Adaptive Mutual Exclusion with Local Spinning*

James H. Anderson and Yong-Jik Kim

Department of Computer Science, University of North Carolina at Chapel Hill

Abstract. We present the first adaptive algorithm for N-process mutual exclusion under read/write atomicity in which all busy waiting is by local spinning. In our algorithm, each process p performs $O(min(k, \log N))$ remote memory references to enter and exit its critical section, where k is the maximum "point contention" experienced by p. The space complexity of our algorithm is $\Theta(N)$, which is clearly optimal.

1 Introduction

In this paper, we consider adaptive solutions to the mutual exclusion problem [7] under read/write atomicity. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [6, 11, 13]. Two notions of contention have been considered in the literature: "interval contention" and "point contention" [1]. These two notions are defined with respect to a history H. The *interval contention* over H is the number of processes that are active in H, i.e., that execute outside of their noncritical sections in H. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. In this paper, we consider only point contention. Throughout the paper, we let N denote the number of processes in the system, and we let k denote the point contention experienced by an arbitrary process over a history that starts when it becomes active and ends when it once again becomes inactive.

In previous work on adaptive mutual exclusion algorithms, two time complexity measures have been considered: "remote step complexity" and "system response time." The *remote step complexity* of an algorithm is the maximum number of shared-memory operations required by a process to enter and then exit its critical section, assuming that each "**await**" statement is counted as one operation [13]. The *system response time* is the length of time between critical section entries, assuming each enabled read or write operation is executed within some constant time bound [6]. Several read/write mutual exclusion algorithms have been presented that are adaptive to some degree under these time complexity measures. One of the first such algorithms was an algorithm of Styer that has $O(min(N, k \log N))$ remote step complexity and $O(min(N, k \log N))$ response time [13]. Choy and Singh later improved upon Styer's result by presenting an algorithm with $O(N)$ remote step complexity and $O(k)$ response time

* Work supported by NSF grants CCR 9732916 and CCR 9972211.

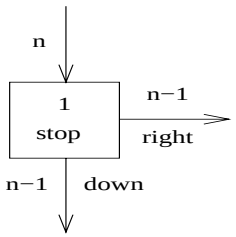
[6]. More recently, Attiya and Bortnikov presented an algorithm with $O(k)$ remote step complexity and $O(\log k)$ response time [5].

Recent work on scalable local-spin mutual exclusion algorithms has shown that *the* most crucial factor in determining an algorithm's performance is the amount of interconnect traffic it generates [4, 8, 10, 14]. In light of this, we define the *time complexity* of a mutual exclusion algorithm to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section. A *remote memory reference* is a shared variable access that requires an interconnect traversal. In local-spin algorithms, all busy-waiting loops are required to be read-only loops in which only locally-accessible shared variables are accessed that do not require an interconnect traversal. On a distributed shared-memory multiprocessor, a shared variable is locally accessible if it is stored in a local memory module. On a cache-coherent multiprocessor, a shared variable is locally accessible if it is stored in a local cache line.

The first local-spin algorithms were algorithms in which read-modify-write instructions are used to enqueue blocked processes onto the end of a "spin queue" [4, 8, 10]. Each of these algorithms has $O(1)$ time complexity; thus, adaptivity is clearly a non-issue if appropriate read-modify-write instructions are available. Yang and Anderson were the first to consider local-spin algorithms under read/write atomicity [14]. They presented a read/write mutual exclusion algorithm with $\Theta(\log N)$ time complexity in which instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree. They also presented a "fast-path" variant of this algorithm that allows the tree to be bypassed in the absence of contention. Although the contention-free time complexity of this algorithm is $O(1)$, its time complexity under contention is $\Theta(N)$ in the worst case, rather than $\Theta(\log N)$. In recent work, Anderson and Kim presented a new fast-path mechanism that results in with $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ time complexity under contention, when used with Yang and Anderson's algorithm [3].

All of the previously-cited adaptive algorithms are not local-spin algorithms, and thus they have unbounded time complexity under the remote-memory-references time measure. One could argue that for an algorithm to be considered truly adaptive, it must be adaptive under this measure. After all, the underlying hardware does not distinguish between remote memory references generated by **await** statements and remote memory references generated by other statements. Surprisingly, while adaptivity and local spinning have been the predominate themes in recent work on mutual exclusion, the problem of designing an adaptive, local-spin algorithm under read/write atomicity has remained open. In this paper, we close this problem by presenting an algorithm that has $O(min(k, \log N))$ time complexity under the remote-memory-references measure.

Our algorithm can be seen as an extension of the fast-path algorithm of Anderson and Kim [3]. This algorithm was devised by thinking about connections between fast-path mechanisms and long-lived renaming [12]. Long-lived renaming algorithms are used to "shrink" the size of the name space from which process identifiers are taken. The problem is to design operations that processes may in-




```
shared variables X: {⊥} ∪ {0..N − 1} init ⊥;
                 Y: boolean init false

private variable dir: {D, R, S} /* down, right, stop */

1: X := p;
2: if Y then dir := R
   else
3:      Y := true;
4:      if X = p  then dir := S
                  else dir := D
        fi
   fi
```

Fig. 1. The splitter element and the code fragment that implements it.

voke in order to acquire new names from the reduced name space when they are needed, and to release any previously-acquired name when it is no longer needed. In Anderson and Kim's algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. Our adaptive algorithm can be seen as a generalization of Anderson and Kim's fast-path mechanism in which *every* name is associated with some "path" to the critical section. The length of the path taken by a process is determined by the point contention that it experiences.

