Using k-Exclusion to Implement Resilient, Scalable Shared Objects*

(Extended Abstract)

James H. Anderson and Mark Moir

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

Abstract

We present a methodology for the implementation of resilient shared objects that allows the desired level of resiliency to be selected based on performance concerns. This methodology is based on the k-exclusion and renaming problems. To make this methodology practical, we present a number of fast k-exclusion algorithms that employ "local spin" techniques to minimize the impact of the processor-to-memory bottleneck. We also present a new "long-lived" renaming algorithm. Our k-exclusion algorithms are based on commonly-available synchronization primitives, are fast in the absence of contention, and have scalable performance when contention exceeds expected thresholds. By contrast, all prior k-exclusion algorithms either require unrealistic atomic operations or perform badly. Our k-exclusion algorithms are also the first algorithms based on local-spin techniques that tolerate process failures.

1 Introduction

In recent years, the distributed algorithms community has spent considerable effort investigating resilient shared object implementations for shared-memory multiprocessing systems. A *shared object* is a data structure, along with associated operations, that is shared by a collection of processes. An implementation of such an object is *k-resilient* iff any process can complete any operation in a finite number of steps, provided at most k other processes fail undetectably. In the literature on resilient shared objects, "wait-free" objects have received the most attention. An N-process object implementation is *wait-free* iff it is $(N-1)$-resilient.

Although wait-free shared objects have many advantages, they have seen only limited application in real systems. One reason is that wait-free algorithms often have time complexity that is at least proportional to N, the number of processes; hence, performance does not scale well as the number of processes increases. In many such algorithms, such poor scalability is largely the result of having to tolerate $N-1$ process failures.[1] However, in a well-designed application, one would seldom expect all processes to compete simultaneously for a single object. In short, wait-freedom links resiliency to worst-case contention, and this can be overkill in practical settings.

From a performance standpoint, linking resiliency to expected levels of contention may be preferable. So doing requires a methodology for implementing objects at intermediate levels of resiliency. Such a methodology can be based on solutions to the *k-exclusion* [9] and *k-assignment* [3] problems. The k-exclusion problem extends the well-known mutual exclusion problem [7] by allowing up to k processes to be in their critical sections simultaneously. k-assignment extends k-exclusion by requiring each process in its critical section to obtain a unique name taken from a fixed set of k names. Solutions to both problems must be able to cope with the failure of up to $k-1$ processes.

A $(k-1)$-resilient shared object can be implemented by encasing a wait-free, k-process implementation of that object within a k-assignment "wrapper". This wrapper permits only k processes to enter the wait-free implementation, and assigns entering processes unique

*Work supported, in part, by NSF Contract CCR 9216421. Email: {anderson,moir}@cs.unc.edu. Phone: (919)962-1757.

[1] Exceptions include objects, such as snapshot objects, in which operations return state information that is at least $O(N)$ in size. By definition, such objects will not scale well, even at lower levels of resiliency.

Ref.	Complexity w/ Contention	Complexity w/o Contention	Instructions Used
[9]	∞	$O(1)$	Large Critical Sections
[10]	∞	$O(1)$	Large Critical Sections
[8]	∞	$O(N^2)$	Safe Bits
[1]	∞	$O(N)$	Atomic Read and Write
Thm. 3	$O(k \log(N/k))$ w/ coherent cache	$O(k)$	Read, Write, Fetch-and-Increment
Thm. 7	$O(k \log(N/k))$	$O(k)$	Above and Compare-and-Swap

Table 1: A comparison of k-exclusion algorithms.

names from a range of size k to use within that implementation. This approach allows $k-1$ process failures to be tolerated. Hence, if contention is at most k, such an implementation is effectively wait-free.

For this methodology to be useful, fast k-assignment algorithms are needed. Unfortunately, prior work on such algorithms has been devoted to message-passing systems. Even for the easier k-exclusion problem, the situation is not encouraging. As shown in Table 1, all prior k-exclusion algorithms for shared memory systems either require unrealistic atomic operations or perform badly. In this table, time complexity is measured as the number of remote accesses of shared memory required per critical section acquisition. An access is *remote* if it requires a traversal of the global interconnect between processors and shared memory, and *local* otherwise. We measure time complexity in terms of remote accesses because performance studies have shown that minimizing such accesses is important for scalable performance [2, 11, 12, 14]. In practice, a shared variable can be made locally-accessible by storing it in a local cache-line or in a local partition of distributed shared memory.

In this paper, we present several fast k-exclusion algorithms, for cache-coherent and distributed shared-memory machines. For both classes of machines, we present algorithms with $O(k)$ complexity if contention is at most k, and $O(k \log(N/k))$ if contention exceeds k. As seen in Table 1, these algorithms are based on commonly-available synchronization primitives. We present several other algorithms in addition to these, including algorithms that exhibit performance that degrades gracefully as contention rises. To achieve good performance in the presence of contention, our algorithms employ "local-spin" techniques to minimize remote accesses of shared memory [2, 11, 12, 14]. We show that our k-exclusion algorithms can be extended to solve k-assignment, specifically by using a new solution to the *renaming* problem [3, 4, 5]. This new solution is the first that is *long-lived*, i.e., that allows each process to repeatedly obtain and release names. It is based on test-and-set, requires a name-space of exactly k, and has time complexity $O(k)$.

