6-3-2011

A Solution to k-Exclusion with O(logk) RMR Complexity

Jonathan H. Choi
Dartmouth College

A Solution to k-Exclusion with $O(\log k)$ RMR Complexity

Dartmouth Computer Science Technical Report TR2011-682

Jonathan Choi
Thesis Advisor: Prasad Jayanti

June 3, 2011

Abstract

We specify and prove an algorithm solving k-Exclusion, a generalization of the Mutual Exclusion problem. k-Exclusion requires that at most k processes be in the Critical Section (CS) at once; in addition, we require bounded exit, starvation freedom and fairness properties. The goal within this framework is to minimize the number of Remote Memory References (RMRs) made. Previous algorithms have required $\Omega(k)$ RMRs in the worst case. Our algorithm requires $O(\log k)$ RMRs in the worst case under the Cache-Coherent (CC) model, a considerable improvement in time complexity.

1 Introduction

The k-Exclusion problem, first introduced by Fischer et al. [1], is a generalization of the well-known Mutual Exclusion problem. Mutual Exclusion requires that at most one process be in the *Critical Section* (CS) of code at any point, whereas k-Exclusion requires that at most k processes be in the CS at any point for $k \geq 1$. We seek to minimize the algorithm's worst-case time complexity, as measured in *Remote Memory References* (RMRs). Previous work has produced algorithms with worst-case complexity of $O(n)$ and $O(k)$ RMRs, depending on the memory model used. This paper specifies and proves an algorithm with $O(\log k)$ worst-case RMR complexity in the *Cache Coherent* (CC) model.

1.1 Model

1.1.1 Basics

The basic requirement of any exclusion algorithm is to limit the number of processes in a certain section of code, designated the *Critical Section* (CS). The contents of the CS are irrelevant to k-Exclusion; however, we could imagine the CS code accessing a scarce resource. In addition, we conventionally designate a *Remainder Section*, representing the code in which the k-Exclusion routine is couched. Its contents are also irrelevant. Each process may enter from and exit to the remainder section multiple times, with no guarantees about timing. Finally, we designate the section of code in our algorithm before the CS the *Try Section* and designate the section of code after the CS the *Exit Section*.

The theoretical framework for a collection of processes is that we think of each state as a *configuration*, denoted $\mathcal{C}$. Each process has a transition function mapping $\mathcal{C}$ to a set of configurations conceptually reachable with a single step (configurations one step away from $\mathcal{C}$ are denoted $\mathcal{C}$.s). An *execution* is a sequence of configurations that can be obtained with repeated applications of transitions function on the initial configuration. The correctness and complexity of our algorithm must hold over all such executions.

Finally, specific processes may be either *enabled* or *crashed*. A process is *enabled* in a configuration if and only if it will enter the CS in a finite number of its own steps, regardless of the steps taken by other processes. A process is *crashed* if, in an infinite execution, it takes no more steps.

1.1.2 Memory Models and Time Complexity

There are two dominant memory paradigms in the field of distributed computing, known as the *Cache Coherent* (CC) and *Distributed Shared Memory* (DSM) models. In the CC model, each process has a number of *local* variables that are costless to access, but accessible only by that specific process; there are also a number of *shared* (or *global*) variables that are costly to access, but once accessed, can be cached and waited upon at no cost until the global variable is modified. Moreover, the CC model allows each process to *spin* on multiple variables, paying an upfront cost for a read and subsequently only paying when each variable is recached. (This fact is crucial in the algorithm).

The DSM model treats local variables identically, but associates global variables with specific processes as well. Under DSM, access to a global variable is free for the process associated with it, and costly for every other process. That cost must be paid each time access is made, in contrast to the CC model. Intuitively, processes in the CC model access a shared pool of global variables,

while processes in the DSM model each have global memory modules which can be accessed by their peers. Each model has advantages and disadvantages; the two paradigms are known not to be equivalent in power. DSM is typically thought of as more versatile, but CC allows strictly more efficient solutions to k-Exclusion.

Closely associated with the two paradigms is the idea of cost and time complexity. The time complexity of an operation is defined in terms of *Remote Memory References* (RMRs). RMRs occur whenever a CC process recaches a global variable, or modifies a global variable; likewise, RMRs occur whenever a DSM process accesses or modifies a shared variable on another process. RMR costs are incurred by the particular process doing the reading or writing. Thus we can consider both per-process and amortized RMR complexity.

1.1.3 Shared Objects

Finally, the types of allowable shared variables are important to the algorithm. We make reference to three kinds: *read/write* objects, *Fetch and Increment* (F&I) objects, and *Compare and Swap* (CAS) objects. The objects are distinguished by the operations that they support *atomically*. An atomic operation is an operation that takes a single step in any execution.

- **Read/Write**: supports *read* and *write*. $read(X)$ returns the value stored in X; $write(X)$ atomically overwrites the value currently in X.
- **F&I**: supports *read* and *F&I*. $read(X)$ is as above; $F\&I(X)$ first returns the value in X, then increments it. The two operations occur atomically.
- **CAS**: supports *read* and *CAS*. $read(X)$ is still as above; $CAS(X, a, b)$ atomically sets $X \leftarrow b$ if $X = a$, and does nothing if $X \neq a$. It returns *true* if the swap occurs, *false* otherwise.

Our algorithm uses only F&I and CAS objects; $read(X)$ is often abbreviated by referring to X directly. For example, line **7** of the algorithm reads $a \leftarrow A[i][j]$, meaning that $A[i][j]$ is read, and the result is stored in local variable a.

The RMR complexity of each of these operations is constant for an initial read and constant for write, F&I and CAS operations (regardless of the success of the CAS). A process spinning on X will also incur an RMR every time X is modified. However, there are somewhat tricky border cases. In particular, we need to define whether a failed $CAS(X, a, b)$ causes recaching such that any process spinning on X incurs RMRs.

1.1.4 "Smart" Cache

Whether failed CAS causes RMRs on spinning processes or not is a hardware question out of the scope of this paper. For conservativeness, we will consider both the case where it does not ("Smart" Cache) and the case where it does. With Smart Cache, our algorithm has $O(\log k)$ worst-case per-process RMR complexity; without Smart Cache, it has $O(\log k)$ amortized per-process RMR complexity.

