A New Fast-Path Mechanism for Mutual Exclusion*

James H. Anderson and Yong-Jik Kim

Department of Computer Science

University of North Carolina at Chapel Hill

December 1999

Abstract

In 1993, Yang and Anderson presented an N-process algorithm for mutual exclusion under read/write atomicity that has $\Theta(\log N)$ time complexity, where "time" is measured by counting remote memory references. In this algorithm, instances of a two-process mutual exclusion algorithm are embedded within a binary arbitration tree. In the two-process algorithm that was used, all busy-waiting is done by "local spinning." Performance studies presented by Yang and Anderson showed that their N-process algorithm exhibits scalable performance under heavy contention. One drawback of using an arbitration tree, however, is that each process is required to perform $\Theta(\log N)$ remote memory operations even when there is no contention. To remedy this problem, Yang and Anderson presented a variant of their algorithm that includes a "fast-path" mechanism that allows the arbitration tree to be bypassed in the absence of contention. This algorithm has the desirable property that contention-free time complexity is $O(1)$. Unfortunately, the fast-path mechanism that was used caused time complexity under contention to rise to $\Theta(N)$ in the worst case. To this day, the problem of designing a read/write mutual exclusion algorithm with $O(1)$ time complexity in the absence of contention and $O(\log N)$ time complexity under contention has remained open. In this paper, we close this problem by presenting a fast-path mechanism that achieves these time complexity bounds when used in conjunction with Yang and Anderson's arbitration-tree algorithm.

Keywords: Fast mutual exclusion, local spinning, read/write atomicity, scalability, shared memory

*Work supported by NSF grants CCR 9732916 and CCR 9972211. The first author was also supported by an Alfred P. Sloan Research Fellowship. A preliminary version of this paper was presented at the 13th International Symposium on Distributed Computing [2].

1 Introduction

The mutual exclusion problem has been studied for many years, dating back to the seminal paper of Dijkstra [6]. In this problem, each of a set of N processes repeatedly executes a "critical section" of code. Each process's critical section is preceded by an "entry section" and followed by an "exit section." The objective is to design the entry and exit sections to ensure that at most one process executes its critical section at any time, and that each process in its entry section eventually enters its critical section.

Recent work on mutual exclusion has focused on the design of "scalable" algorithms that minimize the impact of the processor-to-memory bottleneck through the use of *local spinning*. A mutual exclusion algorithm is *scalable* if its performance degrades only slightly as the number of contending processes increases. In local-spin mutual exclusion algorithms, good scalability is achieved by requiring all busy-waiting loops to be read-only loops in which only locally-accessible shared variables are accessed that do not require a traversal of the processor-to-memory interconnect. 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.

A number of queue-based local-spin mutual exclusion algorithms have been proposed in which only $O(1)$ remote memory references are required for a process to enter and exit its critical section [3, 7, 9]. In each of these algorithms, waiting processes form a "spin queue." Read-modify-write instructions are used to enqueue a blocked process onto the end of the queue. Performance studies presented in several papers [3, 7, 9, 11] have shown that these algorithms scale well as contention increases.

In subsequent work, Yang and Anderson questioned whether read-modify-write operations are in fact *necessary* for scalable mutual exclusion [11]. The main contribution of their work was an N-process mutual exclusion algorithm with $\Theta(\log N)$ time complexity that uses only read and write operations. Somewhat surprisingly, experiments conducted by them showed that this algorithm is only slightly slower than the fastest queue locks. In Yang and Anderson's algorithm, instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree, as depicted in Figure 1(a). The entry and exit sections associated with the two links connecting a given node to its children constitute a two-process mutual exclusion algorithm. Initially, all processes start at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each link on this path. Upon exiting its critical section, a process traverses this path in reverse, this time executing the exit section of each link. Yang and Anderson's two-process algorithm is designed to ensure that all spins are local, even though the two processes that invoke the algorithm are determined dynamically. Because of the structure of the tree, $\Theta(\log N)$ remote memory references are required for a process to enter and exit its critical section.

