Wait-Free Algorithms for Fast, Long-Lived Renaming*

Mark Moir and James H. Anderson

Department of Computer Science
The University of North Carolina at Chapel Hill
Chapel Hill, North Carolina 27599-3175, USA

August 1994

Abstract

We consider wait-free solutions to the renaming problem for shared-memory multiprocessing systems [3, 5]. In the renaming problem, processes are required to choose new names in order to reduce the size of their name space. Previous solutions to the renaming problem have time complexity that is dependent on the size of the original name space, and allow processes to acquire names only once. In this paper, we present several new renaming algorithms. Most of our algorithms have time complexity that is independent of the size of the original name space, and some of our algorithms solve a new, more general version of the renaming problem called long-lived renaming. In long-lived renaming algorithms, processes may repeatedly acquire and release names.

1 Introduction

In the *M-renaming* problem [2], each of k processes is required to choose a distinct value, called a *name*, that ranges over $\{0, ..., M-1\}$. Each process is assumed to have a unique process identifier ranging over $\{0..N-1\}$. It is further required that $k \leq M < N$. Thus, an M-renaming algorithm is invoked by k processes in order to reduce the size of their name space from N to M .

Renaming is useful when processes perform a computation whose time complexity is dependent on the size of the name space containing the processes. By first using an efficient renaming algorithm to reduce the size of the name space, the time complexity of that computation can be made independent of the size of the original name space.

The renaming problem has been studied previously for both message-passing [2] and shared-memory multiprocessing systems [3, 5]. In this paper, we consider wait-free implementations of renaming in asynchronous, shared-memory systems. A renaming algorithm is *wait-free* iff each process is guaranteed to acquire a name after a finite number of that process's steps, regardless of the execution speeds of other processes.

Previous research on the renaming problem has focused on *one-time* renaming: each process acquires a name only once. In this paper, we also consider *long-lived* renaming, a new, more general version of renaming in which processes may repeatedly acquire and release names.

A solution to the long-lived renaming problem is useful in settings in which processes repeatedly access identical resources. The specific application that motivated us to study this problem is the implementation of shared objects. The complexity of a shared object implementation is often dependent on the size of the name space containing the processes that access that implementation. For such implementations, performance can

*Work supported, in part, by NSF Contract CCR-9216421. Authors' e-mail addresses: {moir,anderson}@cs.unc.edu. Preliminary version to be presented at the Eighth International Workshop on Distributed Algorithms, Terschelling, The Netherlands, September, 1994.

Reference	M	Time Complexity	Long-Lived?
[3]	$k(k+1)/2$	$\Theta(Nk)$	No
[3]	$2k-1$	$\Theta(N4^k)$	No
[5]	$2k-1$	$\Theta(Nk^2)$	No
Thm. 1	$k(k+1)/2$	$\Theta(k)$	No
Thm. 2	$2k-1$	$\Theta(k^4)$	No
Thm. 3	$k(k+1)/2$	$\Theta(Nk)$	Yes

Table 1: A comparison of wait-free M-renaming algorithms that employ only atomic reads and writes.

be improved by restricting the number of processes that concurrently access the implementation, and by using long-lived renaming to acquire a name from a reduced name space. This is the essence of an approach for the implementation of resilient, scalable shared objects presented by us in [1]. Note that this approach only restricts the number of processes that access the implementation *concurrently*. Over time, many processes may access the implementation. Thus, it is not sufficient to simply acquire a name once and retain that name for future use: a process must be able to release its name so that another process may later acquire the same name. In [1], we presented a simple long-lived renaming algorithm. To our knowledge, this is the only previous work on long-lived renaming. In this paper, we present several new long-lived renaming algorithms, one of which is a generalization of the algorithm we presented in [1].

In the first part of the paper, we present three renaming algorithms that use only atomic read and write instructions. It has been shown that if $M < 2k-1$, then M-renaming cannot be implemented in a wait-free manner using only atomic reads and writes [7]. Some of the previous wait-free, read/write algorithms for one-time renaming [3, 5] yield an optimal name space of $M = 2k-1$. However, in all of these algorithms, the time complexity of choosing a name is dependent on N. Thus, these algorithms suffer from the same shortcoming that the renaming problem is intended to overcome, namely time complexity that is dependent on the size of the original name space.

We consider one-time and long-lived renaming using reads and writes. We present two read/write algorithms for one-time renaming, one of which has an optimal name space of $M = 2k-1$. In contrast to prior algorithms, our one-time renaming algorithms have time complexity that depends only on k, the number of participating processes. These algorithms employ a novel technique that uses "building blocks" based on the "fast path" mechanism employed by Lamport's fast mutual exclusion algorithm [8]. We also present a read/write algorithm for long-lived renaming that yields a name space of size $k(k+1)/2$. This algorithm uses a modified version of the one-time building block that allows processes to "reset" the building block, so that it may be used repeatedly. Unfortunately, this results in time complexity that is dependent on N. Nevertheless, this result breaks new ground by showing that long-lived renaming can be implemented with only reads and writes.

Previous and new renaming algorithms that use only read and write operations are summarized in Table 1. We leave open the question of whether read and write operations can be used to implement long-lived renaming with a name space of size $2k-1$ and with time complexity that depends only on k.

In the second part of the paper, we consider long-lived k-renaming algorithms. By definition, M-renaming for $M < k$ is impossible, so with respect to the size of the name space, k-renaming is optimal. As previously mentioned, it is impossible to implement k-renaming using only atomic read and write operations. Thus, all of our k-renaming algorithms employ stronger read-modify-write operations.

We present three wait-free, long-lived k-renaming algorithms. The first such algorithm uses two read-modify-write operations, *set_first_zero* and *clr_bit*. The *set_first_zero* operation is applied to a b-bit shared variable X whose bits are indexed from 0 to $b-1$. If some bit of X is clear, then *set_first_zero*(X) sets the first clear bit of X, and returns its index. If all bits of X are set, then *set_first_zero*(X) leaves X unchanged and returns b. Note that for $b = 1$, *set_first_zero* is equivalent to *test_and_set*. The *set_first_zero* operation for

Reference	Time Complexity	Bits / Variable	Instructions Used
Thm. 4	$\Theta(k)$	1	write and test_and_set
Thm. 4	$\Theta(k/b)$	b	set_first_zero and clr_bit
Thm. 5	$\Theta(\log k)$	$\Theta(\log k)$	bounded_decrement and atomic_add
Thm. 6	$\Theta(\log(k/b))$	$\Theta(\log k)$	above, set_first_zero, and clr_bit

Table 2: A comparison of wait-free long-lived k-renaming algorithms.

$b > 1$ can be implemented, for example, using the *atomff0andset* operation available on the BBN TC2000 multiprocessor [4]. The *clr_bit*(X, i) operation clears the ith bit of X. For $b = 1$, *clr_bit* is a simple write operation. For $b > 1$, *clr_bit* can be implemented, for example, using the *fetch_and_and* operation available on the BBN TC2000.

Our second long-lived k-renaming algorithm employs the *bounded_decrement* and *atomic_add* operations. The *bounded_decrement* operation is similar to the commonly-available *fetch_and_add* operation, expect that *bounded_decrement* does not modify a variable whose value is zero. We do not know of any systems that provide *bounded_decrement* as a primitive operation. However, at the end of Section 5, we show that *bounded_decrement* can be approximated in a lock-free manner using the commonly-available *fetch_and_add* operation. This allows us to obtain a lock-free, long-lived k-renaming algorithm based on *fetch_and_add*.

Our third long-lived k-renaming algorithm combines both algorithms discussed above, improving on the performance of each. Our wait-free, long-lived k-renaming algorithms are summarized in Table 2.

The remainder of the paper is organized as follows. Section 2 contains definitions used in the rest of the paper. In Sections 3 and 4, we present one-time and long-lived renaming algorithms that employ only atomic reads and writes. In Section 5, we present long-lived renaming algorithms that employ stronger read-modify-write operations. Concluding remarks appear in Section 6.

2 Definitions

Our programming notation should be self-explanatory; as an example of this notation, see Figure 2. In this and subsequent figures, each labeled program fragment is assumed to be atomic,[1] unless no labels are given, in which case each line of code is assumed to be atomic.

Notational Conventions: We assume that $1 < k \leq M < N$, and that p and q range over $0..N-1$. Other free variables in expressions are assumed to be universally quantified. We use $P^{x_1, x_2, \ldots, x_n}_{y_1, y_2, \ldots, y_n}$ to denote the expression P with each occurrence of x_i replaced by y_i. The predicate $p@i$ holds iff process p's program counter has the value i. We use $p@S$ as shorthand for $(\exists i : i \in S :: p@i)$, $p.i$ to denote statement i of process p, and $p.var$ to denote p's local variable var. The following is a list of symbols we use in our proofs, in increasing order of binding power: $\equiv$, $\Rightarrow$, $\vee$, $\wedge$, $(=, \neq, <, >, \leq, \geq)$, $(+, -)$, (multiplication,/), $\neg$, $(., @)$, $(\{, \})$. Symbols in parentheses have the same binding power. We sometimes use parentheses to override these binding rules. In our proofs, we sometimes use Hoare triples [6] to denote the effects of a statement execution. $\Box$

In the *one-time M-renaming* problem, each of k processes, with distinct process identifiers ranging over $\{0, ..., N-1\}$, chooses a distinct value ranging over $\{0, ..., M-1\}$. A solution to the M-renaming problem

[1] To simplify our proofs, we sometimes label somewhat lengthy blocks of code. Nonetheless, such code blocks are in keeping with the atomic instructions used. For example, statement 3 in Figure 4 is assumed to atomically read $X[i, j]$, assign $stop := true$ or $i := i + 1$ depending on the value read, check the loop condition, and set the program counter of the executing process to 0 or 4, accordingly. Note, however, that $X[i, j]$ is the only shared variable accessed by statement 3. Because all other variables accessed by this statement are private, statement 3 can be easily implemented using a single atomic read of a shared variable. This is in keeping with the read/write atomicity assumed for this algorithm.

```
process p                                                         /* 0 ≤ p < N */
private variable name : 0..M − 1                                  /* Name received */
   while true do
      Remainder Section;               /* Ensure at most k processes rename concurrently */
      Getname Section;                 /* Assigns a value ranging over {0..M − 1} to p.name */
      Working Section;
      Putname Section                                      /* Release the name obtained */
   od
```

Figure 1: Organization of processes accessing a long-lived renaming algorithm.

consists of a wait-free code fragment for each process p that assigns a value ranging over $\{0, ..., M-1\}$ to a private variable $p.name$ and then halts. For $p \neq q$, the same value cannot be assigned to both $p.name$ and $q.name$.

In the *long-lived M-renaming* problem, each of N distinct processes repeatedly executes a *remainder section*, acquires a name by executing a *getname section*, uses that name in a *working section*, and then releases the name by executing a *putname section*. The organization of these processes is shown in Figure 1. It is assumed that each process is initially in its remainder section, and that the remainder section guarantees that at most k processes are outside their remainder sections at any time. A solution to the long-lived M-renaming problem consists of wait-free code fragments that implement the getname and putname sections shown in Figure 1, along with associated shared variables. The getname section for process p is required to assign a value ranging over $\{0..M-1\}$ to $p.name$. If distinct processes p and q are in their working sections, then it is required that $p.name \neq q.name$.

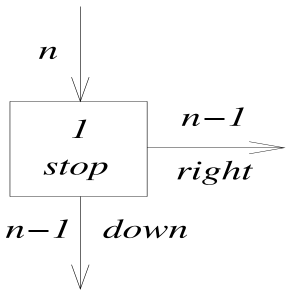
As discussed in the introduction, our algorithms use the *set_first_zero*, *clr_bit*, and *bounded_decrement* operations, among other well-known operations. We define these operations formally by the following atomic code fragments, where X is a b-bit shared variable whose bits are indexed from 0 to $b-1$, and Y is a non-negative integer.

```
set_first_zero(X)        ≡  if (∃n : 0 ≤ n < b :: ¬X[n]) then
                               m := (min n : 0 ≤ n < b :: ¬X[n]);  X[m] := true;  return m
                            else
                               return b
                            fi

clr_bit(X, i)            ≡  X[i] := false

bounded_decrement(Y)     ≡  m := Y;  if Y ≠ 0 then Y := Y − 1 fi;  return m
```

We measure the time complexity of our algorithms in terms of the worst case number of shared variable accesses required to acquire (and release, if long-lived) a name once.

3 One-Time Renaming using Reads and Writes

In this section, we present one-time renaming algorithms that employ only atomic read and write operations. We start by presenting a $(k(k+1)/2)$-renaming algorithm that has $\Theta(k)$ time complexity. We then describe how this algorithm can be combined with previous results [5] to obtain a $(2k-1)$-renaming algorithm with $\Theta(k^4)$ time complexity. It has been shown that renaming is impossible for fewer than $2k-1$ names when using only reads and writes, so with respect to the size of the resulting name space, this algorithm is optimal. Our one-time $(k(k+1)/2)$-renaming algorithm is based on a "building block", which we describe next.




```
shared variable  X : {⊥} ∪ {0..N − 1};
                 Y : boolean
initially X = ⊥ ∧ ¬Y

private variable  move : {stop, right, down}

X := p;
if Y then move := right
else
     Y := true;
     if X = p  then move := stop
               else  move := down
     fi
fi
```

Figure 2: The one-time building block and the code fragment that implements it.

3.1 The One-Time Building Block

The one-time building block, depicted in Figure 2, is in the form of a wait-free code fragment that assigns to a private variable *move* one of three values: *stop*, *right*, or *down*. If each of n processes executes this code fragment at most once, then at most one process receives a value of *stop*, at most $n-1$ processes receive a value of *right*, and at most $n-1$ processes receive a value of *down*. We say that a process that receives a value of *down* "goes down", a process that receives a value of *right* "goes right", and a process that receives a value of *stop* "stops". Figure 2 shows n processes accessing a building block, and the maximum number of processes that receive each value.

The code fragment shown in Figure 2 shows how the building block can be implemented using atomic read and write operations. The technique employed is essentially that of the "fast path" mechanism used in Lamport's fast mutual exclusion algorithm [8]. A process that stops corresponds to a process successfully "taking the fast path" in Lamport's algorithm. The value assigned to *move* by a process p that fails to "take the fast path" is determined by the branch p takes: if p detects that Y holds, then p goes right, and if p detects that $X \neq p$ holds, then p goes down.

To see that the code fragment shown in Figure 2 satisfies the requirements of our building block, note that it is impossible for all n processes to go right — a process can go right only if another process previously assigned $Y := true$. Second, the last process p to assign $X := p$ cannot go down, because if it tests X, then it detects that $X = p$, and therefore stops. Thus, it is impossible for all n processes to go down. Finally, because Lamport's algorithm prevents more than one processes from "taking the fast path", it is impossible for more than one process to stop. Thus, the code fragment shown in Figure 2 satisfies the requirements of the building block.