The remainder of this paper is organized as follows. In Section 2, we present definitions used in the rest of the paper. In Section 3, we present our k-exclusion algorithms, and in Section 4, we show that these algorithms can be extended to implement k-assignment. Concluding remarks appear in Section 5.

2 Definitions

Our programming notation should be self-explanatory; as an example of this notation, see Figure 1. In this and subsequent figures, each numbered statement is assumed to be atomic. A program's semantics is defined by a set of histories. A *history* of a program is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state and $t_i \xrightarrow{s_i} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of statement s_i.

When reasoning about programs, we define safety properties using invariant and unless assertions and progress properties using leads-to assertions [6]. A state assertion is an *invariant* iff it holds in each state of every history. For state assertions B and C, B *unless* C holds iff for each pair of consecutive states in each history, if $B \wedge \neg C$ holds in the first state, then $B \vee C$ holds in the second. B *leads-to* C in a history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ iff for each state t_i in which B holds, there is a state t_j in which C holds, where $j \geq i$.

In the *k-exclusion* problem, each process cycles through a *noncritical section*, an *entry section*, a *critical section*, and an *exit section*. At most k processes may be in their critical sections at any time. To define the progress property for this problem, it is necessary to distinguish between faulty and nonfaulty processes. For a given history, if a process is not in its noncritical section, and executes no statements after some state, then it is *faulty*; otherwise, it is *nonfaulty*. If at most $k-1$ processes are faulty, then any nonfaulty process in its entry (exit) section must eventually reach its critical (noncritical) section. The *k-assignment* problem extends the k-exclusion problem by requiring each process p to have a local variable $p.name$ ranging over $0..k-1$. If distinct processes p and q are in their critical sections, then it is required that $p.name \neq q.name$.

As mentioned previously, we focus on cache-coherent and distributed shared-memory machines, and measure

```
shared variable X : (k − N)..k ;  Q : queue of 0..N − 1
initially X = k ∧ Q = null

process p                                                         /* 0 ≤ p < N */
  while true do
0:   Noncritical Section ;
1:   ⟨ if fetch_and_increment(X, −1) ≤ 0 then          /* If no critical section slots are available... */
        Enqueue(p, Q) ⟩ ;                                       /* ... then get into queue ... */
2:      while Element(p, Q) do /* null */ od               /* ... and busy-wait until released */
     fi ;
     Critical Section ;
3:   ⟨ Dequeue(Q) ;                                             /* Remove first process from Q */
     fetch_and_increment(X, 1) ⟩                      /* Increase counter of available slots again */
  od
```

Figure 1: (N, k)-exclusion using atomic queue procedures.

complexity by counting "remote" references of shared memory. On distributed shared-memory machines, each shared variable is local to one processor, and remote to all others. Thus, the distinction between local and remote references is straightforward. On cache-coherent machines, making this distinction is more problematic. The main difficulty is determining how many cache misses a busy-wait construct generates. In our cache-coherent algorithms, all busy-waiting is by means of simple loops of the form "**while** $Q = p$ **do od**", where Q is a shared variable and p is the id of the spinning process. We assume that such a loop generates at most two remote references. In particular, we assume that the first read of Q generates a remote reference that causes a copy of Q to migrate to p's local cache. Subsequent reads before Q is written are therefore local. When another process modifies Q, the cache entry is invalidated, so the next read of Q generates a second remote reference, and the loop to terminates. We define *contention* to be the maximum number of processes outside their noncritical sections. Suppose that each matching entry and exit section of an algorithm together generate at most t remote references if executed while contention is at most c. We say that such an algorithm has *time complexity t if contention is at most c.*

Notational Conventions: The predicate $p@i$ holds iff process p's program counter has the value i. We use $p@S$ as shorthand for $(\exists i : i \in S :: p@i)$. We use $p.i$ to denote statement i of process p, and $p.var$ to represent p's local variable var. We refer to k-exclusion for N processes as (N, k)-exclusion; similarly for (N, k)-assignment. It is assumed that $k > 0$, $N > k$, and that p, q, and r range over $0..N-1$. □

3 k-Exclusion

In this section, we present algorithms that efficiently implement k-exclusion using commonly available primitives. On first thought, it may seem that k-exclusion can be easily solved using a queue. To see the difficulties involved with such an approach, consider the simple algorithm in Figure 1. The shared variable X counts the number of processes that may safely enter the critical section and is initially k. When $X \leq 0$, a process trying to enter the critical section waits within the queue Q. $Enqueue(p, Q)$ and $Dequeue(p, Q)$ are the normal queue operations, and $Element(p, Q)$ is a function that returns true iff p is in Q. Multi-line atomic statements are enclosed in angle brackets.

Aside from the multi-line atomic statements, there are two difficulties involved with implementing this algorithm. First, the queue operations typically require several atomic steps if implemented using only simple primitives. Such an implementation is complicated by the possibility that a process may fail after having only partially executed a queue operation. Second, a queue imposes a linear order on the waiting processes. If a process in the queue fails, then other processes in the queue are blocked.