1.2 k-Exclusion

1.2.1 Intuition

The basic problem of k-Exclusion is to limit the number of processes in the critical section to at most k processes. However, this is not the only desirable property. We would like guaranteed progress under ordinary circumstances (clearly, though, if k processes have crashed in the CS, no further progress can occur). We would like to ensure fairness - that is, we would like to ensure that processes that have entered the try section earlier also enter the CS earlier. (We formalize this property with the *doorway*, a bounded fragment of code at the start of the algorithm). Finally, we would like to guarantee that processes past the CS will exit in a bounded number of steps.

1.2.2 Desired Properties

- **k-Exclusion**: At most k processes are in the CS at any time.
- **Starvation Freedom**: If fewer than k processes crash outside the remainder section, a non-crashing process in the try section eventually enters the CS.
- **First-In First-Enabled**: If a process p enters the doorway before another process p', and p' is in the CS, then either p already entered the CS or p is enabled to enter the CS.
- **Bounded Exit**: All processes complete their Exit Sections in a bounded number of steps, regardless of speeds or interleavings of the other processes.
- $O(\log k)$ **RMR Complexity**: Every process makes $O(\log k)$ remote memory references per iteration of the algorithm.

1.2.3 Previous Research

Several solutions to k-Exclusion already exist in the literature, both in the cache-coherent (CC) and distributed shared memory (DSM) models. Anderson and Moir [2] specify algorithms that satisfy k-Exclusion in either $\Theta(klog(n/k))$ or $\Theta(c)$ RMRs, where c is point contention. To do so, they require Read/Write, Fetch & Add (F&A) and Compare and Swap (CAS) objects. Danek [3] specifies and proves an algorithm with $\Theta(n)$ RMR complexity, but which requires only Read/Write objects.

The previously known algorithm with the lowest worst-case per-process run time is by Decker [4]. This algorithm requires Smart Cache and achieves $O(k)$ worst-case RMRs. (It is not known what the amortized cost of this algorithm would be in the absence of Smart Cache; its worst-case per-process RMR complexity would be $\Omega(n)$). The algorithm in this thesis represents an improvement in time complexity over Decker's, without an increase in space complexity.

Interestingly, Danek and Hadzilacos [5] provide a proof that no algorithm in the DSM model can improve upon a $\Omega(n)$ lower bound for RMR complexity. This demonstrates that for k-Exclusion, the CC model is significantly more powerful than the DSM model, although this is not necessarily true in general.

2 Algorithm

2.1 Intuition

Those new to k-Exclusion often suggest that a queue could be used to keep track of waiting processes (unenabled processes in the try section). Perhaps exiting processes could dequeue and enable before exiting, which would result in constant time complexity. This naive solution fails because a process could dequeue a waiting process and crash, thus failing to enable the waiting process and causing starvation. Starvation can be avoided if each exiting process steps through the queue, and only removes waiting processes from it once certain that they are enabled. However, a malign execution could result in an exiting process being continually pre-empted by other processes. The pre-empted process would never successfully enable a waiting process, and would therefore never exit.

This difficulty can be avoided if we bound the number of enabling attempts each exiting process makes. This is the idea behind Decker's algorithm, which bounds the number of attempts at $O(k)$. Intuitively, Decker observed that so long as each exiting process enables a corresponding k processes in order, all required properties are satisfied.

Our algorithm takes this solution one step further. Observe that if x processes have exited, then the first $x + k$ processes can safely be allowed to enter. We first match each exiting process with the latest trying process that can safely be enabled. Each exiting process then enables its partner, as well as the k trying processes preceding its partner.

Enabling the processes naively would still require $O(k)$ RMRs per exiting process. We make two novel improvements that allow us to reduce this to $O(\log k)$. First, we observe that under the CC model, a process is allowed to spin on more than one shared variable. Exiting processes can therefore modify certain shared variables in a pattern that enables the required k waiting processes, without performing k operations.

The second novelty is the scheme we use to match waiting and exiting processes together. We implement a two-dimensional array A with N slots, broken up into N/k blocks, each of k size. N is simply the minimal integer divisible by k and greater than or equal to n.

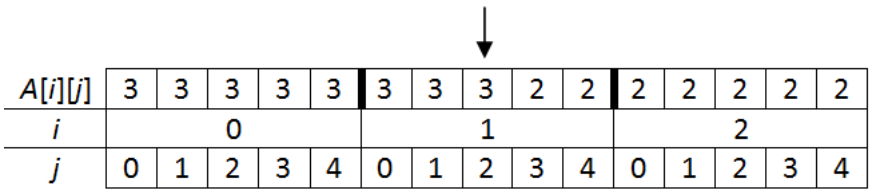

Figure 1: A, for $k = 5, N = 15$. $(2 \cdot 15) + 8 = 38$ processes have been enabled.

Entering processes receive successive tokens. To satisfy FIFE, every process with a lower token must be enabled earlier or simultaneously. Based on this token, each waiting process calculates its location in A and a value for which to wait. We define this calculation so that new processes are conceptually introduced from left to right; when wraparound occurs, the value waited for increases by 1. In figure 1, for example, the processes with the first 38 tokens are enabled.

However, as mentioned above, trying processes spin on more than one location in A. Once a waiting process calculates the location in A corresponding to its token - composed of a block index and a within-block index, the latter of which we will call w - it generates Wait-Set(w), and waits on each number in Wait-Set(w) within its assigned block. Similarly, each exiting process calculates

a Release-Set(r) (based on its partner's token).

Where binary length $= \lceil \log_2 k \rceil - 1$, and $BP(i)$ is the bit-pattern of i,
Wait-Set(w) $= \{w \leq b \leq k-1 \mid BP(b) = prefix(BP(w)) \cdot 10^*\} \cup \{k-1\}$
Release-Set(r) $= prefix(BP(r)) \cdot 0^*$

For any $w \leq r$, we require that $|\text{Wait-Set}(w) \cap \text{Release-Set}(r)| \geq 1$. Moreover, we require that $\forall w \leq w' \leq r : min(\text{Wait-Set}(w) \cap \text{Release-Set(r)}) \leq min(\text{Wait-Set}(w') \cap \text{Release-Set(r)})$. Both these properties are proved formally below: but the intuition is that the intersection point is based on the leftmost bit differing from r, which will be more significant for w' than for w.