In the next section, we show how these building blocks can be used to solve the renaming problem. The basic approach is to use such building blocks to "split" processes into successively smaller groups. Because at most one process stops at any particular building block, a process that stops can be given a unique name associated with that building block. Furthermore, when the size of a group has been decreased often enough that at most one process remains, that process (if it exists) can be given a name immediately.

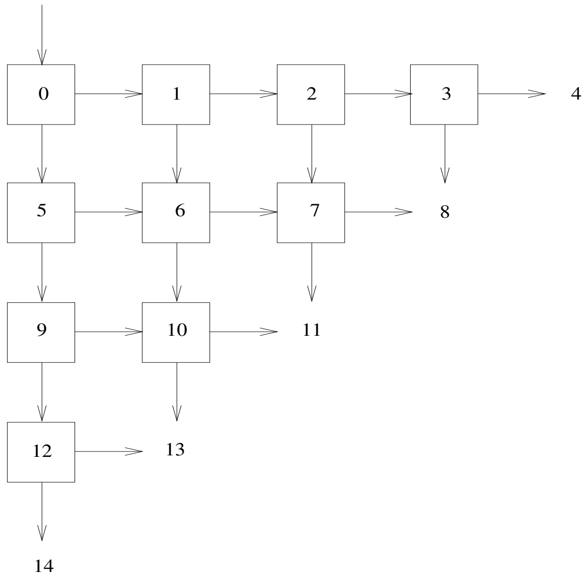


Figure 3: $k(k-1)/2$ building blocks in a grid, depicted for $k = 5$.

3.2 Using the One-Time Building Block to Solve Renaming

In this section, we use $k(k-1)/2$ one-time building blocks arranged in a "grid" to solve one-time renaming; this approach is depicted in Figure 3 for $k = 5$. In order to acquire a name, a process p accesses the building block at the top left corner of the grid. If p receives a value of *stop*, then p acquires the name associated with that building block. Otherwise, p moves either right or down in the grid, according to the value received. This is repeated until p receives a value of *stop* at some building block, or p has accessed $k-1$ building blocks. The name returned is calculated based on p's final position in the grid. In Figure 3, each grid position is labeled with the name associated with that position. Because no process takes more than $k-1$ steps, only the upper left triangle of the grid is used, as shown in Figure 3.

The algorithm is presented more formally in Figure 4. Note that each building block in the grid is implemented using the code fragment shown in Figure 2. At most one process stops at each building block, so a process that stops at a building block receives a unique name. However, a process may also obtain a name by taking $k-1$ steps in the grid. The correctness proof in the following section shows that distinct processes that take $k-1$ steps in the grid acquire distinct names. Specifically, (I9) implies that no two processes arrive at the same grid position after taking $k-1$ steps in the grid.

3.3 Correctness Proof

The following simple properties follow directly from the program text in Figure 4, and are stated without proof. Note that (I2) is used to prove (I4) and (I5).

invariant $p.i \geq 0 \ \wedge \ p.j \geq 0$ (I1)

invariant $p@\{0..3\} \Rightarrow \neg p.stop \ \wedge \ p.i + p.j < k-1$ (I2)

invariant $p@\{4..5\} \ \wedge \ \neg p.stop \Rightarrow p.i + p.j = k-1$ (I3)

invariant $p@\{0..3\} \ \vee \ (p@\{4..5\} \ \wedge \ p.stop) \Rightarrow p.i + p.j < k-1$ (I4)

invariant $0 \leq p.i + p.j \leq k-1$ (I5)

invariant $p@5 \Rightarrow p.name = p.i(k-(p.i-1)/2) + p.j$ (I6)

shared variable X : **array**$[0..k-2, 0..k-2]$ **of** $\{\bot\} \cup \{0..N-1\}$;
Y : **array**$[0..k-2, 0..k-2]$ **of boolean**
initially $(\forall r, c : 0 \leq r < k-1 \ \wedge\ 0 \leq c < k-1 :: X[r,c] = \bot \ \wedge\ \neg Y[r,c])$

process p /* k distinct processes ranging over $0..N-1$ */
private variable $name : 0..k(k+1)/2 - 1$;
$stop$: **boolean**;
$i, j : 0..k-1$
initially $i = 0 \ \wedge\ j = 0 \ \wedge\ \neg stop$

    while $i + j < k - 1 \ \wedge\ \neg stop$ **do** /* Move down or across grid until stopping or reaching edge */
0:       $X[i,j] := p$;
1:       **if** $Y[i,j]$ **then** $j := j + 1$ /* Move down */
        else
2:           $Y[i,j] := true$;
3:           **if** $X[i,j] = p$ **then** $stop := true$ **else** $i := i + 1$ **fi** /* Stop or move right */
        fi
    od;
4:     $name := i(k - (i-1)/2) + j$; /* Calculate name based on position in grid */
5:     **halt**

Figure 4: One-time renaming using a grid of building blocks.

For each of the remaining invariants, a correctness proof is given.[2]

invariant $r \geq 0 \ \wedge\ c \geq 0 \ \wedge\ r + c < k - 1 \ \wedge\ Y[r,c] \Rightarrow$
$$(\exists p :: (p@\{3..5\} \ \wedge\ p.i = r \ \wedge\ p.j = c) \ \vee\ (p.i > r \ \wedge\ p.j = c)) \qquad \text{(I7)}$$

Proof: Assume $r \geq 0 \wedge c \geq 0 \wedge r+c < k-1$. Initially $Y[r,c]$ is false, so (I7) holds. To prove that (I7) is not falsified, it suffices to consider those statements that may establish[3] $Y[r,c]$, or that may falsify $p@\{3..5\}$ or modify $p.i$ or $p.j$ for some p. The statements to check are $q.2$, $p.1$, and $p.3$, where q is any process. Observe that $q.2$ may establish $Y[r,c]$ only if executed when $q.i = r$ and $q.j = c$, in which case it also establishes $q@3 \ \wedge\ q.i = r \ \wedge\ q.j = c$. For statement $p.1$, we have the following.

$p@1 \ \wedge\ p.i > r \ \wedge\ p.j = c \ \wedge\ Y[p.i, p.j] \wedge \text{(I7)}^{r,c,p}_{p.i,p.j,q}$

$\Rightarrow p@1 \ \wedge\ p.i > r \ \wedge\ p.j = c \ \wedge\ p.i \geq 0 \ \wedge\ p.j \geq 0 \ \wedge\ p.i + p.j < k - 1 \ \wedge\ Y[p.i, p.j] \wedge \text{(I7)}^{r,c,p}_{p.i,p.j,q}$
, by (I1) and (I4).

$\Rightarrow p@1 \ \wedge\ p.i > r \ \wedge\ p.j = c \ \wedge\ (\exists q :: (q@\{3..5\} \ \wedge\ q.i = p.i \ \wedge\ q.j = p.j) \ \vee\ (q.i > p.i \ \wedge\ q.j = p.j))$
, by definition of (I7).

$\Rightarrow (\exists q : q \neq p :: q.i > r \ \wedge\ q.j = c)$, $p@1 \ \wedge\ (q@\{3..5\} \ \vee\ q.i > p.i)$ implies $p \neq q$.

$\{p@1 \ \wedge\ p.i > r \ \wedge\ p.j = c \ \wedge\ Y[p.i, p.j] \wedge \text{(I7)}^{r,c,p}_{p.i,p.j,q}\}\ p.1\ \{(\exists q : q \neq p :: q.i > r \ \wedge\ q.j = c)\}$
, by preceding derivation and axiom of assignment.

$\{p@1 \ \wedge\ p.i > r \ \wedge\ p.j = c \ \wedge\ \neg Y[p.i, p.j]\}\ p.1\ \{p.i > r \ \wedge\ p.j = c\}$, $p.1$ does not modify $p.j$ in this case.

[2] We prove that an assertion I is an invariant by showing that it holds inductively or that it follows from established invariants. In an inductive proof, it is required to show that I holds initially and is not falsified by any statement execution, i.e., if I (and perhaps other established invariants) holds before a given statement is executed then I holds afterwards.

[3] We say that an execution of statement $p.i$ *establishes* a predicate P iff $\neg P$ holds before that statement execution and P holds afterwards.

$\{p@1 \;\wedge\; \neg(p.i > r \;\wedge\; p.j = c) \wedge \text{(I7)}\}\; p.1$
$\quad \{\neg Y[r,c] \;\vee\; (\exists q : q \neq p :: (q@\{3..5\} \;\wedge\; q.i = r \;\wedge\; q.j = c) \;\vee\; (q.i > r \;\wedge\; q.j = c))\}$
, by definition of (I7), precondition implies postcondition, which is unchanged by $p.1$.

The above assertions imply that $p.1$ does not falsify (I7). The following assertions imply that $p.3$ does not falsify (I7).

$\{p@3 \;\wedge\; p.i \geq r \;\wedge\; p.j = c \;\wedge\; X[p.i, p.j] = p\}\; p.3\; \{p@4 \;\wedge\; p.i \geq r \;\wedge\; p.j = c\}$
, $p.i$ is unchanged and $p.stop$ is established in this case.

$\{p@3 \;\wedge\; p.i \geq r \;\wedge\; p.j = c \;\wedge\; X[p.i, p.j] \neq p\}\; p.3\; \{p.i > r \;\wedge\; p.j = c\}$, $p.i$ is incremented in this case.

$\{p@3 \;\wedge\; \neg(p.i \geq r \;\wedge\; p.j = c) \wedge \text{(I7)}\}\; p.3$
$\quad \{\neg Y[r,c] \;\vee\; (\exists q : q \neq p :: (q@\{3..5\} \;\wedge\; q.i = r \;\wedge\; q.j = c) \;\vee\; (q.i > r \;\wedge\; q.j = c))\}$
, by definition of (I7), precondition implies postcondition, which is unchanged by $p.3$. $\Box$

The following invariant shows that if $X[r,c]$ has been modified since process q assigned $X[r,c]$, then there is some process p in row r at or to the right of column c. This property is used to show that not all processes that access building block (r,c) proceed to row $r+1$.

invariant $r \geq 0 \;\wedge\; c \geq 0 \;\wedge\; r + c < k - 1 \;\wedge\; q@\{1..3\} \;\wedge\; q.i = r \;\wedge\; q.j = c \;\wedge\; X[r,c] \neq q \Rightarrow$
$\quad (\exists p : p \neq q :: p.i = r \;\wedge\; ((p.j = c \;\wedge\; ((p@\{1..3\} \;\wedge\; X[r,c] = p) \;\vee\; p@\{4..5\})) \;\vee\; p.j > c))$ (I8)

Proof: Assume $r \geq 0 \;\wedge\; c \geq 0 \;\wedge\; r + c < k - 1$. Initially $q@0$ holds, so (I8) holds. To prove that (I8) is not falsified, it suffices to consider those statements that may establish $q@\{1..3\}$; falsify $p@\{1..3\}$ or $p@\{4..5\}$; or modify $q.i$, $q.j$, X, $p.i$, or $p.j$, where $p \neq q$. The statements to check are $q.0$, $q.1$, $q.3$, $p.0$, $p.1$, and $p.3$, where $p \neq q$.

Observe that $q.i = r \;\wedge\; q.j = c \;\wedge\; X[r,c] \neq q$ does not hold after the execution of $q.0$, and that $q.1$ and $q.3$ both establish $q@\{0,4\}$. Furthermore, $p.0$ establishes $X[r,c] \neq q$ only if $p.i = r$ and $p.j = c$, in which case $p@1 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; X[r,c] = p$ holds afterwards. Also, statement $p.1$ can only increment $p.j$. Therefore, if $p.1$ falsifies $p.j = c$, then it establishes $p.j > c$, and it does not falsify $p.i = r$.

This leaves only statement $p.3$. Statement $p.3$ could falsify (I8) only by falsifying $p.i = r \;\wedge\; p.j = c \;\wedge\; X[r,c] = p$ or by falsifying $p.i = r \;\wedge\; p.j > c$. In the first case, we have $\{p@3 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; X[r,c] = p\}\; p.3\; \{p@4 \;\wedge\; p.i = r \;\wedge\; p.j = c\}$, so $p.3$ does not falsify (I8). For the second case, observe that $p.3$ can only falsify $p.i = r \;\wedge\; p.j > c$ if executed when $p@3 \;\wedge\; p.i = r \;\wedge\; p.j > c \;\wedge\; X[p.i, p.j] \neq p$ holds. The following assertions imply that statement $p.3$ does not falsify (I8) in this case.

$\{p@3 \;\wedge\; p.i = r \;\wedge\; p.j > c \;\wedge\; X[p.i, p.j] \neq p \;\wedge\; (q.i \neq r \;\vee\; q.j \neq c)\}\; p.3\; \{q.i \neq r \;\vee\; q.j \neq c\}$
, $p.3$ does not modify $q.i$ or $q.j$ (recall that $p \neq q$).

$\quad p@3 \;\wedge\; p.i = r \;\wedge\; p.j > c \;\wedge\; X[p.i, p.j] \neq p \;\wedge\; q.i = r \;\wedge\; q.j = c \wedge \text{(I8)}^{q,r,c,p}_{p,p.i,p.j,s}$

$\Rightarrow p@3 \;\wedge\; p.i = r \;\wedge\; p.j > c \;\wedge\; q.j = c \;\wedge\; p.i \geq 0 \;\wedge\; p.j \geq 0 \;\wedge\; p.i + p.j < k - 1 \;\wedge$
$\quad X[p.i, p.j] \neq p \wedge \text{(I8)}^{q,r,c,p}_{p,p.i,p.j,s}$, by (I1) and (I4).

$\Rightarrow p@3 \;\wedge\; p.i = r \;\wedge\; p.j > c \;\wedge\; q.j = c \;\wedge$
$\quad (\exists s : s \neq p :: s.i = p.i \;\wedge\; ((s.j = p.j \;\wedge\; ((s@\{1..3\} \;\wedge\; X[p.i, p.j] = s) \;\vee\; s@\{4..5\})) \;\vee\; s.j > p.j)))$
, by definition of (I8).

$\Rightarrow p.j > c \;\wedge\; q.j = c \;\wedge\; (\exists s : s \neq p :: s.i = r \;\wedge\; s.j > c)$, predicate calculus.

$\Rightarrow (\exists s : s \neq p \ \wedge \ s \neq q :: s.i = r \ \wedge \ s.j > c)$, $q.j = c \ \wedge \ s.j > c$ implies $s \neq q$.