Note that both problems disappear when $N = k + 1$, because at most one process ever waits in the queue. This insight is the basis of the algorithms we present. Specifically, we concentrate on solving $(k+1, k)$-exclusion, and then inductively apply such a solution to solve (N, k)-exclusion.

3.1 Algorithms for Cache-Coherent Machines

In this section, we present a $(k+1, k)$-exclusion algorithm for cache-coherent machines, and then use the inductive approach explained above to solve (N, k)-exclusion. The idea of having one process in the queue is approximated by using a shared variable Q to store the identifier of the process "in the queue".

The algorithm is shown in Figure 2. It uses two procedures, *Acquire* and *Release*, which are assumed to im-

```
shared variable X : −1..k ; Q : 0..N − 1                /* Counter of available slots and spin location */
initially X = k ∧ (∀p : 0 ≤ p ≤ N :: p@0)

process p                                               /* 0 ≤ p < N */
    while true do
0:      Noncritical Section ;
1:      Acquire(N,k + 1) ;                              /* Entry section of (N, k + 1)-exclusion */
2:      if fetch_and_increment(X, −1) = 0 then          /* No slots available */
3:          Q := p ;                                    /* Initialize spin location */
4:          if X < 0 then                               /* Still no slots available - must wait */
5:              while Q = p do /* null */ od            /* Busy-wait until released */
        fi fi ;
        Critical Section ;
6:      fetch_and_increment(X, 1) ;                     /* Release a slot */
7:      Q := p ;                                        /* Release waiting process (if any) */
8:      Release(N,k + 1)                                /* Exit section of (N, k + 1)-exclusion */
    od
```

Figure 2: (N, k)-exclusion on a cache-coherent machine.

plement $(N, k+1)$-exclusion. That is, we have the following properties, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

$$\textbf{invariant } |\{q :: q@\{2..8\}\}| \leq k+1 \qquad \text{(I1)}$$
$$p@1 \;\; \textit{leads-to} \;\; p@2 \qquad \text{(L1)}$$
$$p@8 \;\; \textit{leads-to} \;\; p@0 \qquad \text{(L2)}$$

It is assumed that the variables used by *Acquire* and *Release* are distinct from those in the remainder of the algorithm. Note that if $N = k+1$, then *Acquire* and *Release* are trivially implemented by skip statements. We later use this as the basis of an induction to show that (N, k)-exclusion can be implemented efficiently.

Lemma 1: The algorithm in Figure 2 implements (N, k)-exclusion.

Proof Sketch: This algorithm is proved correct by establishing the following properties.

- *k-Exclusion*: **invariant** $|\{p :: p@6\}| \leq k$.
- *Starvation-Freedom*: If process p is nonfaulty and at most $k-1$ processes are faulty, then $p@1$ *leads-to* $p@6$. Assuming (L2), the progress proof for the exit section is trivial.

For brevity, we state most assertions without proofs, and give only brief descriptions of the major proofs. Complete proofs will be given in the full paper. The following invariants are used to prove k-Exclusion.

$$\textbf{invariant } X = k - |\{p :: p@\{3..6\}\}| \qquad \text{(I2)}$$
$$\textbf{invariant } X < 0 \Rightarrow (\exists p :: p@3 \vee (p@\{4,5\} \wedge Q = p)) \qquad \text{(I3)}$$
$$\textbf{invariant } |\{p :: p@6\}| \leq k \qquad \text{(I4)}$$

(I2) and (I3) are straightforward to prove directly. To show that (I4) holds, we consider two cases. If $X \geq 0$ holds, then by (I2), $|\{p :: p@\{3..6\}\}| \leq k$ holds, so (I4) holds. If $X < 0$ holds, then by (I3), $(\exists p :: p@\{3,4,5\})$ holds, so by (I1), (I4) holds. This proves k-Exclusion. The following unless property is used in the proof of Starvation-Freedom.

$$p@5 \wedge Q \neq p \;\; \textit{unless} \;\; p@6 \qquad \text{(U1)}$$

By (L1) and (L2), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at $p.5$. Process p only reaches $p.5$ by executing $p.4$ when $X < 0$ holds. By (I2), this implies that $|\{p :: p@\{3..6\}\}| > k$ holds when $p.4$ is executed. By the assumption that at most $k-1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..6\}$ holds when $p.4$ is executed.

If $p@5 \wedge Q \neq p$ holds, then by (U1), $p@6$ eventually holds. If $p@5 \wedge Q = p$ holds, then process q is not blocked at $q.5$ because $q \neq p$. Thus, if $p@5 \wedge Q = p$ continues to hold, then q, being nonfaulty, eventually executes $q.7$ and establishes $Q \neq p$. This concludes the proof of Starvation-Freedom. □

Theorem 1: Using fetch-and-increment, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $7(N-k)$.

Proof Sketch: By induction on k.