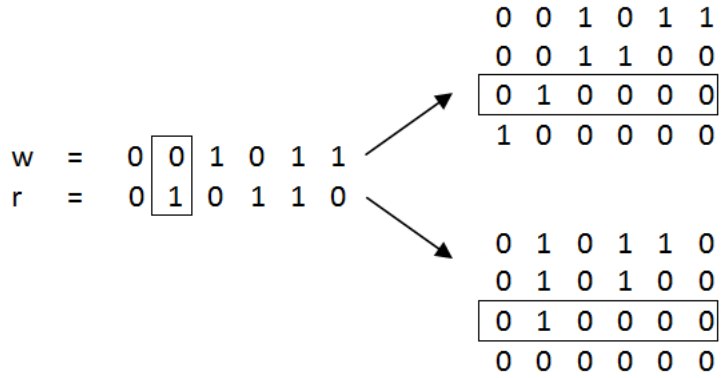

Figure 2: Wait-Set(w) and Release-Set(r), where $k = 33$. The element at which they intersect is determined by the leftmost bit at which they differ.

Using the same w and r, here is a diagram of Wait-Set(w) and Release-Set(r). Note that all elements in Wait-Set(w) are $\geq w$, and all elements in Release-Set(r) are $\leq r$.

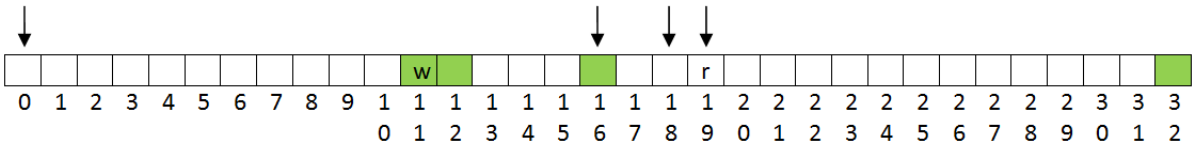

Figure 3: One block, where $k = 33$. The shaded squares indicate Wait-Set(w) and the arrows indicate Release-Set(r), where $w = 11$ and $r = 19$. The indices of A are numbered.

Each exiting process is guaranteed to enable its partner and every previous waiting process in the same block. If in addition to these, it updates the $k - 1^{th}$ entry in the previous block, it will have updated its partner and at least k processes prior to its partner, in the correct order.

Thus, our algorithm does the same work as prior algorithms, considerably more efficiently.

2.2 Variables

$N = min(\{m \geq n \mid m \ mod \ k = 0\})$

Shared Variables:
Entry $= 0$
Exit $= 0$
$A[0][0..k-1] = 1$
$A[1..(N/k)-1][0..k-1] = 0$

Local Variables:
$t = e = i = j = w = -1$
$j' = r = u = -1$
$t' = k - 1$

2.3 Routine

Main Routine: k-Exclusion

Remainder Section
1 $t \leftarrow$ F&I(Entry); $[e, i, j] \leftarrow parse(t)$
2 wait till $\exists w \in$ Wait-Set$(j) : A[i][w] \geq e$
Critical Section
3 $t' \leftarrow$ F&I(Exit) $+ k$; $j' \leftarrow t' \ mod \ k$
4 **foreach** ascending $r \in$ Release-Set$(j') \cup \{-1\}$ **do**
5 $update(t' - j' + r)$
6 $update(t' - j' + r)$

Subroutine: $update(u)$

$[e, i, j] \leftarrow parse(u)$
7 $a \leftarrow A[i][j]$
8 **if** (Exit $\leq u + 2k$) *and* $(a < e)$ **then**
9 CAS$(A[i][j], a, e)$

Subroutine: $parse(t)$

$[e, f] \leftarrow [\lfloor t/N \rfloor + 1, t \ \ mod \ \ N]$
$[i, j] \leftarrow [\lfloor f/k \rfloor, t \ mod \ k]$
return $[e, i, j]$

2.4 Definitions of Wait-Set and Release-Set

Where binary length $= \lceil \log_2 k \rceil - 1$,
Wait-Set$(j) = \{j \leq b \leq k-1 \mid BP(b) = prefix(BP(j)) \cdot 10^*\} \cup \{k-1\}$
Release-Set$(j) = prefix(BP(j)) \cdot 0^*$

3 Proof

3.1 Wait-Set and Release-Set Properties

$\forall h, i, j \in 0..k-1$:

(P1) Wait-Set(i), Release-Set(i) $\subseteq \{0..k-1\}$

(P2) $\forall w \in$ Wait-Set$(i), w \geq i$

(P3) $\forall r \in$ Release-Set$(i), r \leq i$

(P4) $i \leq j \Rightarrow$ Wait-Set$(i) \cap$ Release-Set$(j) \neq \emptyset$

(P5) $h \leq i \leq j \Rightarrow \min($Wait-Set$(h) \cap$ Release-Set$(j)) \leq \min($Wait-Set$(i) \cap$ Release-Set$(j))$

(P6) $|$Wait-Set$(i)| = O(\log k)$ and $|$Release-Set$(i)| = O(\log k)$

3.2 Proof of Set Properties

3.2.1 Property 1

Proof Wait-Set and Release-Set clearly satisfy **(P1)**. By definition, $\forall w \in$ Wait-Set$(i) : 0 \leq w \leq i \leq k-1$. Similarly, $\forall r \in$ Release-Set$(i) : 0 \leq r \leq i \leq k-1$. So Wait-Set($i$), Release-Set($i$) $\subseteq 0..k-1$.

3.2.2 Properties 2 and 3

Proof The restriction that $\forall w \in$ Wait-Set$(i), w \geq i$ is built into the definition of Wait-Set (noting that $i \leq k-1$). The restriction that $\forall r \in$ Release-Set$(i), r \leq i$ follows from the fact that a prefix of i is being concatenated with 0s - thus any element in Release-Set(i) is $\leq i$.

3.2.3 Lemma 1

Lemma 1 $\forall i, j \in 0..k-1, i < j : |$Wait-Set$(i) \cap$ Release-Set$(j)| = 1$

Proof
Some notation:
$i_1..i_{\lceil logk \rceil} =$ BP(i)
$j_1..j_{\lceil logk \rceil} =$ BP(j)

Given $i < j$, then $\exists x \in 0..k-1 : i_x \neq r_x$. Consider the smallest such x (corresponding to the most significant bits in BP(i) and BP(j)), which we will refer to as y. If $i_y = 1$ and $j_y = 0$, then $i > j$; this is contradictory, so $i_y = 0$ and $j_y = 1$.