$\{p@3 \ \wedge \ p.i = r \ \wedge \ p.j > c \ \wedge \ X[p.i, p.j] \neq p \ \wedge \ q.i = r \ \wedge \ q.j = c \wedge (\mathrm{I8})^{q,r,c,p}_{p,p.i,p.j,s}\}$ $p.3$
$\{(\exists s : s \neq q :: s.i = r \ \wedge \ s.j > c)\}$
, by preceding derivation, precondition implies postcondition, which is not falsified by $p.3$. □

The following invariant shows that at most $k - r - c$ processes access building blocks in the "sub-grid" whose top left corner is at position (r, c). In particular, it shows that at most one process accesses any building block that is $k - 1$ "steps" away from the top left corner of the grid.

invariant $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c \leq k - 1 \Rightarrow (|\{p :: p.i \geq r \ \wedge \ p.j \geq c\}| \leq k - r - c)$ (I9)

Proof: Assume $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c \leq k - 1$. Initially $p.i = 0 \ \wedge \ p.j = 0$ holds for all p, so (I9) holds. To prove that (I9) is not falsified, it suffices to consider those statements that may establish $q.i \geq r \ \wedge \ q.j \geq c$ for some q. There are two statements to check, namely $q.1$ and $q.3$.

Observe that statement $q.1$ can establish $q.i \geq r \ \wedge \ q.j \geq c$ only if executed when $q.i \geq r \ \wedge \ q.j = c - 1 \ \wedge \ Y[q.i, q.j]$ holds. To see that (I9) is not falsified in this case, consider the following derivation.

$q@1 \ \wedge \ q.i \geq r \ \wedge \ q.j = c - 1 \ \wedge \ Y[q.i, q.j] \wedge (\mathrm{I7})^{r,c}_{q.i,q.j} \wedge (\mathrm{I9})^{c}_{c-1}$

$\Rightarrow q@1 \ \wedge \ q.i \geq r \ \wedge \ q.j = c - 1 \ \wedge \ q.i \geq 0 \ \wedge \ q.j \geq 0 \ \wedge \ q.i + q.j < k - 1 \ \wedge \ Y[q.i, q.j] \wedge (\mathrm{I7})^{r,c}_{q.i,q.j} \ \wedge$
$c - 1 \geq 0 \wedge (\mathrm{I9})^{c}_{c-1}$, by (I1) and (I4); note that $q.j = c - 1 \wedge (\mathrm{I1})$ implies $c - 1 \geq 0$.

$\Rightarrow q.i \geq r \ \wedge \ q.j = c - 1 \ \wedge \ (\exists s : s \neq q :: s.i \geq q.i \ \wedge \ s.j = q.j) \ \wedge \ c - 1 \geq 0 \wedge (\mathrm{I9})^{c}_{c-1}$
, definition of (I7); note that $q@1 \ \wedge \ (s@\{3..5\} \ \vee \ s.i > q.i)$ implies $s \neq q$.

$\Rightarrow (|\{s :: s.i \geq r \ \wedge \ s.j = c - 1\}| \geq 2 \ \wedge \ c - 1 \geq 0) \ \wedge \ (|\{p :: p.i \geq r \ \wedge \ p.j \geq c - 1\}| \leq k - r - c + 1)$
, predicate calculus and definition of (I9); recall that $r \geq 0$ and $r + c \leq k - 1$.

$\Rightarrow |\{p :: p.i \geq r \ \wedge \ p.j \geq c\}| \leq k - r - c - 1$, predicate calculus.

$\{q@1 \ \wedge \ q.i \geq r \ \wedge \ q.j = c - 1 \ \wedge \ Y[q.i, q.j] \wedge (\mathrm{I7})^{r,c}_{q.i,c-1} \wedge (\mathrm{I9})^{c}_{c-1}\}$ $q.1$
$\{|\{q :: q.i \geq r \ \wedge \ q.j \geq c\}| \leq k - r - c\}$
, by preceding derivation; $q.1$ increases $|\{q :: q.i \geq r \ \wedge \ q.j \geq c\}|$ by at most one.

Statement $q.3$ can establish $q.i \geq r \wedge q.j \geq c$ only if executed when $q.i = r - 1 \wedge q.j \geq c \wedge X[q.i, q.j] \neq q$ holds. The reasoning for this case is similar to that given above for statement $q.1$, except that (I8) is used instead of (I7). □

invariant $p@3 \ \vee \ (p@\{4..5\} \ \wedge \ p.stop) \Rightarrow Y[p.i, p.j]$ (I10)

Proof: Initially $p@0$ holds, so (I10) holds. (I10) could potentially be falsified by any statement that establishes $p@3$ or $p@\{4..5\} \ \wedge \ p.stop$ or that modifies $p.i$ or $p.j$. (Note that no statement falsifies any element of Y.) The statements to check are $p.1$, $p.2$, and $p.3$. By (I2), $p.1$ establishes $p@\{0, 2, 4\} \ \wedge \ \neg p.stop$. Also, statement $p.2$ establishes $Y[p.i, p.j]$. The following assertions imply that $p.3$ does not falsify (I10).

$\{p@3 \ \wedge \ X[p.i.p.j] = p \wedge (\mathrm{I10})\}$ $p.3$ $\{Y[p.i, p.j]\}$
, by definition of (I10), precondition implies postcondition; $p.3$ does not modify Y, $p.i$, or $p.j$ in this case.

$\{p@3 \ \wedge \ X[p.i, p.j] \neq p\}$ $p.3$ $\{p@\{0, 4\} \ \wedge \ \neg p.stop\}$, by (I2), precondition implies $\neg p.stop$. □

The following invariant shows that if some process p stops at building block (r, c), then no other process can stop at that building block. This property is used to show that processes that stop in the grid receive distinct names.

invariant $p \neq q \;\wedge\; p@\{4..5\} \;\wedge\; p.stop \Rightarrow (q.i \neq p.i \;\vee\; q.j \neq p.j \;\vee\; q@0 \;\vee$
$(q@1 \;\wedge\; Y[q.i, q.j]) \;\vee\; (q@\{1..3\} \;\wedge\; X[q.i, q.j] \neq q) \;\vee\; (q@\{4..5\} \;\wedge\; \neg q.stop))$ (I11)

Proof: Assume that $p \neq q$. Initially $p@0$ holds, so (I11) holds. To prove that (I11) is not falsified, it suffices to consider those statements that may establish $p@\{4, 5\}$; falsify $q@0$, $q@1$, $q@\{1..3\}$, or $q\{4, 5\}$; or modify $p.i$, $p.j$, $q.i$, $q.j$, $p.stop$, $q.stop$, X, or Y. The statements to check are $p.0$, $p.1$, $p.2$, $p.3$, $q.0$, $q.1$, $q.2$, and $q.3$. Observe that $p@\{4..5\}$ is false after the execution of $p.0$ or $p.2$. Also, by (I2), $p.stop$ is false after the execution of $p.1$. For $p.3$, we have the following.

$\{p@3 \;\wedge\; X[p.i, p.j] \neq p\}\; p.3\; \{p@0 \;\vee\; (p@4 \;\wedge\; \neg p.stop)\}$, by (I2), precondition implies $\neg p.stop$.

$\{p@3 \;\wedge\; X[p.i, p.j] = p \;\wedge\; (q.i \neq p.i \;\vee\; q.j \neq p.j \;\vee\; q@0)\}\; p.3\; \{p.i \neq q.i \;\vee\; p.j \neq q.j \;\vee\; q@0\}$
, precondition implies postcondition (recall that $p \neq q$); $p.3$ does not modify $p.i$ in this case.

$\{p@3 \;\wedge\; X[p.i, p.j] = p \;\wedge\; q.i = p.i \;\wedge\; q.j = p.j \;\wedge\; q@\{1..3\}\}\; p.3\; \{q@\{1..3\} \;\wedge\; X[q.i, q.j] \neq q\}$
, $p \neq q$, so precondition implies postcondition, which is not falsified by $p.3$.

$\{p@3 \;\wedge\; X[p.i, p.j] = p \;\wedge\; q.i = p.i \;\wedge\; q.j = p.j \;\wedge\; q@\{4..5\} \wedge (\text{I11})^{p,q}_{q,p}\}\; p.3\; \{q@\{4..5\} \;\wedge\; \neg q.stop\}$
, by the definition of (I11), precondition implies postcondition, which is not falsified by $p.3$.

The above assertions imply that $p.3$ does not falsify (I11). As for process q, first note that $q.0$ establishes $q@1$, which by (I10) implies that $\neg(p@\{4, 5\} \;\wedge\; p.stop) \;\vee\; (q@1 \;\wedge\; Y[p.i, p.j])$ holds. The latter disjunct implies that $q.i \neq p.i \;\vee\; q.j \neq p.j \;\vee\; (q@1 \;\wedge\; Y[q.i, q.j])$ holds. Statement $q.1$ can falsify (I11) only if executed when $q@1 \;\wedge\; Y[q.i, q.j]$ or $q@\{1..3\} \;\wedge\; X[q.i, q.j] \neq q$ holds. However, observe the following.

$\{q@1 \;\wedge\; Y[q.i, q.j]\}\; q.1\; \{q@0 \;\vee\; (q@4 \;\wedge\; \neg q.stop)\}$, by (I2), precondition implies $\neg q.stop$.

$\{q@1 \;\wedge\; \neg Y[q.i, q.j] \;\wedge\; X[q.i, q.j] \neq q\}\; q.1\; \{q@2 \;\wedge\; X[q.i, q.j] \neq q\}$, $q.1$ does not modify $q.j$ in this case.

Although $q.2$ modifies Y, it cannot falsify any disjunct of the consequent of (I11).

Statement $q.3$ could only falsify (I11) by falsifying $q@3 \;\wedge\; X[q.i, q.j] \neq q$. However, because $q@3 \wedge$ (I2) implies $\neg q.stop$, we have $\{q@3 \;\wedge\; X[q.i, q.j] \neq q \wedge (\text{I2})\}\; q.3\; \{q@0 \;\vee\; (q@4 \;\wedge\; \neg q.stop)\}$. □

invariant $p \neq q \;\wedge\; p@\{4..5\} \;\wedge\; q@\{4..5\} \Rightarrow p.i \neq q.i \;\vee\; q.j \neq q.j$ (I12)

Proof: If $p \neq q \wedge p@\{4..5\} \wedge q@\{4..5\} \wedge p.stop$ holds, then by (I3), (I4), and (I11), the consequent holds. If $p \neq q \;\wedge\; p@\{4..5\} \;\wedge\; q@\{4..5\} \;\wedge\; \neg p.stop$ holds, then by (I1), (I3), and (I9), $|\{q :: q.i \geq p.i \;\wedge\; q.j \geq p.j\}| \leq 1$ holds, which implies that the consequent holds. □

Claim 1: Let c, d, c', and d' be nonnegative integers satisfying $(c \neq c' \vee d \neq d') \wedge (c+d \leq k-1) \wedge (c'+d' \leq k-1)$. Then, $c(k - (c-1)/2) + d \neq c'(k - (c'-1)/2) + d'$.

Proof: The claim is straightforward if $c = c'$, so assume that $c \neq c'$. Without loss of generality assume that $c < c'$. Then,

$$\begin{aligned}
c(k-(c-1)/2)+d &\leq c(k-(c-1)/2)+k-1-c && ,\ d \leq k-1-c.\\
&= kc - c^2/2 - c/2 + k - 1 \\
&< (c+1)(k-c/2) \\
&\leq c'(k-(c'-1)/2) && ,\ c \leq c'-1.\\
&\leq c'(k-(c'-1)/2)+d' && ,\ d' \text{ is nonnegative.}
\end{aligned}$$
□

invariant $p@5 \ \wedge\ q@5 \ \wedge\ p \neq q \Rightarrow p.name \neq q.name$ (I13)

Proof: The following derivation implies that (I13) is an invariant.

$p@5 \ \wedge\ q@5 \ \wedge\ p \neq q \ \wedge\ p.name = q.name$

$\Rightarrow p@5 \ \wedge\ q@5 \ \wedge\ p.name = q.name \ \wedge\ (p.i \neq q.i \ \vee\ p.j \neq q.j) \ \wedge\ p.i + p.j \leq k-1 \ \wedge\ q.i + q.j \leq k-1$, by (I5) and (I12).

$\Rightarrow (p.i \neq q.i \ \vee\ p.j \neq q.j) \ \wedge\ p.i + p.j \leq k-1 \ \wedge\ q.i + q.j \leq k-1 \ \wedge\ p.i \geq 0 \ \wedge\ p.j \geq 0 \ \wedge$
$q.i \geq 0 \ \wedge\ q.j \geq 0 \ \wedge\ p.i(k-(p.i-1)/2) + p.j = q.j(k-(q.j-1)/2) + q.j$, by (I1) and (I6).

$\Rightarrow false$, by Claim 1 with $c = p.i$, $d = p.j$, $c' = q.i$, and $d' = q.j$. □

This completes the proof that distinct processes that execute the code in Figure 4 acquire distinct names. The following claim is used to prove that each process acquires a name ranging over $\{0..k(k+1)/2-1\}$.

Claim 2: Let c and d be nonnegative integers satisfying $c + d \leq k-1$. Then $0 \leq c(k-(c-1)/2) + d < k(k+1)/2$.

Proof: It follows from the statement of the claim that $c \leq k-1$. Thus, $k-(c-1)/2 > 0$. Also, $c \geq 0$ and $d \geq 0$. Thus, $c(k-(c-1)/2) + d \geq 0$. To see that $c(k-(c-1)/2) + d < k(k+1)/2$, consider the following derivation.

$$\begin{array}{rcll} c(k-(c-1)/2)+d & \leq & c(k-(c-1)/2)+d(d+1)/2 & , d \geq 0. \\ & \leq & c(k-(c-1)/2)+(k-1-c)(k-c)/2 & , d \leq k-1-c. \\ & = & c+k(k-1)/2 & \\ & \leq & k-1+k(k-1)/2 & , c \leq k-1. \\ & < & k(k+1)/2 & \end{array}$$

□

invariant $p@5 \Rightarrow 0 \leq p.name < k(k+1)/2$ (I14)

Proof: (I14) follows from (I1), (I5), (I6), and Claim 2. □

(I13) and (I14) prove that the algorithm shown in Figure 4 correctly implements $(k(k+1)/2)$-renaming. Wait-freedom is trivial because in each pass through the loop, either *p.stop* is established, or *p.i* or *p.j* is incremented. It is easy to see that a process executes the loop at most $k-1$ times before terminating. Each iteration performs at most four shared variable accesses. Thus, we have the following result.

Theorem 1: Using *read* and *write*, wait-free, one-time $(k(k+1)/2)$-renaming can be implemented with time complexity $4(k-1)$. □

Using the algorithm described in this section, k processes can reduce the size of their name space from N to $k(k+2)/2$ with time complexity $\Theta(k)$. Using the algorithm presented in [5], k processes can reduce the size of their name space from N to $2k-1$ with time complexity $\Theta(Nk^2)$. Combining the two algorithms, k processes can reduce the size of their name space from N to $2k-1$ with time complexity $\Theta(k) + \Theta((k(k+1)/2)k^2) = \Theta(k^4)$. Thus, we have the following result. By the results of [7], this algorithm is optimal with respect to the size of the name space.