Basis: $k = N-1$. The *Acquire* and *Release* procedures used by the algorithm in Figure 2 can be implemented by skip statements when $k = N-1$. The correctness of the resulting algorithm follows by Lemma 1. By assumption, the spin-loop at statement 5 generates at most two remote references (see Section 2). Thus, examination of Figure 2 shows that at most five remote

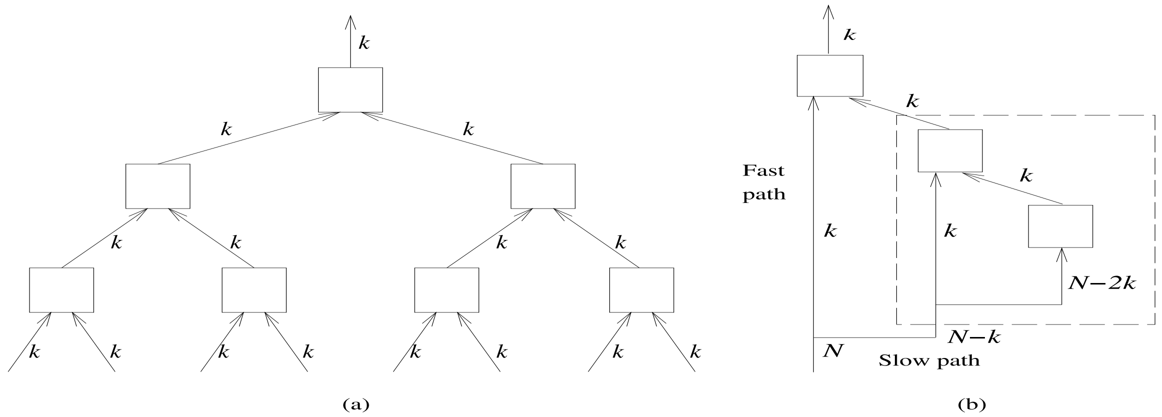

Figure 3: Two approaches for implementing *Acquire*(N,k). *Release*(N,k) is implemented analogously. Each arrow represents a set of processes. Solid boxes represent *Acquire*$(2k,k)$. (a) *Acquire*$(8k,k)$ in a tree. (b) Using a "fast path" to split off k processes. The dotted box represents *Acquire*$(N-k,k)$. One approach for implementing *Acquire*$(N-k,k)$ using nested fast paths is depicted for $N = 4k$ (so $N - 2k = 2k$).

references are generated by the entry section and two remote references by the exit section.

Induction Step: $k < N - 1$. Assume that $(N, k + 1)$-exclusion is implemented with time complexity $7(N - (k+1))$. As shown in the basis, $(k+1, k)$-exclusion can be implemented with time complexity 7. Thus, by Figure 2, (N, k)-exclusion can be implemented with time complexity $7(N - (k+1)) + 7 = 7(N - k)$. □

This inductive algorithm requires $O(N)$ remote references, which is a significant disadvantage. Note, however, that Theorem 1 implies that $(2k, k)$-exclusion can be implemented with time complexity $7k$. We can use such an algorithm as a "building-block" to obtain more efficient implementations of (N, k)-exclusion. (Observe that our $(2k, k)$-exclusion algorithm does not require a process to know the identity of any other process in advance. This is essential for efficiently using such an algorithm as a building-block.) One such approach is to arrange these building-blocks in a tree that halves the number of process at each level, until only k remain. Figure 3(a) depicts this approach for $8k$ processes. This approach yields the following result.

Theorem 2: Using fetch-and-increment, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $7k \log_2 \lceil N/k \rceil$. □

The tree approach offers a significant improvement, but we would like to further reduce the number of remote references performed when contention is low. This can be achieved by adding a "fast path", as shown in Figure 4. A fetch-and-increment[2] instruction is used to select k processes that directly execute *Acquire*$(2k, k)$. The remaining processes must first execute *Acquire*$(N - k, k)$, thereby ensuring that at most $2k$ processes at a time access $(2k, k)$-exclusion. This approach is depicted in Figure 3(b), in which the dotted box represents *Acquire*$(N-k, k)$. Using this algorithm, when at most k processes are participating, the test at statement 2 will always fail, so only *Acquire*$(2k, k)$ and *Release*$(2k, k)$ are executed. Thus, if contention is at most k, the number of remote references is at most $7k + 2$.

The performance degradation with increasing contention for this algorithm is determined by the implementation of $(N - k, k)$-exclusion — the "slow path". One alternative is to use an (N, k)-exclusion tree, as illustrated in Figure 3(a), yielding the following result.

Theorem 3: Using fetch-and-increment, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $7k + 2$ if contention is at most k, and $7k(\log_2 \lceil N/k \rceil + 1) + 2$ otherwise. □

A second alternative is to implement $(N - k, k)$-exclusion inductively using the algorithm given in Figure 4, as depicted inside the dotted box in Figure 3(b). This results in performance that degrades gracefully with increasing contention, rather than performance that drops suddenly when contention exceeds k. In particular, the number of remote references is proportional to contention.