By the definitions of Wait-Set and Release-Set,

$i_1..i_{y-1}10^{k-y-1} \in$ Wait-Set(i)
and
$j_1..j_y0^{k-y-1} \in$ Release-Set(j)

Because y is the smallest x such that $i_x \neq j_x$,

$i_1 i_2 ... i_{y-1} = j_1 j_2 ... j_{y-1}$

Because $j_y = 1$,

$j_y 0^{k-y-1} = 10^{k-y-1}$

Thus for $s = i_1..i_{y-1}10^{k-y-1} = j_1..j_y 0^{k-y-1}$, $s \in$ Wait-Set(i) and $s \in$ Release-Set(j).

Significantly for the proof of property 5, s represents a unique member of Wait-Set(i) and Release(j) that is in both sets. To see this, consider bit positions other than y. $\forall x < y, i_x = j_x$. Consequently, $i_1..i_{x-1}10^{k-x-1} > j_1..j_x 0^{k-x-1}$. $\forall x > y, i_1..i_{x-1}10^{k-x-1} < j_1..j_x 0^{k-x-1}$, since the two will differ at bit y (which is of greater significance than all subsequent bits). The only possible element of Wait-Set that does not fit this definition is $k-1$. However, Release-Set(j) contains $k-1$ iff $j = k-1$ by **(P3)**. So in all cases, s is unique. $\therefore |\text{Wait-Set}(i) \cap \text{Release-Set}(j)| = 1$.

3.2.4 Property 4

Proof Assume $i \leq j$. To prove: Wait-Set(i) $\cap$ Release-Set(j) $\neq 0$.

We will consider two cases exhausting all possibilities: either $i < j$, or $i = j$. If $i < j$, then by Lemma 1, $|\text{Wait-Set}(i) \cap \text{Release-Set}(j)| = 1 \neq 0$. Now consider the case where $i = j$. Because $i \in$ Wait-Set(i) and $j \in$ Release-Set(j), $i = j \in$ Wait-Set(i) and $i = j \in$ Release-Set(j). Then in all cases, Wait-Set(i) $\cap$ Release-Set(j) $\neq 0$.

3.2.5 Property 5

Proof Assume that $\exists h, i, j \in 0..k-1 : h \leq i \leq j$. To prove: min(Wait-Set(h) $\cap$ Release-Set(j)) $\leq$ min(Wait-Set(i) $\cap$ Release-Set(j)).

As with property 2, we will consider two cases. Either $h = i$ or $h < i$. If $h = i$, then Wait-Set(h) = Wait-Set(i) (since elements in Wait-Set are deterministic), and the proof of the case is complete. If $h < i$, then we use Lemma 1. Consider $s_h = j_1..j_y 0^{k-y-1}$ and $s_i = j_1..j_z 0^{k-z-1}$ as defined in Lemma 1 for some y and z.

Assume for contradiction that $s_h > s_i$. Then $y > z$ (since the y^{th} and z^{th} bits are followed by 0*). Noting from the lemma that $h_y = i_z = 0$, $j_1..jz \in prefix(h)$ and $j_1..j_{z-1}0 \in prefix(i)$. This implies that $h > j$, which contradicts our starting assumption. So by contradiction, $s_h < s_i$. Because s_h, s_i are the unique shared elements between Wait-Set(h), Wait-Set(i) and Release-Set(j), they are also the minimal elements in the intersection, and this case is proven.

Then in all cases, min(Wait-Set(h) $\cap$ Release-Set(j)) $\leq$ min(Wait-Set(i) $\cap$ Release-Set(j)).

3.2.6 Property 6

$\forall i$: |Wait-Set(i)| = # of distinct prefixes of $i+1$, |Release-Set(i)| = # of distinct prefixes of i. Because $0 \leq i \leq k-1$, # of distinct prefixes of $i = O(\log k)$. So both Wait-Set and Release-Set are $O(\log k)$.

3.3 Definitions

$En(t)$ = true iff $\exists w \in$ Wait-Set(t) : $A[i][w] \geq e$, where $[e, i, j] \leftarrow parse(t)$

Update-Set(t') = $\{u = t' - j' + r \mid r \in \text{Release-Set}(j') \cup \{-1\}\}$, where $j' = t' \ mod \ k$

3.4 Observations

In what follows, p and q denote processes, i and j denote indices in A, and e denotes values in A. x_p denotes the value of variable x for process p. PC (program counter) indicates the line of the algorithm that the process is about to evaluate. $\langle a, b\rangle$ indicates the set of all processes with PC $\in a..b$. Thus $\langle 1, 1\rangle$ = set of processes in the remainder section, $\langle 2, 2\rangle$ = set of processes in the try section, $\langle 3, 3\rangle$ = set of processes in the critical section, and $\langle 4, 6\rangle$ = set of processes in the exit section.

Lines are numbered in the algorithm above. The *update* subroutine executed in **5** is renumbered **5.7**-**5.9**, while the *update* subroutine executed in **6** is renumbered **6.7**-**6.9**. However, lines **7**-**9** are still referred to directly; $\text{PC}_p = \mathbf{9} \equiv (\text{PC}_p = \mathbf{5.9} \vee \text{PC}_p = \mathbf{6.9})$.

(O1) Entry strictly increases by increments of 1 (on **1**).

(O2) Exit strictly increases by increments of 1 (on **3**).

(O3) $\forall i, j : A[i][j]$ strictly increases on a successful CAS on **9**.

(O4) $|\langle \mathbf{3}, \mathbf{3}\rangle| \leq |\{t \mid En(t)\}| - \text{Exit}$

(O5) $max(\{t' \geq -1 \mid \exists p : t' = t'_p\}) = \text{Exit} + k - 1$

(O6) $\forall p \neq q : t_p \neq t_q$

(O7) $\forall t \geq 0 : En(t)$ never goes from true to false.

3.5 Lemmas

The following lemmas will assist the proofs of our invariants.

Lemma 2 $\forall [e, i, j] \leftarrow parse(t) : t = ((e-1) \times N) + (i \times k) + j$

Proof

$$\begin{array}{lll} & ((e-1) \times N) + (i \times k) + j & \\ = & ((\lfloor t/N \rfloor + 1) - 1) \times N + \lfloor (t \bmod N)/k \rfloor \times k + t \bmod k & \text{by the definition of } parse \\ = & \lfloor t/N \rfloor \times N + t \bmod N & \text{by the properties of } mod \\ = & t & \text{by the properties of } mod \end{array}$$

Lemma 3 $\forall u \in \text{Update-Set}(t') : u \leq t'$

Proof $\forall r \in \text{Release-Set}(j') : r \leq j'$, by **(P3)**. For $r = -1, r < 0 \leq j'$ trivially.
$\therefore -j' + r \leq 0$
$\therefore u = t' - j' + r \leq t'$

Lemma 4 $\forall p : p$ is enabled iff $En(t_p)$.