Theorem 2: Using *read* and *write*, wait-free, one-time $(2k-1)$-renaming can be implemented with time complexity $\Theta(k^4)$. □

4 Long-Lived Renaming using Reads and Writes

In this section, we present a long-lived renaming algorithm that uses only atomic read and write operations. This algorithm is based on the grid algorithm presented in the previous section. To enable processes to release names as well as acquire names, we modify the one-time building block. The modification allows a process to "reset" a building block that it has previously accessed. This algorithm yields a name space of size $k(k+1)/2$ and has time complexity $\Theta(Nk)$. We now give an informal description of the algorithm, and then present a formal correctness proof.

4.1 Using the Long-Lived Building Block for Long-Lived Renaming

Our long-lived renaming algorithm based on reads and writes is shown in Figure 5. As in the one-time algorithm presented in the previous section, a process acquires a name by starting at the top left corner of a grid of building blocks, and by moving through the grid according to the value received from each building block. The building blocks, which are similar to those described in the previous section, are accessed in statements 2 through 5 in Figure 5. There are two significant differences between this algorithm and the one-time renaming algorithm.

Firstly, the single Y-bit used in the one-time algorithm is replaced by N Y-bits — one for each process. Instead of setting a common Y-bit, each process p sets a distinct bit $Y[p]$ (see statement 4). This modification allows a process to "reset" the building block by clearing its Y-bit. A process resets a building block it has accessed before proceeding to the next building block in the grid (see statement 6), or when releasing the name associated with that building block (see statement 8). The building blocks are reset to allow processes to reuse the grid to acquire names repeatedly. (It may seem more intuitive to reset all building blocks accessed when releasing a name. In fact, this does not affect correctness, and resetting each building block before accessing the next avoids the need for a data structure to record which building blocks were accessed.)

To see why N Y-bits are used, observe that in the one-time building block, the Y-variable is never reset, so using a single bit suffices. However, if only one Y-bit is used in the long-lived algorithm, a process might reset Y immediately after another process, say p, sets Y. Because the value assigned by p to Y has been overwritten, another process q may subsequently access the building block and fail to detect that p has accessed the building block. In this case, it is possible for both p and q to receive a value of *stop* from the same building block.

The second difference between the one-time and long-lived building blocks is that they differ in time complexity. Instead of reading a single Y-variable, each process now reads all N Y-bits. This results in $\Theta(N)$ time complexity for accessing the long-lived building block. It may seem that all N Y-bits must be read in an atomic "snapshot" because, for example, p's write to $Y[p]$ might occur concurrently with q's scan of the Y-bits. In fact, this is unnecessary, because the fact that these operations are concurrent is sufficient to ensure that either p or q will not receive a value of *stop* from the building block.

4.2 Correctness Proof

In accordance with the problem specification, we assume the following invariant.

invariant $|\{p :: p@\{1..8\}| \leq k$ (I15)

The following simple properties follow directly from the program text in Figure 5, and are stated without proof. Note that (I17) is used to prove (I18) and (I19).

invariant $p@5 \ \vee\ (p@\{6..8\} \ \wedge\ p.move = stop) \Rightarrow Y[p.i, p.j][p]$ (I16)

invariant $p@\{2..6\} \ \vee\ (p@\{7..8\} \ \wedge\ p.move = stop) \Rightarrow p.i + p.j < k - 1$ (I17)

invariant $p@\{7..8\} \ \wedge\ p.move \neq stop \Rightarrow p.i + p.j = k - 1$ (I18)

invariant $0 \leq p.i + p.j \leq k - 1$ (I19)

invariant $p.i \geq 0 \;\wedge\; p.j \geq 0$ (I20)

invariant $p@3 \Rightarrow 0 \leq p.h < N$ (I21)

invariant $p@8 \Rightarrow p.name = p.i(k - (p.i - 1)/2) + p.j$ (I22)

For the remaining invariants, a formal correctness proof is given.

invariant $r \geq 0 \;\wedge\; c \geq 0 \;\wedge\; r + c < k - 1 \;\wedge\; Y[r, c][p] \Rightarrow$
$p@\{5..8\} \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move \neq right$ (I23)

Proof: Assume $r \geq 0 \;\wedge\; c \geq 0 \;\wedge\; r + c < k - 1$. Initially $Y[r, c][p]$ is false, so (I23) holds. To prove that (I23) is not falsified, it suffices to consider statements that potentially establish $Y[r, c][p]$, falsify $p@\{5..8\}$, modify $p.i$ or $p.j$, or establish $p.move = right$. The statements to check are $p.1$, $p.3$, $p.4$, $p.6$ and $p.8$.

Observe that $p@\{1, 3\} \wedge \text{(I23)} \Rightarrow \neg Y[r, c][p]$ and that statements $p.1$ and $p.3$ do not modify Y. Hence, these statements do not falsify (I23). Note also that $\neg Y[r, c][p] \;\vee\; p.i \neq r \;\vee\; p.j \neq c$ holds after statement $p.8$ is executed (recall that $r + c < k - 1$). For statement $p.4$, we have the following.

$\{p@4 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move \neq right\}$ $p.4$ $\{p@5 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move \neq right\}$
, axiom of assignment.

$\{p@4 \;\wedge\; (p.i \neq r \;\vee\; p.j \neq c \;\vee\; p.move = right) \wedge \text{(I23)}\}$ $p.4$ $\{\neg Y[r, c][p]\}$
, $p@4 \wedge$ (I23) implies $\neg Y[r, c][p]$; $p.4$ does not modify $Y[r, c][p]$ when
$p.i \neq r \;\vee\; p.j \neq c \;\vee\; p.move = right$ holds.

The above assertions imply that statement $p.4$ does not falsify (I23). The following assertions imply that statement $p.6$ does not falsify (I23).

$\{p@6 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move = stop\}$ $p.6$ $\{p@7 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move \neq right\}$
, axiom of assignment.

$\{p@6 \;\wedge\; p.i = r \;\wedge\; p.j = c \;\wedge\; p.move \neq stop\}$ $p.6$ $\{\neg Y[r, c][p]\}$, axiom of assignment.

$\{p@6 \;\wedge\; \neg(p.i = r \;\wedge\; p.j = c) \wedge \text{(I23)}\}$ $p.6$ $\{\neg Y[r, c][p]\}$
, $\neg(p.i = r \;\wedge\; p.j = c) \wedge$ (I23) implies $\neg Y[r, c][p]$; $p.1$ does not establish $Y[r, c][p]$. $\square$

For notational convenience, we define the following predicate. Informally, $EN(p, r, c)$ holds for any p for which $p.i \geq r \;\wedge\; p.j \geq c$ will eventually hold, regardless of the behavior of processes other than p. Note that if the first disjunct holds, then the second disjunct holds after $p.5$ is executed. If the second or third disjunct holds, then $p.i \geq r \;\wedge\; p.j \geq c$ holds after $p.6$ is executed. We use this predicate in (I25) to show that at most one process concurrently accesses a building block that is $k - 1$ steps away from the top left building block in the grid. This shows why a process that takes $k - 1$ steps in the grid can be assigned a name immediately.

Definition: $EN(p, r, c) \equiv (p.i = r - 1 \;\wedge\; p.j \geq c \;\wedge\; p@\{3..5\} \;\wedge\; X[r - 1, p.j] \neq p) \;\vee$
$(p.i = r - 1 \;\wedge\; p.j \geq c \;\wedge\; p@6 \;\wedge\; p.move = down) \;\vee$
$(p.i \geq r \;\wedge\; p.j = c - 1 \;\wedge\; p@\{4, 6\} \;\wedge\; p.move = right) \;\vee$
$(p.i \geq r \;\wedge\; p.j \geq c \;\wedge\; p@\{2..8\})$ $\square$

```
shared variable   X : array[0..k − 2, 0..k − 2] of {⊥} ∪ {0..N − 1};
                  Y : array[0..k − 2, 0..k − 2] of array[0..N − 1] of boolean
initially (∀r, c, p : 0 ≤ r < k − 1  ∧  0 ≤ c < k − 1  ∧  0 ≤ p < N :: X[r, c] = ⊥  ∧  ¬Y[r, c][p])

process p                                                                  /* 0 ≤ p < N */
private variable  name : 0..k(k + 1)/2 − 1;
                  move : {stop, right, down};
                  i, j : 0..k − 1
initially p.i = 0  ∧  p.j = 0  ∧  p.move = down

     while true do
0:      Remainder Section;
1:      i, j, move := 0, 0, down;                    /* Start at top left building block in grid */
        while i + j < k − 1  ∧  move ≠ stop do   /* Move down or across grid until stopping or reaching edge */
2:         X[i, j], h, move := p, 0, stop;
           while h < N  ∧  move ≠ right do
3:            if Y[i, j][h] then move := right else h := h + 1; move := stop fi
           od;
4:         if move ≠ right then
              Y[i, j][p] := true;
5:            if X[i, j] ≠ p then move := down else move := stop fi
           fi;
6:         if move ≠ stop then
              Y[i, j][p] := false;                            /* Reset block if we didn't stop at it */
              if move = down then i := i + 1 else j := j + 1 fi        /* Move according to move */
           fi
        od;
7:      name := i(k − (i − 1)/2) + j;                /* Calculate name based on position in grid */
        Working Section;
8:      if i + j < k − 1 then                                /* If we stopped on a building block ... */
           Y[i, j][p] := false                              /* ... then reset that building block */
        fi
     od
```

Figure 5: Long-lived renaming with $\Theta(k^2)$ name space and $\Theta(Nk)$ time complexity.

invariant $EN(p, r, c) \Rightarrow EN(p, r, c-1) \ \wedge \ EN(p, r-1, c)$ (I24)

Proof: (I24) follows directly from the definition of $EN(p, r, c)$. If either of the first two disjuncts of $EN(p, r, c)$ holds, then that disjunct of $EN(p, r, c-1)$ holds because $c > c-1$. If either of the last two disjunct of $EN(p, r, c)$ holds, then the last disjunct of $EN(p, r, c-1)$ holds. If the first, second, or last disjunct of $EN(p, r, c)$ holds, then the last disjunct of $EN(p, r-1, c)$ holds. Finally, if the third disjunct of $EN(p, r, c)$ holds, then the third disjunct of $EN(p, r-1, c)$ holds because $r > r-1$. □

The following invariant is analogous to (I9).

invariant $r \geq 0 \ \wedge \ c \geq 0 \ \wedge \ r + c \leq k-1 \Rightarrow (|\{s :: EN(s, r, c)\}| \leq k - r - c)$ (I25)

Proof: Assume $r \geq 0 \wedge c \geq 0 \wedge r+c \leq k-1$. Initially $p@0$ holds for all p, so (I25) holds because $r+c \leq k-1$. To prove that (I25) is not falsified, it suffices to consider those statements that may establish $EN(p, r, c)$ for some p. $EN(p, r, c)$ can be established by modifying $p.i$ or $p.j$, or by establishing $X[r-1, p.j] \neq p$, $p@\{3..5\}$, $p@6$, $p@\{4, 6\}$, $p@\{2, 8\}$, $p.move = down$, or $p.move = right$. The statements to check are $p.1$ $p.2$, $p.3$, $p.4$, $p.5$, $p.6$, and $q.2$ for $q \neq p$.

By (I15), if $r+c=0$, then (I25) is an invariant. Henceforth, we assume that $r+c>0$. Statement $p.1$ could only establish the last disjunct of $EN(p,r,c)$. However, $\{p@1 \ \wedge \ r+c>0\}$ $p.1$ $\{p@2 \ \wedge \ (p.i<r \ \vee \ p.j<c)\}$. Thus, statement $p.1$ does not establish $EN(p,r,c)$.

Statement $p.2$ can only establish the first disjunct of $EN(p,r,c)$. However, it only does so if executed when $p.i=r-1 \ \wedge \ p.j \geq c$ holds, in which case it also establishes $X[r-1,p.j]=p$.

Statement $p.4$ does not establish $p@\{3..5\}$, $p@6 \ \wedge \ p.move = down$, or $p@\{4,6\}$, nor does it modify $p.move$, $p.i$, $p.j$, or X.

Statement $p.5$ establishes $p@6 \ \wedge \ p.move \neq right$, and hence can only establish the second disjunct of $EN(p,r,c)$. However, it only does so if executed when $p@5 \ \wedge \ p.i=r-1 \ \wedge \ p.j \geq c \ \wedge \ X[p.i,p.j] \neq p$ holds, in which case $EN(p,r,c)$ already holds.

Statement $p.6$ establishes $p@\{2,7\}$, and hence can only establish the last disjunct of $EN(p,r,c)$. Statement $p.6$ can only establish the last disjunct of $EN(p,r,c)$ if executed when either $p.i=r-1 \ \wedge \ p.j \geq c \ \wedge \ p@6 \ \wedge \ p.move = down$ or $p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ p@6 \ \wedge \ p.move = right$ holds. In either case, $EN(p,r,c)$ already holds.