2 Adaptive Algorithm

In our adaptive algorithm, code sequences from several other algorithms are used. In Sec. 2.1, we present a review of these other algorithms and discuss some of the basic ideas underlying our algorithm. Then, in Sec. 2.2, we present a detailed description of our algorithm.

2.1 Related Algorithms and Key Ideas

At the heart of our algorithm is the splitter element from the grid-based long-lived renaming algorithm of Moir and Anderson [12]. This splitter element was actually first used in Lamport's fast mutual exclusion algorithm [9]. The splitter element is defined by the code fragment shown in Fig. 1. (In this and subsequent figures, we assume that each labeled sequence of statements is atomic; in each figure, each labeled sequence reads or writes at most one shared variable.) Each process that invokes this code fragment either stops, moves down, or moves right (the move is defined by the value assigned to the variable *dir*). One of the key properties of the splitter that makes it so useful is the following: if n processes invoke a splitter, then at most one of them can stop at that splitter, at most $n-1$ can move right, and at most $n-1$ can move down.

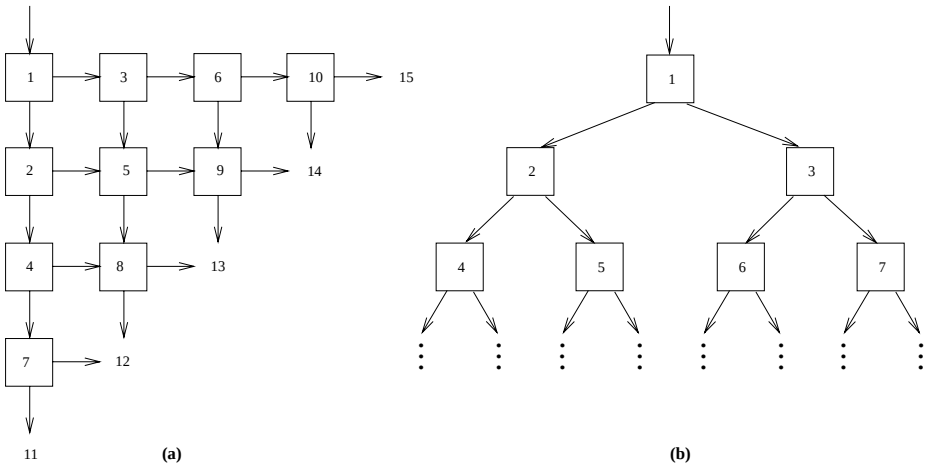


Fig. 2. **(a)** Renaming grid (depicted for $N = 5$). **(b)** Renaming tree.

Because of these properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid as shown in Fig. 2(a). A name is associated with each splitter. If the grid has N rows and N columns, then by induction, every process eventually stops at some splitter. When a process stops at a splitter, it acquires the name associated with that splitter. In the *long-lived* renaming problem [12], processes must have the ability to release the names they acquire. In the grid algorithm, a process can release its name by resetting each splitter on the path traversed by it in acquiring its name. A splitter can be reset by resetting its Y variable to *true*. For the renaming mechanism to work correctly, it is important that a splitter be reset *only* if there are no processes "downstream" from it (i.e., in the sub-grid "rooted" at that splitter). In Moir and Anderson's algorithm, it takes $O(N)$ time to determine whether there are "downstream" processes. This is because each process checks every other process individually to determine if it is downstream from a splitter. As we shall see, a more efficient reset mechanism is needed for our adaptive algorithm.

The main idea behind our algorithm is to let an arbitration tree form dynamically within a structure similar to the renaming grid. This tree may not remain balanced, but its height is proportional to contention. The job of integrating the renaming aspects of the algorithm with the arbitration tree is greatly simplified if we replace the grid by a binary tree of splitters as shown in Fig. 2(b). (Since we are now working with a tree, we will henceforth refer to the directions associated with a splitter as stop, left, and right.) Note that this results in many more names than before. However, this is not a major concern, because we are really not interested in minimizing the name space. The arbitration tree is defined by associating a three-process mutual exclusion algorithm with each node in the renaming tree. This three-process algorithm can be implemented in constant time using the local-spin mutual exclusion algorithm of Yang and Anderson [14]. We