Proof By the definition of enabled, process p is enabled if and only if it will enter the CS in a finite number of its own steps, regardless of the steps taken by other processes. The only point in the Entry section that a process waits is **2**, where it waits until $\exists w \in \text{Wait-Set}(j_p) : A[i][w] \geq e$. At 2, $En(t_p)$ is true iff $\exists w \in \text{Wait-Set}(j_p) : A[i_p][w] \geq e_p$. Since A strictly increases (by **(O3)**), any process for which $En(t_p)$ is true will enter the CS immediately after reading A, regardless of the steps taken by other processes. So the lemma holds.

3.6 Invariants

To preserve logical consistency, invariants with lower numbers are strictly used to prove invariants with higher numbers, and never the reverse.

(I1) Entry $= |\langle \mathbf{2,\ 3} \rangle| + \text{Exit}$

(I2) $\forall t \geq 0 : En(t) \Rightarrow t < \text{Exit} + k$

(I3) $|\langle \mathbf{3,\ 3} \rangle| \leq k$

(I4) $\forall t \geq 0 : t < Exit \Rightarrow En(t)$

(I5) $(\text{PC}_p = \mathbf{6.8} \wedge \text{Exit}) \Rightarrow A[i_p][k-1] \geq e_p$

(I6) $p \in \langle \mathbf{6.7,\ 6.9} \rangle : \forall q : (\text{PC}_q = \mathbf{9} \wedge i_p = i_q \wedge j_p = j_q \wedge a_q = A[i_p][j_p]) \Rightarrow e_q \geq e_p$

(I7) $p \in \langle \mathbf{6.8,\ 6.9} \rangle : A[i_p][j_p] = a_p \vee A[i_p][j_p] \geq e_p$

(I8) $(p \in \langle \mathbf{4,\ 1} \rangle \wedge t' \neq -1) \Rightarrow \forall [e,i,j] = parse(u)$ for $u \in$ Update-Set$(t'_p) \wedge u < u_p : A[i][j] \geq e \vee A[i][k-1] \geq e$

(I9) $(\text{PC}_p = \mathbf{1} \wedge t'_p \neq -1) \Rightarrow \forall [e,i,j] = parse(t')$ where $t \geq 0 : A[i][j] \geq e \vee A[i][k-1] \geq e$

(I10) $(\text{PC}_p = \mathbf{1} \wedge t'_p \neq -1) \Rightarrow \forall t'_p - k \leq t \leq t'_p : En(t)$

(I11) $\forall 0 \leq t < t' : En(t') \Rightarrow En(t)$

3.6.1 Direct Proof

Some of the above invariants can be proven as direct logical consequences of preceding invariants.

(I3) *Proof*

$$\begin{array}{rrcll} & |\langle \mathbf{3},\ \mathbf{3} \rangle| & \leq & |\{t \mid En(t)\}| - \text{Exit} & \text{by } \mathbf{(O4)} \\ & |\{t \mid En(t)\}| & \leq & \text{Exit} + k & \text{by } \mathbf{(I2)} \\ \therefore & |\langle \mathbf{3},\ \mathbf{3} \rangle| & \leq & \text{Exit} + k - \text{Exit} & \\ & & = & k & \end{array}$$

(I10) *Proof* Assume $(\text{PC}_p = 1 \wedge t'p \neq 1)$. To prove: $\forall t'_p - k \leq t \leq t'_p : En(t)$.

Since when $\text{PC}_p = 1 \wedge t'p \neq 1$, $u_p = t'_p$, we can combine **(I8)** and **(I9)** to yield $\forall [e,i,j] = parse(u)$ for $u \in$ Update-Set$(t'_p) : A[i][j] \geq e \vee A[i][k-1] \geq e$. Consider two sequences: $S_{prev} = [t'_p - j'_p - k..t'_p - j'_p - 1]$ and $S_{cur} = [t'_p - j'_p..t'_p]$. By the definition of j'_p, for $[e,i,j] = parse(t'_p - j'_p), j = 0$. Then by Lemma 2, S_{prev} renders $A[i'_p - 1][0..k-1]$ and S_{cur} renders $A[i'_p][0..j'_p]$.

We will first prove $\forall s_{prev} \in S_{prev} : En(s_{prev})$. Set $u = t'_p - j'_p - 1 \in$ Update-Set(t'_p). Then $j = k - 1$, so $A[i][k-1] = A[i'_p][k-1] \geq e$. So $\forall s_{prev} \in S_{prev} : En(s_{prev})$.

Now we will prove $\forall s_{cur} \in S_{cur} : En(s_{cur})$. By **(P4)**, every s_{cur} will intersect with an element in Update-Set(t'_p). For each such intersecting element u, $A[i][j] \geq e \vee A[i][k-1] \geq e$ where $[e,i,j] = parse(u)$. If $A[i][k-1] \geq e$, then $En(s_{cur})$ is true for all s_{cur}, since $k - 1 \in$

$Wait - Set(j)$ by definition. If $A[i][k-1] < e$, then $A[i][j] \geq e$ for each u, and $En(s_{cur})$ is still true for all s_{cur}. Then in all cases, $\forall s_{cur} \in S_{cur} : En(s_{cur})$.

Thus $\forall t \in t'_p - j'_p - k..t'_p : En(t)$. Since $j'_p \geq 0$, we have proven the consequent.

3.6.2 Proof by Induction

An inductive proof of the correctness of our invariants follows. The proof operates for each possible step that an arbitrary process p can take. Inductively, we assume that all of the invariants hold for p prior to its step (when $\text{PC}_p = a$) and prove that all of the invariants still hold after the step (when $\text{PC}_p = b$). We will refer to the configuration when $\text{PC}_p = a$ as $\mathcal{C}$, and the configuration when $\text{PC}_p = b$ as $\mathcal{C}$.s.

Initial Configuration First, we demonstrate that all invariants hold at initialization.

- **(I1)** Entry = Exit = 0, and since no process has executed a step, $|\langle \mathbf{2,\ 3} \rangle| = 0$. So the equality holds.
- **(I2)** Exit and A are unchanged.
- **(I4)** Exit and A are unchanged.
- **(I5)-(I7)** Trivially true.
- **(I8)-(I9)** Trivially true, since $t' = -1$ for all t' at initialization.
- **(I11)** $En(t)$ is true for $t \in 0..k-1$ and false for all other t, so the invariant holds.