Theorem 4: Using fetch-and-increment, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $\lceil c/k \rceil (7k + 2)$ if contention is at most c. □

[2] For simplicity, we assume here that *fetch-and-increment* does not cause a range error, e.g., $fetch - and - increment(X, -1)$ does not change X if executed when X is 0. Removing this assumption results in a slightly more complicated algorithm for Theorem 4 and a small constant factor increase in time complexity; for details, see the full paper.

```
shared variable X : 0..k ;                                    /* Counter of available slots */
initially (∀p : 0 ≤ p ≤ N :: p@0) ∧ X = k
process p                                                     /* 0 ≤ p < N */
private variable slow : boolean                               /* Records path taken */
    while true do
0:      Noncritical Section ;
1:      slow := false ;                                       /* Haven't gone through the slow path yet */
2:      if fetch_and_increment(X, −1) = 0 then                /* No slots available - go through slow path */
3:          slow := true ;                                    /* Record that slow path was taken */
4:          Acquire(N − k, k)                                 /* Slow path */
        fi ;
5:      Acquire(2k, k) ;                                      /* Fast path */
        Critical Section ;
6:      Release(2k, k) ;
7:      if slow then                                          /* Check if slow path was taken */
8:          Release(N − k, k)
9:      else fetch_and_increment(X, 1)
        fi
    od
```

Figure 4: (N, k)-exclusion with a "fast path".

3.2 Algorithms for Distributed Shared-Memory Machines

In the previous section, we showed that k-exclusion can be efficiently implemented on cache-coherent machines. Such implementations are efficient because when a process waits on a variable, that variable migrates to a local cache-line. A distributed shared-memory machine without cache-coherence does not provide this luxury. On such a machine, each variable is local to only one process, so for good scalability, different processes must wait on different variables. This makes k-exclusion significantly more difficult to implement efficiently. In this section, we show that (N, k)-exclusion can be implemented using algorithms in which all busy-waiting is on local shared variables. We use an inductive approach, similar to that in the previous section, to reduce the problem to that of implementing $(k + 1, k)$-exclusion.

The algorithm in Figure 5 implements (N, k)-exclusion. As in the previous section, the *Acquire* and *Release* procedures used in Figure 5 are assumed to implement $(N, k + 1)$-exclusion. Instead of all processes waiting on one spin location Q, each process p now has an unbounded set of local spin locations, $P[p, v]$, $v \geq 0$. In the next algorithm, we bound this set of spin locations. Variable Q is now used to indicate the spin location on which the currently-blocked process is waiting.

Because different processes wait on different variables, for a process q to release a blocked process p, process q must first identify the spin location on which p is waiting. Statements $q.5$ and $q.6$ achieve this by reading the identifier of the spin location from Q, and then updating that spin location. When $k + 1$ processes have successfully executed the *Acquire* procedure, it is required that at least one of these processes wait, so that k-Exclusion is not violated. Thus, when process q releases process p from its spin-loop, process q should itself start waiting. This gives rise to the possibility that before q releases p, another process r releases p and starts waiting. If q does not detect this, then q might start waiting too. If the $k - 1$ remaining processes are faulty, then q and r might wait forever, violating Starvation-Freedom. Thus, we need a mechanism to allow process q to detect that process p has already been released. The compare-and-swap[3] instruction in statement 7 serves this purpose, specifically by allowing a process to detect that Q was modified between its executions of statements 5 and 7. Observe that if $q.5$ and $r.5$ read the same spin location identifier from Q, and if $q.7$ modifies Q, then $r.7$'s compare-and-swap will fail and r will not wait.

Lemma 2: The algorithm in Figure 5 implements (N, k)-exclusion.

Proof Sketch: The proof is similar to the proof of Lemma 1. For convenience, we define the following sets. $A \equiv \{p :: p@\{3, 4\}\}$, $B \equiv \{p :: p@5 \wedge \neg P[p, p.next.loc]\}$, $C \equiv \{p :: p@\{6, 7\} \wedge Q = p.v \wedge \neg P[p, p.next.loc]\}$, and $D \equiv \{p :: p@\{8, 9\} \wedge \neg P[p, p.next.loc]\}$. The following are the main safety properties are used in the proof.

invariant $X = k - |\{p :: p@\{3..10\}\}|$ (I5)

invariant $p@\{4..7\} \Rightarrow q.v \neq p.next \wedge Q \neq p.next$ (I6)

invariant $p@\{5..7\} \Rightarrow \neg P[p, p.next.loc]$ (I7)

invariant $q@\{6, 7\} \wedge X < 0 \Rightarrow$
$(\exists r :: (r@\{3..7\} \vee (\forall q :: q.v \neq r.next)) \wedge$
$r \in A \cup B \cup C \cup D)$ (I8)

invariant $X < 0 \Rightarrow (\exists r :: r \in A \cup B \cup C \cup D)$ (I9)