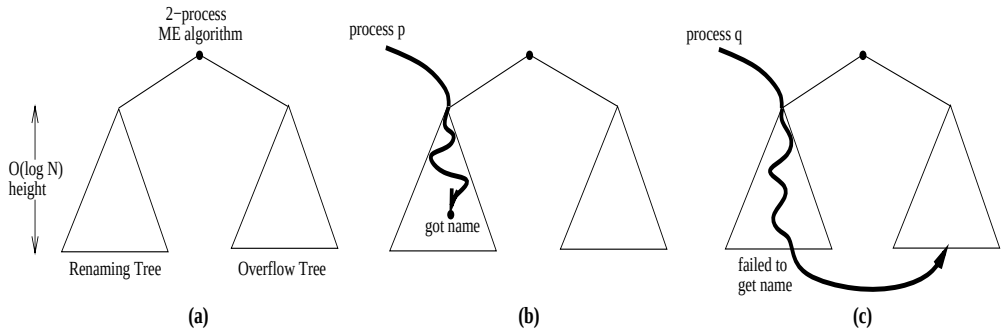


Fig. 3. **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

explain below why a three-process algorithm is needed instead of a two-process algorithm (as one would expect to have in an arbitration tree).

In our algorithm, a process p performs the following basic steps. (For the moment, we are ignoring certain complexities that must be dealt with.)

Step 1 p first acquires a new name by moving down from the root of the renaming tree, until it stops at some node. In the steps that follow, we refer to this node as p's *acquired node*. p's acquired node determines its starting point in the arbitration tree.

Step 2 p then competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root. Note that a node's entry section may be invoked by the process that stopped at that node, and one process from each of the left and right subtrees beneath that node. This is why a three-process algorithm is needed.

Step 3 After competing within the arbitration tree, p executes its critical section.

Step 4 Upon completing its critical section, p releases its acquired name by reopening all of the splitters on the path from its acquired node to the root.

Step 5 After releasing its name, p executes each of the three-process exit sections on the path from the root to its acquired node.

If we were to use a binary tree of height N, just as we previously had a grid with N row and N columns, then the total number of nodes in the tree would be $\Theta(2^N)$. We circumvent this problem by defining the tree's height to be $\lfloor \log N \rfloor$, which results in a tree with $\Theta(N)$ nodes. With this change, a process could "fall off" the end of the tree without acquiring a name. However, this can happen only if contention is $\Omega(\log N)$. To handle processes that "fall off the end," we introduce a second arbitration tree, which is implemented using Yang and Anderson's local-spin arbitration-tree algorithm [14]. We refer to the two trees used in our algorithm as the *renaming tree* and *overflow tree*, respectively. These two trees are connected by placing a two-process version of Yang and Anderson's algorithm on top of each tree, as illustrated in Fig. 3(a). Fig. 3(b)

illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Fig. 3(c) illustrates the steps that might be taken by a process q if contention is $\Omega(\log N)$.

A major difficulty that we have ignored until this point is that of efficiently reopening a splitter, as described in Step 4 above. In Moir and Anderson's renaming algorithm, it takes $O(N)$ time to reopen a splitter. To see why reopening a splitter is difficult, consider again Fig. 1. If a process does succeed stopping at a splitter, then that process can reopen the splitter itself by simply assigning $Y := \mathit{true}$. On the other hand, if no process succeeds in stopping at a splitter, then some process that moved left or right from that splitter must reset it. Unfortunately, because processes are asynchronous and communicate only by means of atomic read and write operations, it can be difficult for a left- or right-moving process to know whether some process has stopped at a splitter.

Anderson and Kim solved this problem in their fast-path mutual exclusion algorithm by exploiting the fact that much of the reset code can be executed within a process's critical section [3]. Thus, the job of designing efficient reset code is much easier here than when designing a *wait-free* long-lived renaming algorithm. As mentioned earlier, in Anderson and Kim's fast-path algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. In our adaptive algorithm, we must efficiently manage acquisitions and releases for a set of names.

2.2 Detailed Description

Having introduced the major ideas that underlie our algorithm, we now present a detailed description of the algorithm and its properties. We do this in three steps. First, we consider a version of the algorithm in which unbounded memory is used to reset splitters in constant time. Second, we consider a variant of the algorithm with $\Theta(N^2)$ space complexity in which all variables are bounded. Third, we present another variant that has $\Theta(N)$ space complexity. In explaining these algorithms, we actually present proof sketches for some of the key properties of each algorithm. Our intent is to use these proof sketches as a means for intuitively explaining the basic mechanisms of each algorithm. A formal correctness proof for the final algorithm is presented in the full version of this paper [2].

Algorithm U. The first algorithm, which we call Algorithm U (for unbounded), is shown in Fig. 4. Before describing how this algorithm works, we first examine its basic structure. At the top of Fig. 4, definitions of two constants are given: D, which is the maximum level in the renaming tree (the root is at level 0), and T, which gives the total number of nodes in the renaming tree. As mentioned earlier, the renaming tree is comprised of a collection of splitters. These splitters are indexed from 1 to T. If splitter i is not a leaf, then its left and right children are splitters $2i$ and $2i+1$, respectively.