1 → 2

- **(I1)** Entry and $|\langle \mathbf{2,\ 3} \rangle|$ both increase by 1, so the equality holds.
- **(I2)**,**(I4)** Exit and A are unchanged.
- **(I5)-(I9)** Trivially true.
- **(I11)** A is unchanged.

2 → 3

- **(I1)** Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.
- **(I2)**,**(I4)** Exit and A are unchanged.
- **(I5)-(I9)** Trivially true.
- **(I11)** A is unchanged.

3 → 4

- **(I1)** Exit increases by 1, but $|\langle \mathbf{2,\ 3} \rangle|$ decreases by 1, so $|\langle \mathbf{2,\ 3} \rangle| + \text{Exit}$ is unchanged, and the equality holds.
- **(I2)** Exit increases by 1; if $\forall t \geq 0 : En(t) \Rightarrow t < \text{Exit} + k$ before this step, then trivially $t < \text{Exit} + k + 1$ now.
- **(I4)** By the inductive hypothesis, $\forall t \geq 0 : t \leq \text{Exit} - 2 \Rightarrow En(t)$. The border case is t such that $t = \text{Exit} - 1$; however, the process with such t is p. $En(t_p)$ must have been true when p stepped **2 → 3**, and by **(O7)**, $En(t_p)$ will never go from true to false. $\therefore En(t_p)$ is true and the invariant holds.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) At $\mathcal{C}$.s, $u_p = t'_p - 1 = min(\text{Update-Set}(t'_p)$, and the invariant trivially holds.

(I11) A is unchanged.

4 → 5.7

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) The first ascending $u \in \text{Update-Set}(u) = t'_p - 1 = min(\text{Update-Set}(t'_p)$, so the invariant trivially holds.

(I11) A is unchanged.

5.7 → 5.8

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) t'_p, u_p and A are unchanged.

(I11) A is unchanged.

5.8 → 5.9

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) t'_p, u_p and A are unchanged.

(I11) A is unchanged.

5.8 → 6.7

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) t'_p, u_p and A are unchanged.

(I11) A is unchanged.

5.9 → 6.7

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2) In the update subroutine, $u_p \in \text{Release-Set}(j') \cup \{-1\}$ from line **4**. By Lemma 3, $u_p \leq t'_p$ for all such u. $\forall t \geq 0$: if $\neg En(t)$ in $\mathcal{C}$ and $En(t)$ in $\mathcal{C}$.s, then $e_p \geq e$, $i_p = i$ and $j_p \in \text{Wait-Set}(j)$ for $[e, i, j] = parse(t)$. Because $\forall w \in \text{Wait-Set}(j) : w \geq j$ by **(P2)**, applying Lemma 2 shows that $u \geq t$. Then $t \leq u \leq t'_p < Exit + k$ (by **(O5)**, and the invariant holds.

(I4) By **(O3)**, this step results only in $En(t)$ going from false to true; thus, for any $t \geq 0$ where the invariant held before, it still holds.

(I5) Trivially true.

(I6) **5.9** $\rightarrow$ **6.7** results in the CAS returning either *true* (upon success) or *false* (upon failure). Given arbitrary q such that $\text{PC}_q = \mathbf{9} \wedge i_q = i_p \wedge j_q = j_p \wedge a_q = A[i_q][j_q]$, we will address both cases:

- If the CAS succeeds, then $A[i_p][j_p] \geq e_p$. Then since $A[i_p][j_p]$ is strictly increasing, any process poised to CAS successfully must have $e_q > A[i_p][j_p] \geq e_p$.
- If CAS fails, then $A[i_p][j_p] \neq a_p$. Because $A[i_p][j_p]$ strictly increases, $a_q = A[i_p][j_p] > a_p$. Assume for contradiction that $e_q < e_p$. $a_q > a_p$ implies step **7** $\rightarrow$ **8** for q occurred after **5.7** $\rightarrow$ **5.8** for p. By Lemma 3 and **(O5)**, $u_p \leq t'_p <$ Exit when $\text{PC}_p = \mathbf{5.7}$, so since Exit strictly increases, $u_p <$ Exit when $q \in \langle 8, 9 \rangle$. Because $i_p = i_q \wedge j_p = j_q \wedge e_p > e_q$, by Lemma 2, $u_q \leq u_p - N$. Then $u_q \leq u_p - N <$ Exit $- N$ when $\text{PC}_q = \mathbf{8}$. **8** $\rightarrow$ **9** cannot have happened, since Exit $> u_q + 2k$ when $\text{PC}_q = \mathbf{8}$. Then $\text{PC}_q \neq \mathbf{9}$, which is contradictory. So $e_q \geq e_p$ by contradiction.

Thus $e_q \geq e_p$ in all cases, and the invariant holds.

(I7) Trivially true.

(I8) t'_p, u_p are unchanged. $A[i_p][j_p]$ may have changed, but strictly increases; so since the invariant held before, it must also hold now.

(I9) Trivially true.

(I11) Thanks to the inductive hypothesis, we know that the invariant holds for all t' such that $En(t')$ is true in $\mathcal{C}$. We define $t_{max} = max(\{t | En(t)\})$ in $\mathcal{C}$. Thus we only need to prove that $\forall t > t_{max} : En(t)$, since by the inductive hypothesis all smaller t have $En(t)$ true in $\mathcal{C}$, and $En(t)$ never goes from true to false. By **(I4)**, Exit $- 1 \leq t_{max}$ in $\mathcal{C}$; because Exit does not change in this step, this is true of $\mathcal{C}$.s as well. $t' <$ Exit $+ 2k$ by **(I2)**, meaning $t' \leq t_{max} + 2k$ in $\mathcal{C}$.s.

By **(P2)**, t' is at most u_p in $\mathcal{C}$.s. (We will subtext all of the local variables of p in $\mathcal{C}$ with *old*, e.g. $t' \leq u_{old}$). Thus we need to demonstrate that given any $u \in$ Update-Set(t'_p), $\forall u_{old} - k \leq t_{max} < t \leq u_{old} : En(t)$.