[3] *compare-and-swap*(Q, v, x) "fails" if $Q \neq v$ holds, and "succeeds" if $Q = v$ holds. In the former case, it simply returns *false*, and in the latter case, it assigns $Q := x$ and returns *true*.

```
type loc_type = record pid: 0..N−1 ; loc: 0..∞ end
shared variable X : −1..k ; Q : loc_type ; P : array[0..N−1, 0..∞] of boolean
initially X = k ∧ Q = (0,0)                          /* P[p,i] for i ≥ 0 is local to process p */

process p                                                                   /* 0 ≤ p < N */
private variable next, v : loc_type
initially p@0 ∧ next.pid = p ∧ next.loc = 0 ∧ v.loc = 0

    while true do
0:      Noncritical Section ;
1:      Acquire(N,k+1) ;                              /* Entry section of (N,k+1)-exclusion */
2:      if fetch_and_increment(X,−1) = 0 then                     /* No slots available */
3:          next.loc := next.loc + 1 ;               /* Use spin location never used before */
4:          P[p, next.loc] := false ;                           /* Initialize spin location */
5:          v := Q ;                                          /* Get current spin location */
6:          P[v.pid, v.loc] := true ;                /* Release currently spinning process */
7:          if compare_and_swap(Q, v, next) then          /* Spinning process still the same */
8:              if X < 0 then                       /* Still no slots available - must wait */
9:                  while ¬P[p, next.loc] do /* null */ od             /* Wait until released */
        fi fi fi ;
        Critical Section ;
10:     fetch_and_increment(X,1) ;                                        /* Release a slot */
11:     v := Q ;                                              /* Get current spin location */
12:     P[v.pid, v.loc] := true ;                               /* Release spinning process */
13:     Release(N,k+1)                                 /* Exit section of (N,k+1)-exclusion */
    od
```

Figure 5: (N,k)-exclusion using an unbounded number of local spin locations on a distributed shared-memory machine.

$$\textbf{invariant } |\{p :: p@10\}| \leq k \qquad \text{(I10)}$$

$$p@\{5..9\} \wedge P[p, p.next.loc] \;\; \textit{unless} \;\; p@10 \qquad \text{(U2)}$$

The proofs of (I5) – (I10) and (U2) are similar to the proofs for (I2) – (I4) and (U1). (I10), which implies k-Exclusion, follows from (I5), (I9), and the assumption that *Acquire* and *Release* implement $(k+1)$-exclusion.

The only risk to Starvation-Freedom is that a nonfaulty process p waits forever at statement $p.9$. For each of the following cases, we show that if $p@9$ holds, then $p@10$ eventually holds.

Case 1: *Q is modified after the preceding $p.7$ is executed.* Because $p@9$ holds, the compare-and-swap instruction of the preceding $p.7$ succeeded, so $Q = p.next$ holds after $p.7$ is executed. By the definition of compare-and-swap, the first modification of Q after the execution of $p.7$ is the result of the execution of statement $q.7$ for some process q such that $q@7 \wedge q.v = Q$ holds. Because $p.7$ establishes $Q = p.next$ prior to $q.7$, $q.v = p.next$ holds immediately before $q.7$ is executed. By (I6) and (I7), $q.v \neq p.next \wedge \neg P[p, p.next.loc]$ holds before $p.7$ is executed. Thus, $q.v = p.next$ is established between the executions of $p.7$ and $q.7$. Process p does not modify $p.next$ between $p.7$ and $p.9$, and statement $q.5$ is the last statement to modify $q.v$ before $q.7$. Thus, either $p.9$ is eventually executed, or $q.5$ and therefore $q.6$ are executed after $p.7$. In the former case, $p@10$ is established. In the latter case, $q.6$ establishes $P[p, p.next.loc]$ after $p.7$ is executed. Thus, by (U2), statement $p.9$ is eventually executed, establishing $p@10$.

Case 2: *Q is not modified after the preceding $p.7$ is executed.* Because $p@9$ holds, $X < 0$ holds before the preceding $p.8$ is executed. Thus, by (I5), $|\{q :: q@\{3..10\}\}| > k$ holds after $p.8$ is executed. Thus, by the assumption that there are at most $k-1$ faulty processes, there is a nonfaulty process $t \neq p$ such that $t@\{3..10\}$ holds. If t does not become permanently blocked at $t.9$, then because t is nonfaulty, and because Q is not modified after $p.7$, statements $t.11$ and $t.12$ eventually establish $P[p, p.next.loc]$. In this case, by (U2), p eventually establishes $p@10$, as p is nonfaulty. If t does become permanently blocked at $t.9$, then $t.7$ must have modified Q. As Q is not modified after $p.7$ is executed, $t.7$ is executed before $p.7$. Then, by Case 1, $t@10$ eventually holds — a contradiction. This completes the proof of Starvation-Freedom. □

The obvious drawback of the algorithm in Figure 5 is that each process uses a new spin location for every execution of the entry section, so the space complexity of the algorithm is unbounded. The algorithm given in Figure 6 uses only a bounded number of spin locations per process. The spin locations for process p are $P[p, v]$, where $0 \leq v < k+2$. Associated with each spin location $P[p, v]$ is a counter $R[p, v]$. Roughly speaking, $R[p, v]$ counts the number of processes that have read (p, v)

from Q and might set $P[p, v]$. When selecting a new spin location, p chooses w such that $R[p, w] = 0$.

In order for the algorithm to use only bounded space, a process p must eventually choose a spin location that it has previously used. If p could choose a spin location not currently stored in any variable, then this would have the same effect as choosing a new location. It is easy for p to ensure that it chooses a spin location different from the one stored in Q. However, some process $q \neq p$ may have previously read Q, and then experienced a delay. Suppose q's read of Q obtained one of p's spin locations — say (p, v). Process p should not use (p, v) again while (p, v) remains in q's local state, but this presents a difficulty. Process p cannot read q's local state, so it is impossible to choose a new spin location that is not stored in any variable.