It remains to consider statements $p.3$ and $q.2$. Statement $p.3$ only can only establish the third disjunct of $EN(p,r,c)$. It only does so if executed when $p@3 \ \wedge \ p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ Y[p.i,p.j][p.h]$ holds. The following assertions imply that (I25) holds after statement $p.3$ is executed in this case.

$$p@3 \ \wedge \ p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ Y[p.i,p.j][p.h] \wedge (\text{I23})^{r,c,p}_{p.i,p.j,p.h} \wedge (\text{I25})^{c}_{c-1}$$

$$\Rightarrow p@3 \ \wedge \ p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ p.i \geq 0 \ \wedge \ p.j \geq 0 \ \wedge \ p.i+p.j<k-1 \ \wedge \ 0 \leq p.h<N \ \wedge$$
$$Y[p.i,p.j][p.h] \wedge (\text{I23})^{r,c,p}_{p.i,p.j,p.h} \wedge (\text{I25})^{c}_{c-1} \qquad \text{, by (I17), (I20), and (I21).}$$

$$\Rightarrow p@3 \ \wedge \ p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ c-1 \geq 0 \wedge (\text{I25})^{c}_{c-1} \ \wedge$$
$$(\exists s : s=p.h \ \wedge \ s \neq p :: s@\{5..8\} \ \wedge \ s.i=p.i \ \wedge \ s.j=p.j \ \wedge \ s.move \neq right)$$
$$\text{, by (I23); note that } p@3 \ \wedge \ s@\{5..8\} \text{ implies } s \neq p.$$

$$\Rightarrow (\exists s : s \neq p :: s@\{5..8\} \ \wedge \ s.i \geq r \ \wedge \ s.j=c-1 \ \wedge \ s.move \neq right) \ \wedge$$
$$(|\{s :: EN(s,r,c-1)\}| \leq k-r-c+1) \qquad \text{, by (I25); recall that } r \geq 0 \text{ and } r+c \leq k-1.$$

$$\{p@3 \ \wedge \ p.i \geq r \ \wedge \ p.j=c-1 \ \wedge \ Y[p.i,p.j][p.h] \wedge (\text{I23})^{r,c,p}_{p.i,p.j,p.h} \wedge (\text{I25})^{c}_{c-1}\} \ p.3 \ \{(\exists s : s \neq p ::$$
$$s@\{5..8\} \ \wedge \ s.i \geq r \ \wedge \ s.j=c-1 \ \wedge \ s.move \neq right) \ \wedge \ (|\{s :: EN(s,r,c-1)\}| \leq k-r-c+1)\}$$

, by above derivation, precondition implies postcondition; $p.3$ does not modify private variables of s; note also that the precondition implies $EN(p,r,c-1)$; $EN(s,r,c-1)$ is not established for $s \neq p$.

$$(\exists s : s \neq p :: s@\{5..8\} \ \wedge \ s.i \geq r \ \wedge \ s.j=c-1 \ \wedge \ s.move \neq right) \ \wedge$$
$$(|\{s :: EN(s,r,c-1)\}| \leq k-r-c+1)$$

$$\Rightarrow (\exists s : s \neq p :: \neg EN(s,r,c) \ \wedge \ EN(s,r,c-1)) \ \wedge \ |\{s :: EN(s,r,c-1)\}| \leq k-r-c+1$$
$$\text{, by the definition of } EN.$$

$$\Rightarrow |\{s :: EN(s,r,c)\}| \leq k-r-c \qquad \text{, by (I24).}$$

The above assertions imply that statement $p.3$ does not falsify (I25). Statement $q.2$ for $q \neq p$ can only establish $EN(p,r,c)$ if executed when $q@2 \wedge q.i=r-1 \wedge q.j=p.j \wedge p.i=r-1 \wedge p.j \geq c \wedge X[r-1,p.j]=p$ holds. The following assertions imply that $q.2$ does not falsify (I25) in this case.

$$q@2 \ \wedge \ q.i=r-1 \ \wedge \ q.j=p.j \ \wedge \ p.i=r-1 \ \wedge \ p.j \geq c \ \wedge \ X[r-1,p.j]=p \wedge (\text{I25})^{r}_{r-1}$$

$$\Rightarrow (\forall s : s \neq p \ \wedge \ s \neq q :: s.j \neq q.j \ \vee \ X[r-1,s.j] \neq s) \ \wedge \ r-1 \geq 0 \wedge (\text{I25})^{r}_{r-1}$$
$$\text{, predicate calculus and (I20).}$$

$\Rightarrow (\forall s : s \neq p \ \wedge \ s \neq q :: s.j \neq q.j \ \vee \ X[r-1, s.j] \neq s) \ \wedge \ (|\{s :: EN(s, r-1, c)\}| \leq k - r - c + 1)$
, definition of (I25); recall that $c \geq 0 \ \wedge \ r + c \leq k - 1$.

$\{q@2 \ \wedge \ q.i = r-1 \ \wedge \ q.j = p.j \ \wedge \ p.i = r-1 \ \wedge \ p.j \geq c \ \wedge \ X[r-1, p.j] = p \wedge (\mathrm{I25})^r_{r-1}\} \ q.2$
$\{q@3 \ \wedge \ q.i = r-1 \ \wedge \ q.j \geq c \ \wedge \ X[q.i, q.j] = q \ \wedge \ (|\{s :: EN(s, r-1, c)\}| \leq k - r - c + 1)$
, by above derivation and axiom of assignment; $q.2$ does not establish $EN(s, r-1, c)$ for any s in this case.

$q@3 \ \wedge \ q.i = r-1 \ \wedge \ q.j \geq c \ \wedge \ X[q.i, q.j] = q \ \wedge \ (|\{s :: EN(s, r-1, c)\}| \leq k - r - c + 1)$

$\Rightarrow \neg EN(q, r, c) \ \wedge \ EN(q, r-1, c) \ \wedge \ |\{s :: EN(s, r-1, c)\}| \leq k - r - c + 1$, by the definition of EN.

$\Rightarrow |\{s :: EN(s, r, c)\}| \leq k - r - c$, by (I24). □

The following invariant is analogous to (I11).

invariant $p \neq q \ \wedge p@\{6..8\} \ \wedge \ p.move = stop \Rightarrow$
$(q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ q@\{0..2\} \ \vee \ (q@3 \ \wedge \ q.h \leq p) \ \vee \ (q@4 \ \wedge \ q.move = right) \ \vee$
$(q@\{3..5\} \ \wedge \ X[q.i, q.j] \neq q) \ \vee \ (q@\{6..8\} \ \wedge \ q.move \neq stop))$ (I26)

Proof: Assume that $p \neq q$. Initially $p@0$ holds, so (I26) holds. To prove that (I26) is not falsified, it suffices to consider statements that potentially establish $p@\{6..8\} \ \wedge \ p.move = stop$, statements that modify $p.i$, or $p.j$, or process q's private variables. The statements to check are $p.5$, $p.6$, and all statements of process q. After the execution of statement $q.0$, $q.1$, $q.2$, or $q.8$, we have $q@\{0..2\} \ \vee \ (q@3 \ \wedge \ q.h \leq p)$. If statement $q.5$ falsifies $q@\{3..5\} \ \wedge \ X[q.i, q.j] \neq q$, then $q@6 \ \wedge \ q.move \neq stop$ holds afterwards. Statements $q.7$ does not falsify $q@\{6..8\} \ \wedge \ q.move \neq stop$. Statement $p.6$ does not establish the antecedent and if the antecedent holds before statement $p.6$ is executed, then $p.6$ does not assign $p.i$ or $p.j$, and hence does not affect the consequent. It remains to consider statements $p.5$, $q.3$, $q.4$, and $q.6$.

$\{p@5 \ \wedge \ (q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ q@\{0..2\})\} \ p.5 \ \{q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ q@\{0..2\}\}$
, precondition implies postcondition, which is unchanged by $p.5$.

$\{p@5 \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j \ \wedge \ q@\{6..8\} \ \wedge \ q.move \neq stop\} \ p.5 \ \{q@\{6..8\} \ \wedge \ q.move \neq stop\}$
, precondition implies postcondition, which is unchanged by $p.5$.

$p@5 \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j \ \wedge \ q@\{6..8\} \ \wedge \ q.move = stop \wedge (\mathrm{I26})^{p,q}_{q,p} \Rightarrow p@5 \ \wedge \ X[p.i, p.j] \neq p$
, by the definition of (I26).

$\{p@5 \ \wedge \ X[p.i, p.j] \neq p\} \ p.5 \ \{p@6 \ \wedge \ p.move \neq stop\}$, axiom of assignment.

$\{p@5 \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j \ \wedge \ q@\{6..8\} \ \wedge \ q.move = stop \wedge (\mathrm{I26})^{p,q}_{q,p}\} \ p.5 \ \{p@6 \ \wedge \ p.move \neq stop\}$
, by two previous assertions.

$\{p@5 \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j \ \wedge \ \neg q@\{0..2, 6..8\} \ \wedge \ X[p.i, p.j] = p\} \ p.5 \ \{q@\{3..5\} \ \wedge \ X[q.i, q.j] \neq q)\}$
, $p \neq q$, so precondition implies postcondition, which is unchanged by $p.5$.

The above assertions imply that $p.5$ does not falsify (I26). For statement $q.3$, we have the following.

$\{q@3 \ \wedge \ (\neg p@\{6..8\} \ \vee \ p.move \neq stop)\} \ q.3 \ \{\neg p@\{6..8\} \ \vee \ p.move \neq stop\}$
, precondition implies postcondition, which is unchanged by $q.3$.

$\{q@3 \ \wedge \ p@\{6..8\} \ \wedge \ p.move = stop \ \wedge \ (q.i \neq p.i \ \vee \ q.j \neq p.j)\} \ q.3 \ \{q.i \neq p.i \ \vee \ q.j \neq p.j\}$
, precondition implies postcondition, which is unchanged by $q.3$.

$\{q@3 \ \wedge \ p@\{6..8\} \ \wedge \ p.move = stop \ \wedge \ q.h = p \ \wedge \ q.i = p.i \ \wedge \ q.j = p.j\} \ q.3$
$\{q@4 \ \wedge \ q.move = right\}$, by (I16), precondition implies $q@3 \ \wedge \ Y[q.i, q.j][q.h]$.

$\{q@3 \ \wedge \ p@\{6..8\} \ \wedge \ p.move = stop \ \wedge \ q.h > p \wedge (\text{I26})\} \ q.3$
$\{q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ (q@\{3..5\} \ \wedge \ X[q.i, q.j] \neq q)\}$
, by definition of (I26), precondition implies postcondition, which is unchanged by $q.3$.

$\{q@3 \ \wedge \ q.h < p\} \ q.3 \ \{(q@3 \ \wedge \ q.h \leq p) \ \vee \ (q@4 \ \wedge \ q.move = right)\}$
, loop at statement 3 either repeats or terminates.

The above assertions imply that $q.3$ does not falsify (I26). For statement $q.4$ we have the following.

$\{q@4 \ \wedge \ q.move = right\} \ q.4 \ \{q@6 \ \wedge \ q.move \neq stop\}$, axiom of assignment.

$\{q@4 \ \wedge \ (\neg p@\{6..8\} \ \vee \ p.move \neq stop)\} \ q.4 \ \{\neg p@\{6..8\} \ \vee \ p.move \neq stop\}$
, precondition implies postcondition, which is unchanged by $q.4$.

$\{q@4 \ \wedge \ p@\{6..8\} \ \wedge \ p.move = stop \ \wedge \ q.move \neq right \wedge (\text{I26})\} \ q.4$
$\{q.i \neq p.i \ \vee \ q.j \neq p.j \ \vee \ (q@\{3..5\} \ \wedge \ X[q.i, q.j] \neq q)\}$
, by definition of (I26), precondition implies postcondition, which is unchanged by $q.4$.

The above assertions imply that statement $q.4$ does not falsify (I26). The following assertions imply that statement $q.6$ does not falsify (I26).

$\{q@6 \ \wedge \ q.move \neq stop\} \ q.6 \ \{q@2 \ \vee \ (q@7 \ \wedge \ q.move \neq stop)\}$, axiom of assignment.

$\{q@6 \ \wedge \ q.move = stop \wedge (\text{I26})^{p,q}_{q,p}\} \ q.6 \ \{\neg p@\{6..8\} \ \vee \ p.move \neq stop \ \vee \ p.i \neq q.i \ \vee \ p.j \neq q.j\}$
, by definition of (I26), precondition implies postcondition;
postcondition is unchanged by $q.6$ because $q.move = stop$. □

invariant $p \neq q \ \wedge \ p@\{7..8\} \ \wedge \ q@\{7..8\} \Rightarrow p.i \neq q.i \ \vee \ q.j \neq q.j$ (I27)

Proof: If $p \neq q \wedge p@\{7..8\} \wedge q@\{7..8\} \wedge p.move = stop$ holds, then by (I17), (I18), and (I26), the consequent holds. If $p \neq q \ \wedge \ p@\{7..8\} \ \wedge \ q@\{7..8\} \ \wedge \ p.move \neq stop$ holds, then by (I17), (I18), and $(\text{I25})^{r,c}_{p.i,p.j}$, it follows that $|\{s :: EN(s, p.i, p.j)\}| \leq 1$. By the definition of EN, $p@\{7..8\}$ implies $EN(p, p.i, p.j)$. Therefore, $EN(q, p.i, p.j)$ does not hold, which implies that the consequent holds. □

invariant $p \neq q \ \wedge \ p@8 \ \wedge \ q@8 \Rightarrow p.name \neq q.name$ (I28)

Proof: The following derivation implies that (I28) is an invariant.

$p \neq q \ \wedge \ p@8 \ \wedge \ q@8 \ \wedge \ p.name = q.name$

$\Rightarrow p@8 \ \wedge \ q@8 \ \wedge \ p.name = q.name \ \wedge \ (p.i \neq q.i \ \vee \ p.j \neq q.j) \ \wedge \ p.i + p.j \leq k-1 \ \wedge \ q.i + q.j \leq k-1$
, by (I19) and (I27).

$\Rightarrow (p.i \neq q.i \ \vee \ p.j \neq q.j) \ \wedge \ p.i + p.j \leq k-1 \ \wedge \ q.i + q.j \leq k-1 \ \wedge \ p.i \geq 0 \ \wedge \ p.j \geq 0 \ \wedge$
$q.i \geq 0 \ \wedge \ q.j \geq 0 \ \wedge \ p.i(k - (p.i - 1)/2) + p.j = q.j(k - (q.j - 1)/2) + q.j$, by (I20) and (I22).

$\Rightarrow$ *false* , by Claim 1 (Section 3.3) with $c = p.i$, $d = p.j$, $c' = q.i$, and $d' = q.j$. □

invariant $p@5 \Rightarrow 0 \leq p.name < k(k+1)/2$ (I29)

Proof: (I29) follows from (I19), (I20), (I22), and Claim 2 (Section 3.3). □

(I28) and (I29) prove that the algorithm shown in Figure 5 correctly implements long-lived k-renaming. To see that the wait-freedom requirement is satisfied, consider the two loops in Figure 5. The inner loop clearly terminates after at most N iterations. To see that the outer loop terminates, consider statement $p.4$. If $p.move = right$ holds before statement $p.4$ is executed, then $p.j$ is incremented when statement $p.6$ is executed. Otherwise, statement $p.5$ establishes either $p.move = stop$ or $p.move = down$. In the first case, the outer loop terminates. In the second case, $p.i$ is incremented when statement $p.6$ is executed. Because of the loop condition $p.i + p.j < k - 1$, the outer loop is therefore executed at most $k - 1$ times. The inner loop executes at most N shared references, and the outer loop executes at most four more. Releasing a name causes at most one more shared access. Thus, we have the following result.

Theorem 3: Using *read* and *write*, wait-free, long-lived $(k(k+1)/2)$-renaming can be implemented with time complexity $(N + 4)(k - 1) + 1 = \Theta(Nk)$. □

5 Long-Lived Renaming using Read-Modify-Writes

In this section, we present three wait-free, long-lived renaming algorithms and one lock-free, long-lived algorithm. By using read-modify-write operations, these algorithms significantly improve upon the performance of the algorithms in the previous section. Furthermore, these algorithms yield a name space of size k, which is clearly optimal (the lower bound results of [7] do not apply to algorithms that employ read-modify-write operations).