If $u_{old} = t'_p - j'_p - 1$, then the CAS is to $A[i'_p - 1][k - 1]$, and $En(t)$ is true for the preceding $k - 1$ processes, which all have $k - 1$ in their Wait-Set. If $u_{old} \geq t'_p - j'_p$, we apply **(I8)**. Wait-Set(t) and Release-Set(j'_p) intersect for all $t < j'_p$ by **(P4)**. Moreover, the point of intersection for lower $t \leq$ point of intersection for higher t, by **(P5)**. These properties along with **(I8)** imply that either $A[i_{old}][k-1] \geq e_{old}$, in which case $En(t)$ for all t in our range, or for every t there is $j \in 0..k-1$ such that $A[i_{old}][j] \geq e_{old}$ and j is in the Wait-Set associated with t. So in all cases, $En(t)$ for arbitrary $u_{old} - k \leq t \leq u_{old}$.

6.7 $\rightarrow$ **6.8**

(I1) Entry, Exit and $|\langle \mathbf{2, 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5) Consider $t_{en} = t'_p - j'_p + k - 1$.
$En(t_{en})$ iff $A[i_p][k-1] \geq e_p$, by Lemma 2 and since $\{k-1\} = \text{Wait-Set}(k-1)$.
Since $\forall r \in \text{Release-Set} : r \leq k-1$ (by **(P1)**, $t'_p \leq u_p + k$. Therefore $t_{en} \leq u_p + k - j'_p + k - 1 < u_p + 2k < \text{Exit}$.
$t_{en} < \text{Exit} \Rightarrow En(t_{en})$ by **(I4)**, implying $A[i_p][k-1] \geq e_p$. So the invariant holds.

(I6) $A[i_p][j_p]$, i_p, j_p and e_p are unchanged.

(I7) After the atomic step **6.7** $\rightarrow$ **6.8**, $A[i_p][j_p] = a_p$, so the invariant holds.

(I8) t'_p, u_p and A are unchanged.

(I9) Trivially true.

(I11) A is unchanged.

6.8 $\rightarrow$ **6.9**

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5) Trivially true.

(I6) $A[i_p][j_p]$, i_p, j_p and e_p are unchanged.

(I7) If after this step $A[i_p][j_p] \neq a_p$, then earlier in the execution, while $p \in \langle 6.7, 6.9 \rangle$, $\exists q : \text{PC}_q = \mathbf{9} \wedge i_p = i_q \wedge j_p = j_q \wedge a_q = A[i_p][j_p]$. By **(I6)**, $e_q \geq e_p$ for all such q. So if $A[i_p][j_p] \neq a_p$, $A[i_p][j_p] \geq e_p$. This is logically equivalent to the invariant, so it holds.

(I8) t'_p, u_p and A are unchanged.

(I9) Trivially true.

(I11) A is unchanged.

6.8 $\rightarrow$ **5.7**

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) By the inductive hypothesis, we already know that in $\mathcal{C}$ the invariant holds for $u \in \text{Update-Set}(t'_p) < u_p$. Because u_p in $\mathcal{C}$.s is the next ascending u after $u_{old} = u_p$ in $\mathcal{C}$, we need to demonstrate the consequent now holds for u_{old}. Note that this step is taken iff $(\text{Exit} > u_p + k) \vee (a \geq e)$ in $\mathcal{C}$. In the first case, by **(I5)**, $A[i_p][k-1] \geq e_p$ in $\mathcal{C}$. Because $A[i_p][k-1]$ strictly increases, $[e_{old}, i_{old}, j_{old}] = parse(u_{old})$ gives us $A[i_{old}][j_{old}] \geq e_{old}$. In the second case, the invariant holds directly, so it holds in all cases in $\mathcal{C}$.s.

(I11) A is unchanged.

6.8 $\rightarrow$ **1**

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Exit and A are unchanged.

(I5)-**(I7)** Trivially true.

(I8) t'_p, r_p and A are unchanged.

(I9) Analogous to the proof of **(I8)** for **6.8** $\rightarrow$ **5.7**. u_p is the same in $\mathcal{C}$ and $\mathcal{C}$.s, but this does not alter the proof.

(I11) A is unchanged.

6.9 $\rightarrow$ **5.7**

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Proof analogous to step **5.9** $\rightarrow$ **6.7**.

(I5)-**(I7)**,**(I9)** Trivially true.

(I8) Analogous to **6.8** $\rightarrow$ **5.7**, we only need to demonstrate the invariant holds for $[e_{old}, i_{old}, j_{old}] = parse(u_p)$ in $\mathcal{C}$. By **(I7)**, $A[i_p][j_p] = a_p \vee A[i_p][j_p] \geq e_p$ in $\mathcal{C}$. In the first case, the CAS succeeds and $A[i_{old}][j_{old}] = e_{old}$. In the second case, $A[i_{old}][k-1] \geq e_{old}$. Either way, the invariant is satisfied in $\mathcal{C}$.s.

(I11) Proof analogous to step **5.9** $\rightarrow$ **6.7**.

6.9 $\rightarrow$ **1**

(I1) Entry, Exit and $|\langle \mathbf{2,\ 3} \rangle|$ are unchanged.

(I2),**(I4)** Proof analogous to step **5.9** $\rightarrow$ **6.7**.

(I5)-**(I7)** Trivially true.

(I8) t'_p, r_p and A are unchanged.

(I9) Analogous to the proof of **(I8)** for **6.9** $\rightarrow$ **5.7**. Note that u_p is the same in $\mathcal{C}$ and $\mathcal{C}$.s. By **(I7)**, $A[i_p][j_p] = a_p \vee A[i_p][j_p] \geq e_p$ in $\mathcal{C}$. In the first case, the CAS succeeds and $A[i_p][j_p] = e_p$. In the second case, $A[i_p][k-1] \geq e_p$. Either way, the invariant is satisfied in $\mathcal{C}$.s.

(I11) Proof analogous to step **5.9** $\rightarrow$ **6.7**.

3.7 Proof of Desired Properties

3.7.1 k-Exclusion

Proof **(I3)** directly implies k-Exclusion.

3.7.2 Bounded Exit

Proof The *update* subroutine requires O(1) RMRs (and O(1) computation in general). Line **3** thus require O(1) steps. Lines **4-6** require O(1) $\cdot$ O(|Release-Set(q')| + 1) steps. The whole Exit section therefore requires O(|Release-Set(q')|) steps. |Release-Set| is $O(k)$ by axiom 1, so Bounded Exit is satisfied.

3.7.3 First-In First-Enabled

(I11) directly implies FIFE.