Each splitter i is defined by four shared variables and an infinite shared array: $X[i]$, $Y[i]$, $\mathit{Reset}[i]$, $\mathit{Rnd}[i]$ (the array), and $\mathit{Acquired}[i]$. Variables $X[i]$ and $Y[i]$ are as in Fig. 1, with the exception that $Y[i]$ now has an additional

```
const
    D = ⌊log N⌋;                          /* depth of renaming tree = O(log N) */
    T = 2^{D+1} − 1                       /* size of renaming tree = O(N) */
type
    Ytype = record free: boolean; rnd: 0..∞ end;      /* stored in one word */
    Dtype = {L, R, S};                    /* splitter moves (left, right, stop) */
    Ptype = record nd: 1..2T + 1; dir: Dtype end      /* path information */

shared variables
    X: array[1..T] of 0..∞;
    Y, Reset: array[1..T] of Ytype init (true, 0);
    Rnd: array[1..T][0..∞] of boolean init false;
    Acquired: array[1..T] of boolean init false

private variables
    nd, n: 1..2T + 1;
    lvl, j: 0..D + 1;
    y: Ytype;   dir: Dtype;
    path: array[0..D] of Ptype

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:      X[nd], dir := p, S;
3:      y := Y[nd];
        if ¬y.free then dir := R
        else
4:          Y[nd] := (false, 0);
5:          if X[nd] ≠ p ∨
6:                Acquired[nd] then
              dir := L
            else
7:              Rnd[nd][y.rnd] := true;
8:              if Reset[nd] ≠ y then
9:                  Rnd[nd][y.rnd], dir := false, L
        fi fi fi;
10:     path[lvl] := (nd, dir);
        if dir ≠ S then
            lvl, nd := lvl + 1, 2 · nd;
            if dir = R then nd := nd + 1 fi
        fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
11:     Acquired[nd] := true;
        for j := lvl downto 0 do
12:         ENTRY_3(path[j].nd, path[j].dir)
        od;
13:     ENTRY_2(0)
    else     /* didn't get a name */
14:     ENTRY_N(p);
15:     ENTRY_2(1)
    fi;
16: Critical Section;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
        if path[j].dir ≠ R then
17:         n := path[j].nd;
18:         y := Reset[n];
19:         Reset[n] := (false, y.rnd);
20:         if j = lvl ∨
                ¬Rnd[n][y.rnd] then
21:             Reset[n] := (true, y.rnd + 1);
22:             Y[n] := (true, y.rnd + 1)
            fi
        fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
23:     EXIT_2(0);
        for j := 0 to lvl do
24:         EXIT_3(path[j].nd, path[j].dir)
        od;
25:     Acquired[nd] := false
    else
26:     EXIT_2(1);
27:     EXIT_N(p)
    fi
od
```

Fig. 4. Algorithm U: adaptive algorithm with unbounded memory.

integer *rnd* field. As explained below, Algorithm U works by associating "round numbers" with the various rounds of competition for the name corresponding to each splitter. In Algorithm U, these round numbers grow without bound. The *rnd* field of $Y[i]$ gives the current round number for splitter i. $Reset[i]$ is used to reinitialize the *rnd* field of $Y[i]$ when name i is released. $Rnd[i][r]$ is used to identify a potential "winning" process that has succeeded in acquiring name i in round r. $Acquired[i]$ is set when some process acquires name i.

Each process descends the renaming tree, starting at the root, until it either acquires a name or "falls off the end" of the tree, as discussed earlier. A process determines if it can acquire name i by executing statements 2-10 with $nd = i$. Of these, statements 2-5 correspond to the splitter code in Fig. 1. Statements 6-9 are executed as part of a handshaking mechanism that prevents a process that is releasing a name from adversely interfering with processes attempting to acquire that name; this mechanism is discussed in detail below. Statement 10 simply prepares for the next iteration of the **repeat** loop (if there is one).

If a process p succeeds in acquiring a name while descending within the renaming tree, then it competes within the renaming tree by moving up from its acquired name to the root, executing the three-process entry sections on this path (statements 11-12). Each of these three-process entry sections is denoted "$\texttt{ENTRY}_3(n, d)$," where n is the corresponding tree node, and d is the "identity" of the invoking process. The "identity" that is used is simply the invoking process's direction out of node n (S, L, or R) when it descended the renaming tree. After ascending the renaming tree, p invokes the two-process entry section "on top" of the renaming and overflow trees (as illustrated in Fig. 3(a)) using "0" as a process identifier (statement 13). This entry section is denoted "$\texttt{ENTRY}_2(0)$."

If a process p does *not* succeed in acquiring a name while descending within the renaming tree, then it competes within the overflow tree (statement 14), which is implemented using Yang and Anderson's N-process arbitration-tree algorithm. The entry section of this algorithm is denoted $\texttt{ENTRY}_N(p)$. Note that p uses its own process identifier in this algorithm. After competing within the overflow tree, p executes the two-process algorithm "on top" of both trees using "1" as a process identifier (statement 15). This entry section is denoted "$\texttt{ENTRY}_2(1)$."

After completing the appropriate two-process entry section, process p executes its critical section (statement 16). It then resets each of the splitters that it visited while descending the renaming tree (statements 17-22). This reset mechanism is discussed in detail below. Process p then executes the exit sections corresponding to the entry sections it executed previously (statements 23-27). The exit sections are specified in a manner that is similar to the entry sections.