The first algorithm uses *set_first_zero* and *clr_bit* and has time complexity $\Theta(k/b)$. As discussed in Section 1, these operations can be implemented, for example, using operations available on the BBN TC2000 [4]. The second algorithm in this section has time complexity $\Theta(\log k)$ — a significant improvement over the first algorithm. To achieve this improvement, this algorithm uses the *bounded_decrement* operation. We then describe how the techniques from these two algorithms can be combined to obtain an algorithm whose time complexity is better than that of either algorithm.

We do not know of any systems that provide *bounded_decrement* as a primitive operation. However, at the end of this section, we discuss how the *bounded_decrement* operation can be approximated in a lock-free manner using the commonly-available *fetch_and_add* operation. We show how this approximation can be used to provide a lock-free algorithm for long-lived k-renaming.

5.1 Long-Lived Renaming using *set_first_zero* and *clr_bit*

Our first long-lived k-renaming algorithm employs the *set_first_zero* and *clr_bit* operations. The algorithm is shown in Figure 6. For clarity, we have expanded the definition of *set_first_zero* (see statement 1). In order to acquire a name, a process tests each name in order. Using the *set_first_zero* operation, up to b names can be tested in one atomic shared variable access, where b is the number of bits per shared variable. If $k \leq b$, this results in a long-lived renaming algorithm that acquires a name with just one shared variable access. If $k > b$, then "segments" of size b of the name space are tested in each access. To release a name, a process clears the bit that was set by that process when the name was acquired. An example is shown in Figure 7 for $b = 4$ and $k = 10$. In this figure, process p releases name 1 by executing *clear_bit*$(X[0], 1)$ and process q acquires name 5 by executing *set_first_zero*$(X[1])$.

```
shared variable X : array[0..⌊k/b⌋] of array[0..b − 1] of boolean        /* b-bit "segments" of the name space */
initially (∀i, j : 0 ≤ i ≤ ⌊k/b⌋ ∧ 0 ≤ j < b :: ¬X[i][j])

process p                                                                         /* 0 ≤ p < N */
private variable h : 0..⌊k/b⌋ + 1; v : 0..b; name : 0..k − 1

     while true do
0:       Remainder Section;
         h, v := 0, b;                                  /* Initialize h and v after remainder section */
         while v = b do                                                    /* Loop until a bit is set */
1:           if (∃n : 0 ≤ n < b :: ¬X[h][n]) then       /* set_first_zero operation, as defined in Section 2 */
                 m := (min n : 0 ≤ n < b :: ¬X[h][n]); X[h][m], v := true, m
             else
                 v := b
             fi;
             h := h + 1;
         od;
2:       name := b(h − 1) + v;                                                   /* Calculate name */
         Working Section;
3:       X[h − 1][v] := false                                           /* Clear the bit that was set */
     od
```

Figure 6: Long-lived k-renaming using *set_first_zero* and *clear_bit*.

Because each process tests the available names in segments, and because processes may release and acquire names concurrently, it may seem possible for a process to reach the last segment when none of the names in that segment are available. In fact, this is not possible, as is shown by the following correctness proof.

5.2 Correctness Proof

In accordance with the problem specification, we assume the following invariant.

invariant $|\{p :: p@\{1..3\}\}| \leq k$ (I30)

The following invariants follow directly from the program text in Figure 6, and are stated without proof. Note that (I32) is used to prove (I33).

invariant $p@3 \Rightarrow p.name = b(p.h - 1) + p.v$ (I31)

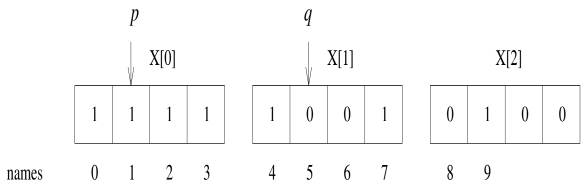
invariant $p.h \geq 0$ (I32)

invariant $p@\{2..3\} \Rightarrow 0 \leq p.v < b \;\wedge\; p.h > 0$ (I33)

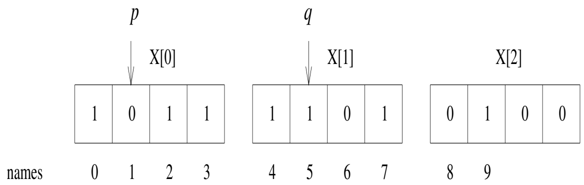
Formal correctness proofs are given below for the remaining invariants. We first prove that the conjunction of the following two assertions is an invariant. These assertions show that two processes do not concurrently "hold" the same bit.

$q \neq p \;\wedge\; q@\{2..3\} \;\wedge\; p@\{2..3\} \;\wedge\; 0 \leq p.h \leq \lceil k/b \rceil \Rightarrow q.h \neq p.h \;\vee\; q.v \neq p.v$ (A1)

$0 \leq i \leq \lfloor k/b \rfloor \;\wedge\; 0 \leq j < b \Rightarrow (X[i][j] = (\exists r :: r@\{2,3\} \;\wedge\; r.h = i+1 \;\wedge\; r.v = j))$ (A2)



(a) In this state, $p@3 \ \wedge \ p.h = 1 \ \wedge \ p.v = 1$ holds, so p is about to execute *clear_bit*$(X[0], 1)$, thereby releasing name 1. For process q, $q@1 \ \wedge \ q.h = 1$ holds, so q is about to execute *set_first_zero*$(X[1])$. As $X[1][1]$ is the first clear bit in $X[1]$, $q.1$ will establish $q@2 \ \wedge \ q.h = 2 \ \wedge \ q.v = 1$, and will therefore acquire name 5.



(b) Process p has released name 1 and process q has acquired name 5.

Figure 7: Example steps of the k-renaming algorithm shown in Figure 6 for $b = 4$ and $k = 10$.

invariant (A1) $\wedge$ (A2) (I34)

Proof: Initially $(\forall p :: p@0) \ \wedge \ \neg X[i][j]$ holds, so (I34) holds. We first consider statements that potentially falsify (A1). Assume that $q \neq p$. By symmetry, we need only consider statements that may establish $q@\{2..3\}$ or modify $q.h$ or $q.v$. The statements to check are $q.0$ and $q.1$. The antecedent does not hold after $q.0$ is executed. For $q.1$, we have the following.

$\{q@1 \ \wedge \ (\neg p@\{2..3\} \ \vee \ q.h + 1 \neq p.h \ \vee \ p.h < 0 \ \vee \ p.h > \lceil k/b \rceil)\} \ q.1$
$\quad \{\neg p@\{2..3\} \ \vee \ q.h \neq p.h \ \vee \ p.h < 0 \ \vee \ p.h > \lceil k/b \rceil\}$, axiom of assignment $(q \neq p)$.

$\quad q@1 \ \wedge \ p@\{2..3\} \ \wedge \ q.h + 1 = p.h \ \wedge \ 0 \leq p.h \leq \lceil k/b \rceil \wedge (\text{A2})^{i,j}_{q.h,p.v}$

$\Rightarrow p@\{2..3\} \ \wedge \ q.h + 1 = p.h \ \wedge \ 0 \leq q.h \leq \lfloor k/b \rfloor \ \wedge \ 0 \leq p.v < b \wedge (\text{A2})^{i,j}_{q.h,p.v}$, by (I33).

$\Rightarrow X[q.h, p.v] \ \wedge \ p.v \neq b$, by definition of (A2).

$\{q@1 \ \wedge \ p@\{2..3\} \ \wedge \ q.h + 1 = p.h \wedge (\text{A2})^{i,j}_{q.h,p.v}\} \ q.1 \ \{q.v \neq p.v\}$
, by above derivation and definition of *set_first_zero*.

For (A2), assume that $0 \leq i \leq \lfloor k/b \rfloor \ \wedge \ 0 \leq j < b$. (A2) can be falsified by statements that modify X, establish or falsify $r@\{2..3\}$, or modify $r.h$ or $r.v$ for some r. The statements to check are $r.0$, $r.1$, and $r.3$. Statement $r.0$ does not modify X; also $r@\{2,3\}$ (and hence $r@\{2,3\} \ \wedge \ r.h = i \ \wedge \ r.v = j$) is false both before and after the execution of $r.0$. For statement $r.1$, consider the following assertions.

$\{r@1 \ \wedge \ r.h \neq i \wedge (\text{A2})\} \ r.1 \ \{r.h \neq i + 1 \wedge (\text{A2})\}$
, $r.1$ does not modify $X[i][j]$ because $r.h \neq i$;
also pre- and post-conditions imply $\neg(r@\{2..3\} \ \wedge \ r.h = i + 1 \ \wedge \ r.v = j)$.

$\{r@1 \ \wedge \ r.h = i \ \wedge \ X[i][j] \wedge (\text{A2})\} \ r.1 \ \{X[i][j] \ \wedge \ (\exists s : s \neq r :: s@\{2,3\} \ \wedge \ s.h = i + 1 \ \wedge \ s.v = j)\}$
, by definition of (A2), precondition implies postcondition, which is not falsified by $r.1$;

note that $r@1 \ \wedge \ s@\{2,3\}$ implies $s \neq r$.

$\{r@1 \ \wedge \ r.h = i \ \wedge \ \neg X[i][j] \ \wedge \ (\exists n : 0 \leq n < j :: \neg X[i][n]) \wedge (\text{A2})\} \ r.1 \ \{\neg X[i][j] \ \wedge \ r.v < j \wedge (\text{A2})\}$
, $r.1$ does not modify $X[i][j]$ or establish $r.v = j$ in this case.

$\{r@1 \ \wedge \ r.h = i \ \wedge \ \neg X[i][j] \ \wedge \ (\forall n : 0 \leq n < j :: X[i][n])\} \ r.1 \ \{X[i][j] \ \wedge \ r@2 \ \wedge \ r.h = i+1 \ \wedge \ r.v = j\}$
, by definition of *set_first_zero*, $X[i][j] \ \wedge \ r.v = j$ is established; loop terminates because $j < b$.

The above assertions imply that $r.1$ does not falsify (A2). The following assertions imply that $r.3$ does not falsify (A2).

$\{r@3 \ \wedge \ (r.h \neq i+1 \ \vee \ r.v \neq j) \wedge (\text{A2})\} \ r.3 \ \{r@0 \ \wedge \ (r.h \neq i+1 \ \vee \ r.v \neq j) \wedge (\text{A2})\}$
, $r.3$ does not modify $r.h$, $r.v$, or $X[i][j]$ in this case.

$\{r@3 \ \wedge \ r.h = i+1 \ \wedge \ r.v = j \wedge (\text{A1})\} \ r.3$
$\{\neg X[i][j] \ \wedge \ r@0 \ \wedge \ (\forall s : s \neq r :: \neg s@\{2..3\} \ \vee \ s.h \neq i+1 \ \vee \ s.v \neq j)\}$
, because $0 \leq i \leq \lfloor k/b \rfloor$, the precondition implies that $0 < r.h \leq \lceil k/b \rceil$; thus, by definition of (A1), precondition implies $(\forall s : s \neq r :: \neg s@\{2..3\} \ \vee \ s.h \neq i+1 \ \vee \ s.v \neq j)$, which is not falsified by $r.3$; also, $r.3$ establishes $\neg X[i][j] \ \wedge \ r@0$ in this case. □

The following assertion shows that there are always enough names left for the number of processes seeking names from $X[i] ... X[\lfloor k/b \rfloor]$.

invariant $0 \leq i \leq \lfloor k/b \rfloor \Rightarrow (|\{p :: (p@1 \ \wedge \ p.h \geq i) \ \vee \ (p@\{2..3\} \ \wedge \ p.h > i)\}| \leq k - ib)$ (I35)

Proof: By (I30), (I35) holds if $i = 0$. Henceforth, assume $0 < i \leq \lfloor k/b \rfloor$. Initially $(\forall p :: p@0)$ holds, and because $i \leq \lfloor k/b \rfloor$, it follows $k - ib \geq 0$, so (I35) holds initially. (I35) can only be falsified by establishing $(q@1 \ \wedge \ q.h \geq i) \ \vee \ (q@\{2..3\} \ \wedge \ q.h > i)$ for some process q. The statements to check are $q.0$ and $q.1$. After statement $q.0$ is executed, $q.h < i$ holds because $i > 0$. Statement $q.1$ can only establish $q@\{2..3\} \ \wedge \ q.h > i$ (the second disjunct) if executed when $q@1 \ \wedge \ q.h \geq i$ (ie. the first disjunct already holds). Thus, statement $q.1$ can only establish $(q@1 \ \wedge \ q.h \geq i) \ \vee \ (q@\{2..3\} \ \wedge \ q.h > i)$ if executed when $q@1 \ \wedge \ q.h = i-1$ holds. The following assertions imply that $q.1$ does not falsify (I35) in this case.

$\{q@1 \ \wedge \ q.h = i-1 \ \wedge \ (\exists n : 0 \leq n < b :: \neg X[i-1][n])\} \ q.1 \ \{q@2 \ \wedge \ q.h = i \ \wedge \ q.v < b\}$
, definition of *set_first_zero*; loop terminates because $q.v < b$.

$q@1 \ \wedge \ q.h = i-1 \ \wedge \ (\forall n : 0 \leq n < b :: X[i-1][n] \wedge (\text{I34})^{i,j}_{i-1,n}) \wedge (\text{I35})^{i}_{i-1}$

$\Rightarrow q@1 \ \wedge \ q.h = i-1 \ \wedge \ |\{p :: p@\{2..3\} \ \wedge \ p.h = i\}| \geq b \ \wedge \ 0 \leq i-1 < \lfloor k/b \rfloor \wedge (\text{I35})^{i}_{i-1}$
, (I34) implies (A2); recall that $0 < i \leq \lfloor k/b \rfloor$; thus $0 \leq i-1 < \lfloor k/b \rfloor$.

$\Rightarrow q@1 \ \wedge \ q.h = i-1 \ \wedge \ |\{p :: p@\{2..3\} \ \wedge \ p.h = i\}| \geq b \ \wedge$
$|\{p :: (p@1 \ \wedge \ p.h \geq i-1) \ \vee \ (p@\{2..3\} \ \wedge \ p.h > i-1)| \leq k - ib + b$, definition of (I35).

$\Rightarrow q@1 \ \wedge \ q.h = i-1 \ \wedge \ |\{p :: (p@1 \ \wedge \ p.h \geq i-1) \ \vee \ (p@\{2..3\} \ \wedge \ p.h > i)| \leq k - ib$
, predicate calculus; note that $p.h = i \Rightarrow p.h > i-1 \ \wedge \ \neg(p.h > i)$.

$\Rightarrow |\{p :: (p@1 \ \wedge \ p.h \geq i) \ \vee \ (p@\{2..3\} \ \wedge \ p.h > i)\}| \leq k - ib - 1$
, predicate calculus; note that $q.h = i-1 \Rightarrow q.h \geq i-1 \ \wedge \ \neg(q.h \geq i)$.