All hope is not lost, however. The requirement that p must choose a location that is not stored in any variable can be relaxed. Instead, we require that p chooses a location that will not be prematurely set by another process. That is, we require that p chooses a spin location (p, v) that will not be set before p executes the *compare_and_swap* in statement $p.11$. Thus, statement $p.11$ announces that p will spin on local spin location $P[p, v]$, and we require that $P[p, v]$ is not set before this announcement is made. To achieve this, we introduce some feedback between processes. As seen in Figure 6, after $q.7$ reads (p, v) from Q, $q.8$ "informs" p by incrementing $R[p, v]$, and then $q.9$ reads Q again. If Q changes between $q.7$ and $q.9$, then process q will not set $P[p, v]$. On the other hand, if Q does not change between $q.7$ and $q.9$, then it can be shown that if p is using the spin location (p, v) when $q.9$ is executed, then p has already executed statement $p.11$. Also, $q.8$ informs p not to choose (p, v) for its spin location before q executes $q.15$.

It may seem possible that each time statement $p.4$ reads a counter $R[p, v]$, that counter is positive. In fact, using the assumption that *Acquire* and *Release* are correct, it can be shown that this is not possible. In particular, it can be shown that when process p starts reading the counters in R, there is some $v \neq p.last$ such that $R[p, v] = 0$ holds, and that this continues to hold until p reads $R[p, v]$. Thus, the loop at statements $p.4$ and $p.5$ terminates after at most $k + 1$ iterations. Each process uses $k+2$ spin locations to ensure that the most-recently-used spin location is not chosen again. Proofs of the results below are similar to previous ones, and will appear in the full version of the paper.

Theorem 5: Using fetch-and-increment and compare-and-swap, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $14(N - k)$.

Proof Sketch: By induction, as in Theorem 1, using the algorithm of Figure 6. Note that all accesses in statements 4 and 5 are local to process p. □

Theorem 6: Using fetch-and-increment and compare-and-swap, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $14k \log_2\lceil N/k\rceil$.

Proof Sketch: By Theorem 5, $(2k, k)$-exclusion can be implemented on a distributed shared-memory machine with time complexity $14k$. The theorem follows by arranging such implementations in a tree as illustrated in Figure 3(a). □

Theorem 7: Using fetch-and-increment and compare-and-swap, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $14k + 2$ if contention is at most k, and $14k(\log_2\lceil N/k\rceil + 1) + 2$ if contention exceeds k.

Proof Sketch: This result is proved using the algorithm in Figure 4, with $(2k, k)$-exclusion implemented as in Theorem 5, and using a tree to implement $(N-k, k)$-exclusion as depicted in Figure 3(a). □

Theorem 8: Using fetch-and-increment and compare-and-swap, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $\lceil c/k\rceil(14k + 2)$ if contention is at most c.

Proof Sketch: This result is proved by using the algorithm shown in Figure 4, with $(2k, k)$-exclusion implemented as in Theorem 5 and $(N-k, k)$-exclusion implemented using the algorithm in Figure 4. This approach is depicted in Figure 3(b). □

4 k-Assignment

In the renaming problem [3], each of k processes must be assigned a unique name from a fixed name space. A solution to the renaming problem can be used to extend a k-exclusion algorithm to a k-assignment algorithm. To be useful in this setting, a solution to the renaming problem must be *long-lived*: each process must be able to repeatedly obtain and release names. In this section, we present a long-lived renaming algorithm that uses test-and-set.