3.7.4 Starvation Freedom

Proof Assume for contradiction that in some configuration, fewer than $k-1$ processes have crashed, but there are processes in the try section that will never be enabled. We have already proven bounded exit; therefore, in an infinite run, there will be a point beyond which the only processes in the exit section are crashed and the only processes in the entry section are crashed or not enabled. Select an arbitrary configuration $\mathcal{C}$ beyond this point. We designate the number of processes crashed in the entry section $crash_n$ and the number of processes crashed in the exit $crash_x$.

Consider the highest t' such that no process p with t'_p is in the exit section. Designate this value $t'_x \geq \text{Exit} + k - 1 - crash_x$ by **(O5)**. By **(I10)** and **(I11)**, we know that all processes with $t < En(t_x)$ are enabled. Further consider the uncrashed process with lowest t in the try section. Designate this $t_n \leq \text{Exit} + crash_n$.

$$\begin{aligned} t_n - t'_x &\leq \text{Exit} + crash_n - (\text{Exit} + k - 1 - crash_x) \\ &= (crash_n + crash_x) - (k-1) \\ &\leq 0 \qquad \text{since total crashes} \leq k-1 \\ \therefore\ t_n &\leq t'_x \end{aligned}$$

Thus the process with t_n is enabled, uncrashed and in the try section. This is contradictory, since the only processes in the try section are crashed or not enabled. Thus by contradiction, starvation with fewer than $k-1$ crashes is impossible.

3.7.5 O(log$k$) RMR Complexity

With Smart Cache

Proof If we do not count CAS failures as incurring RMRs, time complexity is straightforward. The *parse* subroutine is local and therefore makes no RMR; the *update* subroutines costs O(1) RMR. Likewise, the time complexity of **1** and **3** is O(1). **2** requires $O(\log k)$ to initialize over $O(\log k)$ objects (by **(P6)**), and O(1) subsequently, because we assume that CAS failure does not incur RMRs. The loop in **4-6** runs $O(\log k)$ times by **(P6)**, and requires O(1) RMR per iteration from above. Thus total RMR cost is O(1) + $O(\log k)$ + $O(\log k)$ = $O(\log k)$.

Without Smart Cache

Proof We will prove that including the cost of failed CAS writes on readers results in O(1) additional amortized RMR if we increase N. Specifically, we set $N = kn/logk$.

First, note that not all CAS failures cause additional RMRs. Only when some process p is waiting for $A[i][j] \geq e$, and CAS$(A[i][j], a, e')$ occurs for a and $e' < e$ is RMR cost incurred. Because $N = kn/logk$, it is impossible to advance a full round without Exit incrementing at least $(k-1)N/logk$ times. Thus when an exiting process executes a failed CAS that causes RMRs, it may execute at most one (after which it will fail to satisfy the if statement at **8**).

Each such failed CAS is waited upon by $\leq k$ trying processes. If some p is waiting on $A[i][j]$ for e, then all other processes waiting on $A[i][j]$ are either enabled or waiting for $e' \geq e$. This is also due to the advancement required for a process to spin on $A[i][j]$ with $e > e'$. The required $(k-1)N/logk$ increments to Exit along with **(I4)** imply that any process with smaller e is enabled.

Thus, since each process can cause at most 1 failed CAS affecting at most k trying processes, and since $< n$ processes may remain in the exit section if progress continues, each round through A

results in $< nk$ RMRs from bad CAS. Since each round involves $nk/logk$ processes, the amortized additional cost is $O(\log k)$.

3.7.6 Space Complexity

Proof The space complexity of shared variables in our algorithm is the size of $A + 2$. With Smart Cache, the size of A $\leq n + k - 1$, since N is the smallest integer $\geq n$ such that $N \ mod \ k = 0$ and $n > k$. So the worst-case total space complexity is $n + k + 1 = \mathrm{O}(n)$. Without Smart Cache, as discussed above, the worst-case total space complexity will be $(nk/logk) + k + 1 = \mathrm{O}(nk/logk)$.

3.8 Model Checking

In addition to the invariant proof, the algorithm in this paper has been checked using Leslie Lamport's Temporal Logic of Actions+ (TLA+) language [6], and the associated TLA+ Model Checker (TLC) [7]. The algorithm has been tested for deadlock, k-Exclusion and the FIFE property on systems consisting of ≤ 6 processes with $k \leq 3$. More than 58 million distinct states have been checked without violation of any of these properties.

It should be noted that TLC does not check every possible configuration even for a bounded number of processes, and that TLC checking therefore does not prove correctness. Moreover, starvation freedom cannot be tested using TLC. However, model checking provides reassurance of correctness without requiring the reader to address the complexities of invariant-based proof.

4 Further Research

The variables used in our algorithm are unbounded; a straightforward improvement would be to bound the variables. Improvements in the time complexity of k-Exclusion may be possible as well. In particular, modifying Wait-Set and Release-Set might result in $o(\log k)$ worst-case RMR complexity, although no such modifications are evident. Intuition suggests that $\Omega(\log k)$ is the lower bound for k-Exclusion. $\omega(1)$ or $\Omega(logk)$ RMR lower bound proofs would help confirm this.

5 Acknowledgements

This thesis was made possible by the thoughtful collaboration of Jack Bowman, Michael Diamond, Matthew Elkherj, Zhiyu Liu and Nancy Zheng. Thanks to Lilai Guo for her support and assistance in proofreading. Most of all, thanks to Prasad Jayanti, whose inspiring teaching, excellent feedback and untrammelled enthusiasm for knowledge brought this research to life.

References

[1] Michael J. Fischer, Nancy A. Lynch, James E. Burns, and Allan Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *FOCS'79*, pages 234–254, 1979.

[2] James H. Anderson and Mark Moir. Using local-spin k-exclusion algorithms to improve wait-free object implementations. *Distributed Computing*, 11:141–150, 1997.

[3] Robert Danek. The k-bakery: local-spin k-exclusion using non-atomic reads and writes. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 36–44, New York, NY, USA, 2010. ACM.

[4] Chase Decker and Prasad Jayanti. A Solution to k-Assignment in $O(k)$ RMR Complexity. Dartmouth College, 2010.

[5] Robert Danek and Vassos Hadzilacos. Local-spin group mutual exclusion algorithms. In *DISC'04*, pages 71–85, 2004.

[6] Leslie Lamport. Introduction to TLA. Technical Report SRC-TN-1994-001, HP Labs, 1994.

[7] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2003.