$\{q@1 \ \wedge \ q.h = i-1 \ \wedge \ (\forall n : 0 \leq n < b :: X[i-1][n] \wedge (\text{I34})^{i,j}_{i-1,n}) \wedge (\text{I35})^{i}_{i-1}\} \ q.1 \ \{(\text{I35})\}$
, by above derivation; $q.1$ does not establish $(p@1 \ \wedge \ p.h \geq i) \ \vee \ (p@\{2,3\} \ \wedge \ p.h > i)$ for $p \neq q$. □

The following invariant shows that if a process reaches $X[\lfloor k/b \rfloor]$, then its *set_first_zero* will succeed, so it will acquire a name.

invariant $p@1 \ \wedge \ p.h = \lfloor k/b \rfloor \Rightarrow (\exists n : 0 \leq n < k - b\lfloor k/b \rfloor :: \neg X[p.h][n])$ (I36)

Proof: Consider the following derivation.

$$p@1 \ \wedge \ p.h = \lfloor k/b \rfloor \wedge (\mathrm{I35})^i_{\lfloor k/b \rfloor}$$

$$\Rightarrow p@1 \ \wedge \ p.h = \lfloor k/b \rfloor \ \wedge \ (|\{p :: (p@1 \ \wedge \ p.h \geq \lfloor k/b \rfloor) \ \vee \ (p@\{2..3\} \ \wedge \ p.h > \lfloor k/b \rfloor)\}| \leq k - b\lfloor k/b \rfloor)$$
, by (I35).

$$\Rightarrow p.h = \lfloor k/b \rfloor \ \wedge \ (|\{p :: p@\{2..3\} \ \wedge \ p.h = \lfloor k/b \rfloor + 1\}| < k - b\lfloor k/b \rfloor)$$
, predicate calculus.

$$\Rightarrow p.h = \lfloor k/b \rfloor \ \wedge \ (|\{n : 0 \leq n < k - b\lfloor k/b \rfloor :: X[\lfloor k/b \rfloor][n]\}| < k - b\lfloor k/b \rfloor)$$
, observe that $0 \leq n < k - b\lfloor k/b \rfloor$ implies $0 \leq n < b$; thus by (I34), $(|\{n : 0 \leq n < k - b\lfloor k/b \rfloor :: X[\lfloor k/b \rfloor][n]\}|) \leq (|\{p :: p@\{2..3\} \ \wedge \ p.h = \lfloor k/b \rfloor + 1\}|)$.

$$\Rightarrow p.h = \lfloor k/b \rfloor \ \wedge \ (\exists n : 0 \leq n < k - b\lfloor k/b \rfloor :: \neg X[\lfloor k/b \rfloor][n])$$
, pigeonhole principle.

$$\Rightarrow (\exists n : 0 \leq n < k - b\lfloor k/b \rfloor :: \neg X[p.h][n])$$
, predicate calculus. □

invariant $p@1 \Rightarrow 0 \leq p.h \leq \lfloor k/b \rfloor$ (I37)

Proof: Initially $p@0$ holds, so (I37) holds. Only statements $p.0$ and $p.1$ affect (I37). Because $k > 1$ and $b > 0$, (I37) holds after $p.0$ is executed. Statement $p.1$ can only falsify (I37) if executed when $p.h = \lfloor k/b \rfloor$. However, by (I36) and the definition of *set_first_zero*, we have $\{p@1 \ \wedge \ p.h = \lfloor k/b \rfloor\}\ p.1\ \{p@2\}$. □

invariant $p@\{2,3\} \Rightarrow 0 < p.h \leq \lfloor k/b \rfloor \ \vee \ (p.h = \lfloor k/b \rfloor + 1 \ \wedge \ 0 \leq p.v < k - b\lfloor k/b \rfloor)$ (I38)

Proof: Initially, $p@0$ holds, so (I38) holds. Only statements $p.0$ and $p.1$ potentially falsify (I38). The antecedent does not hold after $p.0$ is executed. For $p.1$ we have the following.

$$p@1 \ \wedge \ (p.h < 0 \ \vee \ p.h > \lfloor k/b \rfloor) \Rightarrow \mathit{false}$$
, by (I37).

$$\{p@1 \ \wedge \ 0 \leq p.h < \lfloor k/b \rfloor\}\ p.1\ \{0 < p.h \leq \lfloor k/b \rfloor\}$$
, axiom of assignment.

$$\{p@1 \ \wedge \ p.h = \lfloor k/b \rfloor\}\ p.1\ \{p@2 \ \wedge \ p.h = \lfloor k/b \rfloor + 1 \ \wedge \ 0 \leq p.v < k - b\lfloor k/b \rfloor\}$$
, by (I36) and definition of *set_first_zero*. □

Claim 3: Let c, d, c', and d' be nonnegative integers satisfying $(c \neq c' \ \vee \ d \neq d') \ \wedge \ 0 \leq d < b \ \wedge \ 0 \leq d' < b$. Then, $b(c-1) + d \neq b(c'-1) + d'$.

Proof: The claim is straightforward if $c = c'$, so assume that $c \neq c'$. Without loss of generality assume that $c < c'$. Then,

$$\begin{aligned} b(c-1) + d &< bc && , d < b. \\ &\leq b(c'-1) && , c < c'. \\ &\leq b(c'-1) + d' && , d' \geq 0. \end{aligned}$$ □

invariant $p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \Rightarrow p.name \neq q.name$ (I39)

Proof: Consider the following derivation.

$p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \ \wedge \ p.name = q.name$

$\Rightarrow p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \ \wedge \ p.name = q.name \ \wedge \ 0 < p.h \leq \lceil k/b \rceil$, by (I38).

$\Rightarrow p \neq q \ \wedge \ p@3 \ \wedge \ q@3 \ \wedge \ p.name = q.name \ \wedge \ (q.h \neq p.h \ \vee \ q.v \neq p.v)$, by (A1) ((I34) implies (A1)).

$\Rightarrow (b(p.h-1) + p.v = b(q.h-1) + q.v) \ \wedge \ (q.h \neq p.h \ \vee \ q.v \neq p.v) \ \wedge$
$0 \leq p.v < b \ \wedge \ 0 \leq q.v < b \ \wedge \ p.h \geq 0 \ \wedge \ q.h \geq 0$, by (I31), (I32), and (I33).

$\Rightarrow false$, by Claim 3 with $c = p.h$, $d = p.v$, $c' = q.h$, and $d' = q.v$. □

This concludes the proof that no two processes in their working sections have the same name. The following invariant shows that that each process acquires a name ranging over $0..k-1$.

invariant $p@3 \Rightarrow 0 \leq p.name < k$ (I40)

Proof: Initially $p@0$ holds, so (I40) holds. Only statement $p.2$ potentially falsifies (I40). The following assertions imply that statement $p.2$ does not falsify (I40).

$p@2 \ \wedge \ (p.h \leq 0 \ \vee \ p.h > \lfloor k/b \rfloor + 1) \Rightarrow false$, by (I38).

$p@2 \ \wedge \ 0 < p.h < \lfloor k/b \rfloor + 1$

$\Rightarrow p@2 \ \wedge \ 0 < p.h \leq \lfloor k/b \rfloor$, predicate calculus.

$\Rightarrow (0 \leq b(p.h-1) \leq k-b) \ \wedge \ (0 \leq p.v < b)$, by (I33) and predicate calculus.

$\Rightarrow 0 \leq (b(p.h-1) + p.v) < k$, predicate calculus.

$\{p@2 \ \wedge \ 0 < p.h < \lfloor k/b \rfloor + 1\} \ p.2 \ \{0 \leq p.name < k\}$, above derivation and axiom of assignment.

$p@2 \ \wedge \ p.h = \lfloor k/b \rfloor + 1$

$\Rightarrow (p.h = \lfloor k/b \rfloor + 1) \ \wedge \ (0 \leq p.v < k - b\lfloor k/b \rfloor)$, by (I38) and predicate calculus.

$\Rightarrow 0 \leq (b(p.h-1) + p.v) < (b\lfloor k/b \rfloor + k - b\lfloor k/b \rfloor)$, predicate calculus, $b > 0$, $k > 0$.

$\Rightarrow 0 \leq (b(p.h-1) + p.v) < k$, predicate calculus.

$\{p@2 \ \wedge \ p.h = \lfloor k/b \rfloor + 1\} \ p.2 \ \{0 \leq p.name < k\}$, above derivation and axiom of assignment. □

(I39) and (I40) prove that the algorithm shown in Figure 6 correctly implements long-lived k-renaming. Observe that $p.h$ is incremented each time a shared variable is accessed when acquiring a name, and by (I37), this occurs at most $\lceil k/b \rceil$ times before the loop terminates. Also, releasing a name takes one shared variable access. Thus, we have the following result.

Theorem 4: Using *set_first_zero* and *clr_bit* on b-bit variables, wait-free, long-lived k-renaming can be implemented with time complexity $\lceil k/b \rceil + 1$. □

As discussed in Section 1, when $b = 1$, the *set_first_zero* and *clr_bit* operations are equivalent to the *test_and_set* and *write* operations, respectively. Thus, we have the following.

Corollary 1: Using *test_and_set* and *write*, wait-free, long-lived k-renaming can be implemented with time complexity $k+1$. □

5.3 Long-Lived Renaming using *bounded_decrement* and *atomic_add*

In this section, we present a long-lived k-renaming algorithm that employs the *bounded_decrement* and *atomic_add* operations. In this algorithm, shown in Figure 8, the *bounded_decrement* operation is used to separate processes into two groups *left* and *right*. The right group contains at most $\lceil k/2 \rceil$ processes and the left group contains at most $\lfloor k/2 \rfloor$ processes. This is achieved by initializing a shared variable X to $\lceil k/2 \rceil$, and having each process perform a *bounded_decrement* operation on X. Processes that receive positive return values join the right group, and processes that receive zero join the left group. To leave the right group, a process increments X. To leave the left group, no shared variables are updated.

Because processes must be able to repeatedly join and leave the groups, the normal *fetch_and_add* operation is not suitable for this "splitting" mechanism. If X is decremented below zero, then it is possible for too many processes to be in the left group at once. To see this, suppose that all k processes decrement X. Thus, $\lceil k/2 \rceil$ processes receive positive return values, and therefore join the right group, and $\lfloor k/2 \rfloor$ processes receive non-positive return values, and therefore join the left group. Now, $X = -\lfloor k/2 \rfloor$. If a process leaves the right group by incrementing X, and then decrements X as the result of another call to *Getname*(), then that process receives a non-positive return value, and thus joins the left group. Repeating this for each process in the right group, it is possible for all processes to be in the left group simultaneously. The *bounded_decrement* operation prevents this by ensuring that X does not become negative.

The algorithm employs an instance of long-lived $\lceil k/2 \rceil$-renaming for the right group, and an instance of long-lived $\lfloor k/2 \rfloor$-renaming for the left group, which are inductively assumed to be correct. For notational convenience, we assume that the left instance is accessed by calling the *Getname_left* and *Putname_left* procedures; similarly for the right instance. The algorithm that results from "unfolding" this inductively-defined algorithm forms a tree. To acquire a name, a process goes down a path in this tree from the root to a leaf. As the processes progress down the tree, the number of processes that can simultaneously go down the same path is halved at each level. When this number becomes one, a name can be assigned. Thus, the *Getname*() procedure has time complexity $\lceil \log_2 k \rceil$. To release a name, a process retraces the path it took through the tree in reverse order, incrementing X at any node at which it received a positive return value.

Note that with b-bit variables, if $b < \log_2 \lceil k/2 \rceil$, then X cannot be initialized to $\lceil k/2 \rceil$, so this algorithm cannot be implemented. However, in any practical setting, this will not be the case. In the next section, we prove the algorithm shown in Figure 8 correct.

5.4 Correctness Proof

We inductively assume correctness for the right instance of $\lceil k/2 \rceil$-renaming and the left instance of $\lfloor k/2 \rfloor$-renaming. In accordance with the problem specification, we assume that the following invariant holds.

invariant $|\{p :: p@\{1..7\}\}| \leq k$ (I41)

The following two invariants follow directly from the program text in Figure 8.

invariant $p@\{5..6\} \Rightarrow p.side = right$ (I42)

invariant $p@7 \Rightarrow p.side \neq right$ (I43)

Proofs for the remaining invariants are provided. We first prove that the conjunction of the following two assertions is an invariant. These assertions are used to prove that too many processes do not access the left and right instances. This is required so that the correctness of these instances can be used to prove the algorithm correct inductively.

```
shared variable X : 0..⌈k/2⌉                                      /* Counter of names available on right */
initially X = ⌈k/2⌉

process p                                                                                /* 0 ≤ p < N */
private variable side : {left, right}

    while true do
0:     Remainder Section;
1:     if bounded_decrement(X) > 0 then    /* Ensure at most ⌈k/2⌉ access right and at most ⌊k/2⌋ access left */
2:        side, name := right, Getname_right()                            /* Get name from right instance */
       else
3:        side, name := left, ⌈k/2⌉ + Getname_left()                        /* Get name from left instance */
       fi;
       Working Section;
4:     if side = right then
5:        Putname_right(name);                                           /* Return name to right instance */
6:        atomic_add(X, 1)                                                     /* Increment counter again */
       else
7:        Putname_left(name − ⌈k/2⌉)                                       /* Return name to left instance */
       fi
    od
```

Figure 8: k-renaming using *bounded_decrement*. *Getname_left* and *Putname_left* are inductively assumed to implement long-lived $\lfloor k/2 \rfloor$-renaming. Similarly, *Getname_right* and *Putname_right* are inductively assumed to implement long-lived $\lceil k/2 \rceil$-renaming.

$$0 \leq X \leq \lceil k/2 \rceil \tag{A3}$$

$$|\{p :: p@2 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = right)\}| = \lceil k/2 \rceil - X \tag{A4}$$

invariant (A3) $\wedge$ (A4) (I44)

Proof: Initially (A3) $\wedge$ (A4) holds. (A3) can only be falsified by decrementing X when $X = 0$ holds, or by incrementing X when $X = \lceil k/2 \rceil$ holds. By the definition of *bounded_decrement*, the first case does not arise. Only statement $p.6$ increments X. However, consider the following.

$p@6 \ \wedge \ X = \lceil k/2 \rceil \ \wedge \ p.side \neq right \wedge \text{(I42)} \Rightarrow false$, by (I42).

$p@6 \ \wedge \ X = \lceil k/2 \rceil \ \wedge \ p.side = right \wedge \text{(A4)} \Rightarrow false$, by definition of (A4).