We now consider in detail the code fragments that are executed to acquire (statements 2-10) or reset (statements 18-22) some splitter i. To facilitate this discussion, we will index these statements by i. For example, when we refer to the execution of statement $4[i]$ by process p, we mean the execution of statement 4 by p when its private variable *nd* equals i. Similarly, $18[i]$ denotes the execution of statement 18 with $n = i$.

As explained above, one of the problems with the splitter code is that it

is difficult for a left- or right-moving process at splitter i to know which (if any) process has acquired name i. In Algorithm U, this problem is solved by viewing the computation involving each splitter as occurring in a sequence of rounds. Each round ends when the splitter is reset. During a round, at most one process succeeds in acquiring the name of the splitter. Note that it is possible that *no* process acquires the name during a round. So that processes can know the current round number at splitter i, an additional *rnd* field has been added to $Y[i]$. This field will increase without bound over time, so we will never have to worry about round numbers being reused.

With the added *rnd* field, a left- or right-moving process at splitter i has a way of identifying a process that has acquired the name at splitter i. To see how this works, consider what happens during round r at node i. Of the processes that participate in round r at node i, at least one will read $Y[i] = (true, r)$ at statement $3[i]$ and assign $Y[i] := (false, 0)$ at statement $4[i]$. By the correctness of the original splitter code, of the processes that assign $Y[i]$, at most one will reach statement $7[i]$. A process that reaches statement $7[i]$ will either stop at node i or be deflected left. This gives us two cases to analyze: of the processes that read $Y[i] = (true, r)$ at statement $3[i]$ and assign $Y[i]$ at statement $4[i]$, either all are deflected left, or one, say p, stops at splitter i.

In the former case, at least one of the left-moving processes finds $Rnd[i][r]$ to be false at statement $20[i]$, and then reopens splitter i by executing statements $21[i]$ and $22[i]$, which establish $Y[i] = (true,\ r + 1)\ \wedge\ Y[i] = Reset[i]$. To see why at least one process executes statements $21[i]$ and $22[i]$, note that each process under consideration reads $Y[i] = (true, r)$ at statement $3[i]$, and thus its $y.rnd$ variable equals r while executing within statements $4[i]$-$9[i]$. Note also that $Rnd[i][r] = true$ is established only by statement $7[i]$. Moreover, each process deflected left at statement $9[i]$ first assigns $Rnd[i][r] := false$. Thus, at least one of the left-moving processes finds $Rnd[i][r]$ to be false at statement $20[i]$.

In the case that there is a winning process p that stops at splitter i during round r, we must argue that **(i)** p reopens splitter i upon leaving it, and **(ii)** no left- or right-moving process "prematurely" reopens splitter i before p has left it. Establishing (i) is straightforward. Process p will reopen the splitter by executing statements $18[i]$-$22[i]$ and 25, which establish $Y[i] = (true, r+1) \wedge Acquired[i] = false \wedge Y[i] = Reset[i]$. Note that the assignment to *Acquired* at statement 25 prevents the reopening of splitter i from actually taking effect until after p has finished executing its exit section.

To establish (ii), suppose, to the contrary, that some left- or right-moving process reopens splitter i by executing statement $22[i]$ while p is executing within statements $10[i]$-13 and 16-25. (Note that, because p stops at splitter i, it doesn't iterate again within the **repeat** loop.) Let q be the first left- or right-moving process to execute statement $22[i]$. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute statement $22[i]$ while p is executing within statements 16-22. Moreover, if p is executing within statements 12-13 or 23-25, then *Acquired* is true, and hence the splitter is closed. The remaining possibility is that p is enabled to execute statement $10[i]$ or 11. (Note that, in

this case, if q *were* to reopen splitter i then we could end up with two processes concurrently invoking $\texttt{ENTRY}_3(i, S)$ at statement 12, i.e., both processes use S as a "process identifier." The ENTRY calls obviously cannot be assumed to work correctly if such a scenario could happen.)

So, assume that q executes statement $22[i]$ while p is enabled to execute statement $10[i]$ or 11. For this to happen, q must have read $Rnd[i][r] = \mathit{false}$ at statement $20[i]$ before p assigned $Rnd[i][r] := \mathit{true}$ at statement $7[i]$. (Recall that all the processes under consideration read $Y[i] = (\mathit{true}, r)$ at statement $2[i]$. This is why p writes to $Rnd[i][r]$ instead of some other element of $Rnd[i]$. q reads from $Rnd[i][r]$ at statement $20[i]$ because it is the first process to attempt to reset splitter i, which implies that q reads $Reset[i] = (\mathit{true}, r)$ at statement $18[i]$.) Because q executes statement $20[i]$ before p executes statement $7[i]$, statement $19[i]$ is executed by q before statement $8[i]$ is executed by p. Thus, p must have found $Reset[i] \neq y$ at statement $7[i]$, i.e., it was deflected left at splitter i, which is a contradiction. It follows from the explanation given here that splitter i is eventually reset for round $r + 1$, i.e., we have the following property.