In order to obtain a name using our algorithm, a process test-and-sets each of a sequence of bits in order, until a test-and-set succeeds. The bit $X[j]$ is associated with name j, where $0 \leq j < k$. To release the name, the successfully-set bit is cleared. It is easy to see that distinct processes in their critical sections have distinct names. It can be shown that if a process is about to test-and-set $X[i]$, then there is a j where $i \leq j < k$, such that $\neg X[j]$ holds. Thus, if a process has unsuccessfully tested bits $X[0]..X[k-2]$, then $\neg X[k-1]$ holds, so the

```
type loctype = record pid: 0..N − 1 ; loc: 0..k + 1 end
shared variable   X : −1..k ;   Q : loctype ;
                  P : array[0..N − 1, 0..k + 1] of boolean ;  R : array[0..N − 1, 0..k + 1] of 0..k + 1
initially (∀p, j : 0 ≤ p < N ∧ 0 ≤ j < k + 2 :: R[p, j] = 0) ∧ X = k ∧ Q = (0, 0)

process p                          /* 0 ≤ p < N, P[p,i] and R[p,i] for 0 ≤ i ≤ k + 1 are local to process p */
private variable u, next : loctype ; last : 0..k + 1
initially p@0 ∧ last = 0

    while true do
0:     Noncritical Section ;
1:     Acquire(N,k + 1) ;                                  /* Entry section of (N,k + 1)-exclusion */
2:     if fetch_and_increment(X, −1) = 0 then                                /* No slots available */
3:        next.loc := (last + 1) mod (k + 2) ;                     /* Start at location after last one */
4:        while R[p, next.loc] ≠ 0 do                            /* Search for a spin location not in use */
5:           next.loc := (next.loc + 1) mod (k + 2)
          od ;
6:        P[p, next.loc] := false ;                                         /* Initialize spin location */
7:        u := Q ;                                                         /* Get current spin location */
8:        fetch_and_increment(R[u.pid, u.loc], 1) ;    /* Record that this spin location is about to be written */
9:        if Q = u then                                            /* Spin location has not changed */
10:          P[u.pid, u.loc] := true ;                            /* Release currently spinning process */
11:          if compare_and_swap(Q, u, next) then                   /* Spinning process still the same */
12:             last := next.loc ;                                    /* Record last spin location used */
13:             if X < 0 then                                   /* Still no slots available - must wait */
14:                while ¬P[p, next.loc] do /* null */ od                           /* Wait until released */
          fi fi fi ;
15:       fetch_and_increment(R[u.pid, u.loc], −1)                     /* Finished with this spin location */
       fi ;
       Critical Section ;
16:    fetch_and_increment(X, 1) ;                                                    /* Release a slot */
17:    u := Q ;                                                            /* Get current spin location */
18:    fetch_and_increment(R[u.pid, u.loc], 1) ;       /* Record that this spin location is about to be written */
19:    if Q = u then                                                  /* Spin location has not changed */
20:       P[u.pid, u.loc] := true                                            /* Release spinning process */
       fi ;
21:    fetch_and_increment(R[u.pid, u.loc], −1) ;                      /* Finished with this spin location */
22:    Release(N,k + 1)                                          /* Exit section of (N, k + 1)-exclusion */
    od
```

Figure 6: (N, k)-exclusion using a bounded number of local spin locations on a distributed shared-memory machine.

```
shared variable X : array[0..k − 2] of boolean
initially (∀i : 0 ≤ i ≤ k − 2 :: ¬X[i])

process p                                                                           /* 0 ≤ p < N */
local variable name : 0..k − 1
initially p@0 ∧ name = 0

  while true do
0:  Noncritical Section ;
1:  Acquire(N,k) ;                                                  /* Entry section for (N, k)-exclusion */
2:  while name < k − 1 ∧ test_and_set(X[name]) = true do name := name + 1 od ;  /* Set first clear bit to get a name */
    Critical Section using name name ;
3:  X[name], name := false, 0 ;                                    /* Release name by resetting bit found */
4:  Release(N,k)                                                     /* Exit section for (N, k)-exclusion */
  od
```

Figure 7: Algorithm for k-assignment using test-and-set for renaming.

kth test-and-set will succeed. It can also be shown that at most one process accesses $X[k-1]$ concurrently, so in fact this bit is unnecessary. The algorithm shown in Figure 7 combines this approach with $Acquire(N, k)$ and $Release(N, k)$ to implement k-assignment. This algorithm is presented in more detail in [13].

Our renaming algorithm generates at most k additional remote references. Thus, Theorems 3 and 7 can be extended to give the following results. All the other theorems in Section 3 can be extended similarly.

Theorem 9: Using fetch-and-increment and test-and-set, (N, k)-assignment can be implemented on a cache coherent machine with time complexity $8k + 2$ if contention is at most k, and $7k(\log_2\lceil N/k\rceil + 1) + k + 2$ if contention exceeds k. □

Theorem 10: Using fetch-and-increment, compare-and-swap, and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory machine with time complexity $15k + 2$ if contention is at most k, and $14k(\log_2\lceil N/k\rceil + 1) + k + 2$ if contention exceeds k. □

5 Concluding Remarks

The algorithms presented in this paper provide a good starting point towards a practical methodology for constructing resilient shared objects. Ultimately, we would like to develop k-exclusion algorithms for which performance under contention is completely independent of N. We would also like for such algorithms to have performance that approaches that of the fastest spin-lock algorithms [2, 11, 12, 14] when k approaches 1. If based on universal wait-free constructions, the methodology we suggest could yield a generic approach to shared object design in which resiliency can be "tuned" according to performance demands.

References

[1] A. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "First-in-First-Enabled l-Exclusion", *Proceedings of the 4th International Workshop on Distributed Algorithms*, 1990.

[2] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January, 1990, pp. 6-16.

[3] 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.

[4] 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*, ACM, New York, August 1989, pp. 307-318.

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

[6] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[7] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 9, 1965, p. 569.

[8] D. Dolev, E. Gafni, and N. Shavit, "Towards a Non-atomic Era: l-Exclusion as a Test Case", *Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988, pp. 78-92.

[9] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Resource Allocation with Immunity to Process Failure", *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, October, 1979, pp. 234-254.

[10] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Space", *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 1, January, 1989, pp. 90-114.

[11] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors", *IEEE Computer*, Vol. 23, June, 1990, pp. 60-69.

[12] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991, pp. 21-65.

[13] M. Moir and J. Anderson, "Fast, Long-Lived Renaming", submitted for publication in *Proceedings of the 8th International Workshop on Distributed Algorithms*, 1994.

[14] J.-H. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, August 1993, pp. 171-182.