, satisfies Starvation Freedom, and has a bounded RMR complexity. We have also proved that our algorithm satisfies FIFE, and has an RMR complexity of $O(n)$ in both the CC and DSM models.

One direction for future research is to improve the RMR complexity of our algorithm. The only known algorithm with a better RMR complexity uses unbounded space and synchronization primitives stronger than Read and Write, and is local-spin only in the CC model [8].

The RMR complexity of the class of Mutual Exclusion algorithms that use only Read and Write has an $\Omega(\log n)$ lower bound [15]. Thus, another direction for future research is to prove a better lower bound for the RMR complexity of the class of k-Exclusion algorithms that use only Read and Write (or prove that it does not exist).

Finally, we note that it might be interesting to study the amortized RMR complexity of k-Exclusion algorithms.

Acknowledgment

The author would like to thank Prof. Hagit Attiya for her helpful comments.

References

[1] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Resource allocation with immunity to limited process failure," in *Proc. FOCS*, 1979, pp. 234–254.
[2] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, 1965.
[3] Y. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "A bounded first-in, first-enabled solution to the ℓ-exclusion problem," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 939–953, 1994.
[4] R. Danek, "The k-bakery: local-spin k-exclusion using non-atomic reads and writes," in *Proc. PODC*, 2010, pp. 36–44.
[5] G. L. Peterson, "Myths about the mutual exclusion problem," *Inf. Process. Lett.*, vol. 12, no. 3, pp. 115–116, 1981.
[6] J. H. Anderson and M. Moir, "Using local-spin k-exclusion algorithms to improve wait-free object implementations," *Distrib. Comput.*, vol. 11, no. 1, pp. 1–20, 1997.
[7] R. Danek and H. Lee, "Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems," in *Proc. DISC*, 2008, pp. 512–513.
[8] J. Choi, "A solution to k-exclusion with $\mathrm{O}(\log k)$ RMR complexity," Darthmouth Technical Reports, Tech. Rep. TR2011-682, 2011.
[9] J. E. Burns and G. L. Peterson, "The ambiguity of choosing," in *Proc. PODC*, 1989, pp. 145–157.
[10] D. Dolev, E. Gafni, and N. Shavit, "Toward a non-atomic era: ℓ-exclusion as a test case," in *Proc. STOC*, 1988, pp. 78–92.
[11] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin, "Distributed FIFO allocation of identical resources using small shared space," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 1, pp. 90–114, 1989.
[12] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph, "Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors," *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 164–189, 1983.
[13] R. Danek and W. M. Golab, "Closing the complexity gap between FCFS mutual exclusion and mutual exclusion," *Distrib. Comput.*, vol. 23, no. 2, pp. 87–111, 2010.
[14] A. A. Aravind, "Simple, space-efficient, and fairness improved FCFS mutual exclusion algorithms," *J. Parallel Distrib. Comput.*, vol. 73, no. 8, pp. 1029–1038, 2013.
[15] H. Attiya, D. Hendler, and P. Woelfel, "Tight RMR lower bounds for mutual exclusion and other problems," in *Proc. PODC*, 2008, pp. 217–226.
[16] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel, "RMR-efficient implementations of comparison primitives using read and write operations," *Distrib. Comput.*, vol. 25, no. 2, pp. 109–162, 2012.
[17] D. Hendler and P. Woelfel, "Randomized mutual exclusion with sub-logarithmic RMR-complexity," *Distrib. Comput.*, vol. 24, no. 1, pp. 3–19, 2011.
[18] G. Giakkoupis and P. Woelfel, "A tight RMR lower bound for randomized mutual exclusion," in *Proc. STOC*, 2012, pp. 983–1002.
[19] C. Dwork, M. Herlihy, S. A. Plotkin, and O. Waarts, "Time-lapse snapshots," *SIAM J. Comput.*, vol. 28, no. 5, pp. 1848–1874, 1999.
[20] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.
[21] G. Taubenfeld, "The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms," in *Proc. DISC*, 2004, pp. 56–70.

Meirav Zehavi is a Ph.D. student at the Technion - Israel Institute of Technology, Haifa, Israel.