Property 1: Let $\mathcal{S}$ be the set of all processes that read $Y[i].rnd = r$ at statement $3[i]$. If $\mathcal{S}$ is nonempty, then $Y[i] = (\mathit{true},\ r+1) \wedge Y[i] = Reset[i] \wedge Acquired[i] = \mathit{false}$ is eventually established, and at all states after it is first established, no process in set $\mathcal{S}$ stops at splitter i. □

Because the splitters are always reset properly, it follows that the ENTRY and EXIT routines are always invoked properly. If these routines are implemented using Yang and Anderson's local-spin algorithm, then since that algorithm is starvation-free, Algorithm U is as well.

Having dispensed with basic correctness, we now informally argue that Algorithm U is contention sensitive. For a process p to descend to a splitter at level l in the renaming tree, it must have been deflected left or right at each prior splitter it accessed. Just as with the original grid-based long-lived renaming algorithm [12], this can only happen if the point contention experienced by p is $\Omega(l)$. Note that the time complexity per level of the renaming tree is constant. Moreover, with the ENTRY and EXIT calls implemented using Yang and Anderson's algorithm [14], the $\texttt{ENTRY}_2$, $\texttt{EXIT}_2$, $\texttt{ENTRY}_3$, and $\texttt{EXIT}_3$ calls take constant time, and the $\texttt{ENTRY}_N$ and $\texttt{ENTRY}_N$ calls take $\Theta(\log N)$ time. Note that the $\texttt{ENTRY}_N$ and $\texttt{ENTRY}_N$ routines are called by a process only if its point contention is $\Omega(\log N)$. Thus, we have the following.

Lemma 1: Algorithm U is a correct, starvation-free mutual exclusion algorithm with $O(min(k, \log N))$ time complexity and unbounded space complexity. □

Of course, the problem with Algorithm U is that the *rnd* field of $Y[i]$ that is used for assigning round numbers grows without bound. We now consider a variant of Algorithm U in which space is bounded.

Algorithm B. In Algorithm B (for bounded), which is shown in Fig. 5, modulo-N addition (denoted by $\oplus$) is used when incrementing $Y[i].rnd$. With this change,

the following potential problem arises. A process p may reach statement 8[i] in Fig. 5 with $y.rnd = r$ and then be delayed. While delayed, other processes may repeatedly increment $Y[i].rnd$ (statement 27[i]) until it "cycles back" to r. Another process q could then reach statement 8[i] with $y.rnd = r$. This is a problem because p and q may interfere with each other in updating $Rnd[i][r]$.

Algorithm B prevents such a scenario from happening by preventing $Y[i].rnd$ from cycling while a process p that stops at splitter i executes within statements 8[i]-31. Informally, cycling is prevented by requiring process p to erect an "obstacle" that prevents $Y[i].rnd$ from being incremented beyond the value p. More precisely, note that before reaching statement 8[i], process p must first assign $Obstacle[p] := i$ at statement 5[i]. Note further that before a process can increment $Y[i].rnd$ from r to $r \oplus 1$ (statement 27[i]), it must first read $Obstacle[r]$ (statement 25[i]) and find it to have a value different from i. This check prevents $Y[i].rnd$ from being incremented beyond the value p while p executes within statements 8[i]-31. Note that process p resets $Obstacle[p]$ to 0 at statement 18. This is done to ensure that p's own obstacle doesn't prevent it from incrementing a splitter's round number.

To this point, we have explained every difference between Algorithms U and B except one: in Fig. 5, there are added assignments to elements of Y and X (statements 20 and 21) after the critical section. The reason for these assignments is as follows. Suppose some process p is about to assign $Obstacle[p] := true$ at statement 5[i], but gets delayed. (In other words, p is "about to" erect an obstacle at splitter i.) We must ensure that if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p. Let r be the value read from $Y.rnd$ by p at statement 3[i]. For $Y.rnd$ to be incremented beyond p, some other process q that reads $Y.rnd = r$ must attempt to reopen splitter i.

So, suppose that process q reopens splitter i by executing statement 27[i] while p is delayed at statement 5[i]. If process q executes statement 21[i] after p executes statement 2[i], then p will find $X[i] \neq p$ at statement 6[i] and will be deflected left. So, assume that q executes statement 21[i] before p executes statement 2[i]. This implies that q establishes $Y[i].free = false$ by executing statement 20[i] before p reads $Y[i]$ at statement 3[i]. Note that $Y[i].free = true$ is only established within a critical section (statement 27[i]). Also, note that we have established the following sequence of statement executions (perhaps interleaved with statement executions of other processes): q executes statements 20[i] and 21[i]; p executes statements 2[i]-5[i]; q executes statement 27[i] (q's execution of statements 22[i]-26[i] may interleave arbitrarily with p's execution of statements 2[i]-5[i]). Because statements 17[i]-27[i] are executed as a critical section, this implies that p reads $Y[i].free = false$ at statement 3[i], and thus does not reach statement 5[i], which is a contradiction. We conclude from this reasoning that if p is delayed at statement 5[i], and if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p.

From the discussion above, we have the following property and lemma.