(A4) is potentially falsified by any statement that modifies $p.side$ or X, or establishes or falsifies $p@2$ or $p@\{4..7\}$. The statements to check are $p.1$, $p.2$, $p.3$, $p.6$, and $p.7$ where p is any process. Statement $p.2$ preserves $p@2 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = right)$ and statement $p.3$ preserves $\neg(p@2 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = right))$. Also, neither statement modifies X. By (I42), statement $p.6$ decreases both sides of (A4) by 1. By (I43), statement $p.7$ does not affect either side. The following assertions imply that statement $p.1$ does not falsify (A4).

$p@1 \ \wedge \ X < 0 \wedge \text{(A3)} \Rightarrow false$, definition of (A3).

$\{p@1 \ \wedge \ X = 0 \wedge \text{(A4)}\} \ p.1 \ \{p@3 \wedge \text{(A4)}\}$, by definition of *bounded_decrement*, $p.1$ does not modify X.

$\{p@1 \ \wedge \ X > 0 \wedge \text{(A4)}\} \ p.1 \ \{p@2 \wedge \text{(A4)}\}$, both sides of (A4) are increased by 1 in this case. □

invariant $|\{p :: p@2 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = right)\}| \leq \lceil k/2 \rceil$ (I45)

Proof: (I45) follows directly from (I44). □

invariant $|\{p :: p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)\}| \leq \lfloor k/2 \rfloor$ (I46)

Proof: Initially, $(\forall p :: p@0)$ holds, so (I46) holds because $k > 0$. (I46) is potentially falsified by any statement that establishes $p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)$ for some p. The statements to check are $p.1$, $p.2$, and $p.3$. For statement $p.2$, we have $\{p@2\}\ p.2\ \{p@4 \ \wedge \ p.side = right\}$. Statement $p.3$ preserves $p@3 \ \vee \ (p@\{4..7\} \ \wedge \ p.side = left)$. The following assertions imply that statement $p.1$ does not falsify (I46).

$p@1 \ \wedge \ X < 0 \Rightarrow false$, by (A3) ((I44) implies (A3)).

$\{p@1 \ \wedge \ X > 0\}\ p.1\ \{p@2\}$, definition of *bounded_decrement*.

$p@1 \ \wedge \ X = 0$

$\Rightarrow p@1 \ \wedge \ |\{q :: q@2 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = right)\}| = \lceil k/2 \rceil$, by (A4) ((I44) implies (A4)).

$\Rightarrow |\{q :: q@3 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = left)\}| < \lfloor k/2 \rfloor$, by (I41).

$\{p@1 \ \wedge \ X = 0\}\ p.1\ \{|\{q :: q@3 \ \vee \ (q@\{4..7\} \ \wedge \ q.side = left)\}| \leq \lfloor k/2 \rfloor$
, by preceding derivation; $p.1$ increases the left-hand side of (I46) by at most 1. □

By (I45) and (I46), the right instance is accessed by at most $\lceil k/2 \rceil$ processes concurrently and the left instance is accessed by at most $\lfloor k/2 \rfloor$ processes concurrently. Thus, by the assumption that these instances are correct, we have the following invariants.

invariant $p@\{4,5\} \ \wedge \ p.side = right \Rightarrow 0 \leq p.name < \lceil k/2 \rceil$ (I47)

invariant $p@\{4,7\} \ \wedge \ p.side = left \Rightarrow \lceil k/2 \rceil \leq p.name < k$ (I48)

invariant $p \neq q \ \wedge \ p@\{4..7\} \ \wedge \ q@\{4..7\} \ \wedge \ p.side = q.side \Rightarrow p.name \neq q.name$ (I49)

Correctness of the k-renaming algorithm shown in Figure 8 follows from (I47), (I48), and (I49). Note that, given the assumption that the left and right instances are correct, wait-freedom is trivial. This allows us to prove the following result.

Theorem 5: Using b-bit variables and *bounded_decrement* and *atomic_add*, wait-free, long-lived k-renaming can be implemented with time complexity $2\lceil \log_2 k \rceil$ for $k \leq 2(2^b - 1)$.

Proof: By induction on k.

Basis: $k = 2$. 1-renaming can be trivially implemented with no shared accesses. Thus, in this case, the algorithm in Figure 8 implements 2-renaming with two shared accesses.

Induction: $k > 2$. Inductively assume that $\lceil k/2 \rceil$-renaming and $\lfloor k/2 \rfloor$-renaming can be implemented with time complexity at most $2\lceil \log_2 \lceil k/2 \rceil \rceil$ and $2\lceil \log_2 \lfloor k/2 \rfloor \rceil$, respectively. Thus, the algorithm in Figure 8 has time complexity at most $2 + 2\lceil \log_2 \lceil k/2 \rceil \rceil = 2 + 2\lceil \log_2 k - 1 \rceil = 2\lceil \log_2 k \rceil$, so the theorem holds. Note that because the shared counter X must be represented with b bits, this algorithm can only be implemented if $\lceil k/2 \rceil \leq 2^b - 1$. Thus, the proof only holds if $k \leq 2(2^b - 1)$. □

Note that if the *set_first_zero* and *clr_bit* operations are available, then it is unnecessary to completely "unfold" the tree algorithm described above. If the tree is deep enough that at most b processes can reach a leaf, then by Theorem 4, a name can be assigned with one more shared access. This amounts to "chopping off" the bottom $\lfloor \log_2 b \rfloor$ levels of the tree. The time complexity of the resulting algorithm is $\Theta(\log k - \log b) = \Theta(\log(k/b))$. Thus, using all the operations that are employed by the first two algorithms, it is possible to achieve better time complexity than either of them. This approach yields the following result.

Theorem 6: Using b-bit variables and *set_first_zero*, *clear_bit*, *bounded_decrement*, and *atomic_add*, wait-free, long-lived k-renaming can be implemented with time complexity $2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ for $1 \leq k \leq 2(2^b - 1)$.

Proof: By induction on k.

Basis: $k \leq b$. By Theorem 4, wait-free k-renaming can be implemented with time complexity $\lceil k/b \rceil + 1 = 2 = 2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ when $k \leq b$.

Induction: $k > b$. Inductively assume that $\lceil k/2 \rceil$-renaming and $\lfloor k/2 \rfloor$-renaming can each be implemented with time complexity $2(\lceil \log_2 \lceil k/2b \rceil \rceil + 1) = 2\lceil \log_2 \lceil k/b \rceil \rceil$. Then, the algorithm in Figure 8 implements the k-renaming with time complexity at most $2 + 2\lceil \log_2 \lceil k/b \rceil \rceil = 2(\lceil \log_2 \lceil k/b \rceil \rceil + 1)$ shared accesses. As for Theorem 5, this proof only holds if $k \leq 2(2^b - 1)$. □

5.5 Lock-Free, Long-Lived k-Renaming using *fetch_and_add*

The k-renaming algorithm presented in Figure 8 is the basis of our fastest wait-free k-renaming solutions, as shown by Theorems 5 and 6. Unfortunately, the *bounded_decrement* operation employed by that algorithm is not widely available. While the *bounded_decrement* operation is similar to the well-known *fetch_and_add* operation, we have been unable to design an efficient wait-free implementation of the former using the latter. We have, however, designed a lock-free k-renaming algorithm that is based on the idea of *bounded_decrement*. The algorithm is presented in Figure 9. The *fetch_and_add* operation is used to approximate the *bounded_decrement* operation in such a way that it ensures that at most $\lceil k/2 \rceil$ processes access the right instance of $\lceil k/2 \rceil$-renaming, and similarly for the left instance.

Roughly speaking, this split is achieved by having processes that obtain positive values from X go right, and processes that obtain non-positive values go left (see statements 1 and 2 in Figure 9). However, a process, say p, that decrements the counter X below zero "compensates" by incrementing X again before proceeding left. If p detects that X becomes positive again before this compensation is made, then it is possible that some other process has incremented X and joined the left group. In this case, there is a risk that process p should in fact go right, rather than left. In this case, process p restarts the loop.

The algorithm is lock-free because in order for a process to repeat the loop at statements 1 and 2, some other process must modify X between the execution of statements 1 and 2. As the following proof sketch shows, if this happens repeatedly, then eventually some other process makes progress.

The differences between the proofs for the algorithms shown in Figures 8 and 9 are captured by the following three invariants. These invariants are easy to prove, and are therefore stated without proof.

invariant $|\{p :: (p@2 \ \wedge \ p.side = none) \ \vee \ (p@\{3..9\} \ \wedge \ p.side = right)\}| = \lceil k/2 \rceil - X$ (I50)

invariant $|\{p@\{3..9\} \ \wedge \ p.side = right\}| \leq \lceil k/2 \rceil$ (I51)

invariant $|\{p@\{3..9\} \ \wedge \ p.side = left\}| \leq \lfloor k/2 \rfloor$ (I52)

These invariants are analogous to (A4), (I45), and (I46), respectively. As with the proof for the algorithm shown in Figure 8, (I51) and (I52) are used to show that the left and right instances are not accessed by too many processes concurrently. The rest of the proof is similar to the previous one. The lock-freedom property for the algorithm shown in Figure 9 is captured formally by the following property.

Lock-Freedom: If a non-faulty process p attempts to reach its working section, then eventually some process (not necessarily p) reaches its working section.

```
shared variable X : −⌊k/2⌋..⌈k/2⌉                          /* Counter of names available on right */
initially X = ⌈k/2⌉

process p                                                   /* 0 ≤ p < N */
private variable side : {left, right, none}

     while true do
0:      Remainder Section;
        side := none;
        while side = none do
1:         if fetch_and_add(X, −1) > 0 then side := right
2:         else if fetch_and_add(X, 1) < 0 then side := left fi
           fi
        od;
3:      if side = right then
4:            name := Getname_right()                       /* Get name from right instance */
5:      else  name := ⌈k/2⌉ + Getname_left()                /* Get name from left instance */
        fi;
        Working Section;
6:      if side = right then
7:         Putname_right(name);                            /* Return name to right instance */
8:         atomic_add(X, 1)                                /* Increment counter again */
        else
9:         Putname_left(name − ⌈k/2⌉)                      /* Return name to left instance */
        fi
     od
```

Figure 9: Lock-free k-renaming using *fetch_and_add*.

Proof: We inductively assume that the left and right instances are lock-free. Thus, it is easy to see that the only risk to lock-freedom is that some non-faulty process p executes statements $p.1$ and $p.2$ forever, without any other process reaching its working section. Assume, towards a contradiction that process p repeatedly executes statements $p.1$ and $p.2$. Consider consecutive statement executions, of $p.2$ and $p.1$, respectively. By the assumption that the loop executes repeatedly, it follows that $X > 0$ holds immediately after statement $p.2$ is executed, and that $X \leq 0$ holds immediately before statement $p.1$ is executed. Thus, X is decremented at least once between the execution of statements $p.2$ and $p.1$. Consider the first such decrement. The only statement that decrements X is statement $q.1$, for some process q. As $q.1$ is the first decrement of X after the execution of $p.2$, it follows that $X > 0$ holds when $q.1$ is executed. Thus, $q.1$ establishes $q@3 \wedge q.side = right$. Note that process q can only decrement X again after reaching its working section. Thus, because there are a finite number of processes, it follows that p cannot execute statements $p.1$ and $p.2$ forever, without some other process eventually reaching its working section. $\Box$

Given that it is theoretically possible for a single process to repeatedly execute statements $p.1$ and $p.2$ (while other processes are making progress), the worst-case time complexity for the algorithm in Figure 9 is infinite. However, if no other process takes a step between statements $p.1$ and $p.2$ being executed, then the test at statement $p.2$ will succeed. Therefore, if there is no contention, then the number of shared accesses generated by a process acquiring and releasing a name once is at most 2 plus the contention-free time complexity for the inductively-assumed instances. Thus, by an inductive proof similar to the proof of Theorem 5, we have the following result. This result can be extended, as Theorem 5 was in the previous section, to give a result analogous to Theorem 6.

Theorem 7: Using b-bit variables and *fetch_and_add*, lock-free, long-lived k-renaming can be implemented with contention-free time complexity $2\lceil \log_2 k \rceil$ for $k \leq 2^b - 1$. $\Box$

6 Concluding Remarks

In this paper, we have presented two one-time renaming algorithms that employ only atomic read and write operations. One of these algorithms yields an optimal-size name space. These algorithms improve on previous read/write renaming algorithms in that their time complexity is independent of the size of the original name space.

In addition, we have defined a new version of the renaming problem called long-lived renaming, in which processes can release names as well as acquire names. We have provided several solutions to this problem, including one that employs only read and write operations.

Our algorithms exhibit a trade-off between time complexity, name space size, and the availability of primitives used. All of our wait-free algorithms, except the one shown in Figure 8, have the desirable property that time complexity is proportional to contention. This is an important practical advantage because contention should be low in most well-designed applications [8]. The algorithm in Figure 8 has time complexity that is logarithmic in k, regardless of the level of contention.

There are several questions left open by our research. For example, we have shown that one-time $(2k-1)$-renaming can be solved using reads and writes with time complexity $\Theta(k^4)$. We would like to improve on this time complexity while still providing an optimal-size name space. Our fastest read/write algorithm has time complexity $\Theta(k)$ and yields a name space of size $k(k+1)/2$.

The long-lived renaming algorithm presented in Section 4 yields a name space of size $k(k+1)/2$ with time complexity $\Theta(Nk)$. We would like to improve on this result by obtaining an optimal name space of size $2k-1$ using only read and write operations, and by making the time complexity independent of N.

Our most efficient wait-free, long-lived renaming algorithm uses a *bounded_decrement* operation. Although this operation is similar to the standard *fetch_and_add* operation, we have been unable to design an efficient wait-free implementation of the former using the latter. We have, however, designed an efficient lock-free implementation of k-renaming based on this idea. In this implementation, a process can only be delayed by a very unlikely sequence of events. We believe this implementation will perform well in practice. It remains to be seen whether *fetch_and_add* can be used to implement wait-free, long-lived renaming with sub-linear time complexity.

Acknowledgement: We would like to thank Gadi Taubenfeld and Rajeev Alur for helpful discussions.

References

[1] J. Anderson and M. Moir, "Using k-Exclusion to Implement Resilient, Scalable Shared Objects", to appear in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing.*

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Achievable Cases in an Asynchronous Environment", *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, October 1987, pp. 337-346.

[3] A. Bar-Noy and D. Dolev, "Shared Memory versus Message-Passing in an Asynchronous Distributed Environment", *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, August 1989, pp. 307-318.

[4] BBN Advanced Computers, *Inside the TC2000 Computer*, February, 1990.

[5] E. Borowsky and E. Gafni, "Immediate Atomic Snapshots and Fast Renaming", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, August 1993, pp. 41-50.

[6] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", *Communications of the ACM* 12, October 1969, pp. 576-580,583.

[7] M. Herlihy and N. Shavit, "The Asynchronous Computability Theorem for t-Resilient Tasks", *Proceedings of the 25th ACM Symposium on Theory of Computing*, 1993, pp. 111-120.

[8] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February 1987, pp. 1-11.