Property 2: If distinct processes p and q have executed statement 7[i] and have $nd = i$, then the value of p's private variable $y.rnd$ differs from that of q's. □

```
/* all variable declarations are as defined in Fig. 4 except as noted here */
type
    Ytype = record free: boolean; rnd: 0..N − 1 end        /* stored in one word */
shared variables
    X: array[1..T] of 0..N − 1;
    Rnd: array[1..T][0..N − 1] of boolean init false;
    Obstacle: array[0..N − 1] of 0..T init 0;
    Acquired: array[1..T] of boolean init false

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:      X[nd], dir := p, S;
3:      y := Y[nd];
        if ¬y.free then dir := R
        else
4:          Y[nd] := (false, 0);
5:          Obstacle[p] := nd;
6:          if X[nd] ≠ p ∨
7:                  Acquired[nd] then
                dir := L
            else
8:              Rnd[nd][y.rnd] := true;
9:              if Reset[nd] ≠ y then
10:                 Rnd[nd][y.rnd], dir := false, L
        fi fi fi;
11:     path[lvl] := (nd, dir);
        if dir ≠ S then
            lvl, nd := lvl + 1, 2 · nd;
            if dir = R then nd := nd + 1 fi
        fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
12:     Acquired[nd] := true;
        for j := lvl downto 0 do
13:         ENTRY3(path[j].nd, path[j].dir)
        od;
14:     ENTRY2(0)
    else     /* didn't get a name */
15:     ENTRY_N(p);
16:     ENTRY2(1)
    fi;
17: Critical Section;
18: Obstacle[p] := 0;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
        if path[j].dir ≠ R then
19:         n := path[j].nd;
20:         Y[n] := (false, 0);
21:         X[n] := p;
22:         y := Reset[n];
23:         Reset[n] := (false, y.rnd);
24:         if (j = lvl ∨
                    ¬Rnd[n][y.rnd]) ∧
25:                 Obstacle[y.rnd] ≠ n then
26:             Reset[n] := (true, y.rnd ⊕ 1);
27:             Y[n] := (true, y.rnd ⊕ 1)
            fi;
28:         if j = lvl then
                Rnd[y.rnd] := false
            fi
        fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
29:     EXIT2(0);
        for j := 0 to lvl do
30:         EXIT3(path[j].nd, path[j].dir)
        od;
31:     Acquired[nd] := false
    else
32:     EXIT2(1);
33:     EXIT_N(p)
    fi
od
```

Fig. 5. Algorithm B: adaptive algorithm with $\Theta(N^2)$ space complexity.

Lemma 2: Algorithm B is a correct, starvation-free mutual exclusion algorithm with $O(min(k, \log N))$ time complexity and $\Theta(N^2)$ space complexity. □

The $\Theta(N^2)$ space complexity of Algorithm B is due to the *Rnd* array. We now show that this $\Theta(N^2)$ array can be replaced by a $\Theta(N)$ linked list.

Algorithm L. In Algorithm L (for linear), which is depicted in Fig. 6, a common pool of round numbers ranging over $\{1, \ldots, U\}$ is used for all splitters in the renaming tree. As we shall see, $O(N)$ round numbers suffice. In Algorithm B, our key requirement for round numbers was that they not be reused "prematurely." With a common pool of round numbers, a process should not choose r as the next round number for some splitter if there is a process *anywhere* in the renaming tree that "thinks" that r is the current round number of some splitter.

Fortunately, since each process selects new round numbers within its critical section, it is fairly easy to ensure this requirement. All that is needed are a few extra data structures that track which round numbers are currently in use. These data structures replace the *Obstacle* array of Algorithm B. The main new data structure is a queue *Free* of round numbers. In addition, there is a new shared array *Inuse*, and a new shared variable *Check*. We assume that the *Free* queue can be manipulated by the usual *Enqueue* and *Dequeue* operations, and also by an operation *MoveToTail*(*Free*, i: 1..U), which moves i to the end of *Free*, if it is in *Free*. If *Free* is implemented as a doubly-linked list, then these operations can be performed in constant time. We stress that *Free* is accessed *only* within critical sections, so it is really a *sequential* data structure.

When comparing Algorithms B and L, the only difference before the critical section is statement 5[i]: instead of updating *Obstacle*[p], process p now marks the round number r it just read from $Y[i]$ as being "in use" by assigning *Inuse*[p] := r. The only other differences are in the code after the critical section (statements 18-33 in Fig. 6). Statements 24-27 are executed to ensure that no round number currently "in use" can propagate to the head of the *Free* queue. In particular, if a process p is delayed after having obtained r as the current round number for some splitter, then while it is delayed, r will be moved to the end of the *Free* queue by every N^{th} critical section execution. With $U = T + 2N$ round numbers, this is sufficient to prevent r from reaching the head of the queue while p is delayed. ($T + 2N$ round numbers are needed because the calls to *Dequeue* and *MoveToTail* can cause a round number to migrate toward the head of the *Free* queue by two positions per critical section execution.) Statement 28[i] enqueues the current round number for splitter i onto the *Free* queue. (Note that there may be other processes within the renaming tree that "think" that the just-enqueued round number is the current round number for splitter i; this is why we need a mechanism to prevent round numbers from prematurely reaching the head of the queue.) Statement 29[i] simply dequeues a new round number from *Free*. The rest of the algorithm is the same as before.

The space complexity of Algorithm L is clearly $\Theta(N)$, if we ignore the space required to implement the `ENTRY` and `EXIT` routines. (Each process has a $\Theta(\log N)$ *path* array. These arrays are actually unneeded, as simple calculations can be

```
/* all variable declarations are as defined in Fig. 5 except as noted here */
const  U = T + 2N                /* number of possible round numbers = O(N) */
type   Ytype = record free: boolean; rnd: 0..U end  /* stored in one word */

shared variables
  Y, Reset: array[1..T] of Ytype;
  Rnd: array[1..U] of boolean init false;
  Free: queue of integers;
  Inuse array[0..N − 1] of 0..U init 0;
  Check: 0..N − 1 init 0

initially
  (∀i : 1 ≤ i ≤ T :: Y[i] = (true, i) ∧
    Reset[i] = (true, i)) ∧
    (Free = (T + 1) → ··· → U)

private variables
  ptr: 0..N − 1; nxtrd: 1..U; usdrd: 0..U

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  nd, lvl := 1, 0;
    /* descend renaming tree */
    repeat
2:    X[nd], dir := p, S;
3:    y := Y[nd];
      if ¬y.free then dir := R
      else
4:      Y[nd] := (false, 0);
5:      Inuse[p] := y.rnd;
6:      if X[nd] ≠ p ∨
7:           Acquired[nd] then
          dir := L
        else
8:        Rnd[y.rnd] := true;
9:        if Reset[nd] ≠ y then
10:         Rnd[y.rnd], dir := false, L
      fi fi fi;
11:   path[lvl] := (nd, dir);
      if dir ≠ S then
        lvl, nd := lvl + 1, 2 · nd;
        if dir = R then nd := nd + 1 fi
      fi
    until (lvl > D) ∨ (dir = S);
    if lvl ≤ D then     /* got a name */
12:   Acquired[nd] := true;
      for j := lvl downto 0 do
13:     ENTRY3(path[j].nd, path[j].dir)
      od;
14:   ENTRY2(0)
    else     /* didn't get a name */
15:   ENTRY_N(p);
16:   ENTRY2(1)
    fi;

17: Critical Section;
    /* reset splitters */
    for j := min(lvl, D) downto 0 do
      if path[j].dir ≠ R then
18:     n := path[j].nd;
19:     Y[n] := (false, 0);
20:     X[n] := p;
21:     y := Reset[n];
22:     Reset[n] := (false, y.rnd);
23:     if j = lvl ∨ ¬Rnd[y.rnd] then
24:       ptr := Check;
25:       usdrd := Inuse[ptr];
26:       if usdrd ≠ 0 then
            MoveToTail(Free, usdrd)
          fi;
27:       Check := ptr ⊕ 1;
28:       Enqueue(Free, y.rnd);
29:       nxtrd := Dequeue(Free);
30:       Reset[n] := (true, nxtrd);
31:       Y[n] := (true, nxtrd)
        fi;
        if j = lvl then
32:       Rnd[y.rnd] := false;
33:       Inuse[p] := 0
      fi fi
    od;
    /* execute exit sections */
    if lvl ≤ D then
34:   EXIT2(0);
      for j := 0 to lvl do
35:     EXIT3(path[j].nd, path[j].dir)
      od;
36:   Acquired[nd] := false
    else
37:   EXIT2(1);
38:   EXIT_N(p)
    fi
od
```

Fig. 6. Algorithm L: adaptive algorithm with $\Theta(N)$ space complexity.

used to determine the parent and children of a splitter.) If the `ENTRY/EXIT` routines are implemented using Yang and Anderson's arbitration-tree algorithm [14], then the overall space complexity is actually $\Theta(N \log N)$. This is because in Yang and Anderson's algorithm, each process needs a distinct spin location for each level of the arbitration tree. However, as we will show in the full paper, it is quite straightforward to modify the arbitration-tree algorithm so that each process uses the same spin location at each level of the tree. This modified algorithm has $\Theta(N)$ space complexity. We conclude by stating our main theorem.

Theorem 1. *N-process mutual exclusion can be implemented under read/write atomicity with time complexity $O(min(k, \log N))$ and space complexity $\Theta(N)$.* □

Acknowledgement: Gary Peterson recently conjectured to us that adaptivity under the remote-memory-references time measure must necessitate $\Omega(N^2)$ space complexity. His conjecture led us to develop Algorithm L.

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 91–103, 1999.
2. J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning (full version of this paper). At `http://www.cs.unc.edu/~anderson/papers.html`.
3. J. Anderson and Y.-J. Kim. Fast *and* scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, September 1999. Full version to appear in *Distributed Computing*.
4. T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Sys.*, 1(1):6–16, 1990.
5. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. To appear in *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*.
6. M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
7. E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
8. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.
9. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Sys.*, 5(1):1–11, 1987.
10. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Sys.*, 9(1):21–65, 1991.
11. M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.
12. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.
13. E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 159–168. 1992.
14. J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

This article was processed using the LaTeX macro package with LLNCS style