Although Yang and Anderson's algorithm exhibits scalable performance, in complexity-theoretic terms,

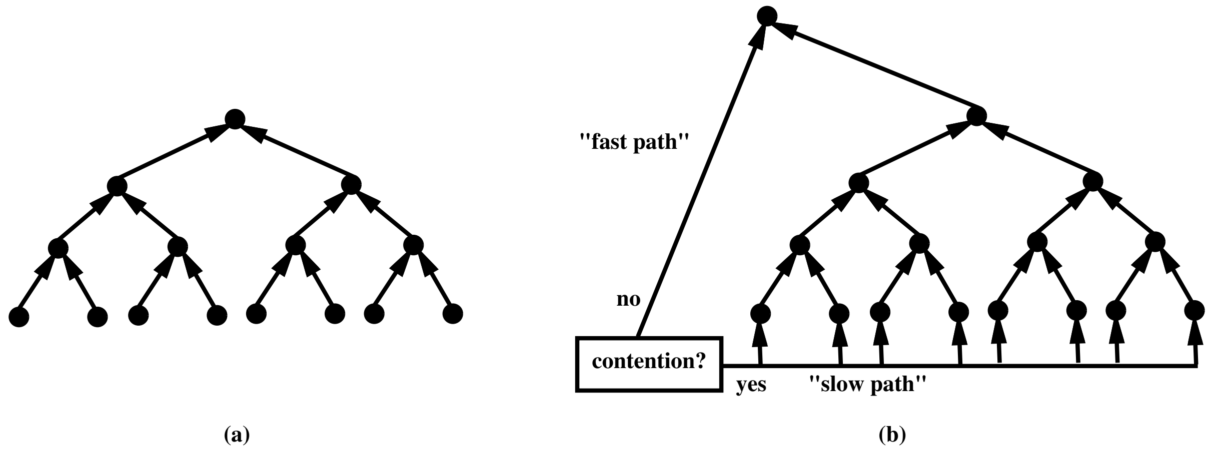

Figure 1: Yang and Anderson's arbitration-tree algorithm (inset (a)) and its fast-path variant (inset (b)).

there is still a gap between the $\Theta(\log N)$ time complexity of their algorithm and the constant time complexity of algorithms based on stronger synchronization primitives. This gap is particularly troubling when considering performance in the absence of contention. Even without contention, the arbitration tree forces each process to perform $\Theta(\log N)$ remote memory references in order to enter and exit its critical section.

To alleviate this problem, Yang and Anderson presented a variant of their algorithm that includes a "fast-path" mechanism that allows the arbitration tree to be bypassed in the absence of contention. This variant is illustrated in Figure 1(b). As the figure shows, an extra two-process mutual exclusion algorithm is placed on top of the arbitration tree to allow any fast-path process to compete with processes from the arbitration tree. This two-process algorithm is the same as that used inside the arbitration tree. The contention-detection mechanism in Yang and Anderson's algorithm is based on one proposed previously by Lamport [8]. Indeed, Lamport was the first to investigate "fast-path" mutual exclusion algorithms under read/write atomicity, and most subsequent fast-path algorithms (including the one in this paper) extend his original fast-path mechanism in some way. A description of Lamport's fast-path algorithm is given in the next section.

Yang and Anderson's fast-path algorithm has the desirable property that contention-free time complexity is $O(1)$. Unfortunately, it has the undesirable property that worst-case time complexity under contention is $\Theta(N)$, rather than $\Theta(\log N)$. In most fast-path algorithms (including Yang and Anderson's), once the fast path has been successfully "acquired" by a process, it must be "closed" so that multiple processes do not simultaneously acquire it. When a period of contention ends, the fast path must be "reopened." Because processes are asynchronous, and because only atomic reads and writes are being assumed, it is quite tricky to ensure that these properties hold. In Yang and Anderson's fast-path algorithm, a process checks whether the fast path can be reopened by "polling" each process individually to see if it is still contending. This polling loop is the reason why the time complexity of their algorithm is $\Theta(N)$ in the worst case.

To this day, the problem of designing a read/write mutual exclusion algorithm with $O(1)$ time complexity in the absence of contention and $O(\log N)$ time complexity under contention has remained open. In this paper, we close this problem by presenting a fast-path mechanism that achieves these time complexity bounds when used in conjunction with Yang and Anderson's arbitration-tree algorithm. Our fast-path mechanism has the novel feature that it can be reopened after a period of contention without having to poll each process individually to see if it is still contending. In our algorithm, a process reopens the fast path after executing its critical section if it detects that no other process has successfully acquired the fast path. Unfortunately, because processes execute asynchronously, it can be difficult to distinguish between a situation in which no process has acquired the fast path and a situation in which some process is "about to" acquire the fast path. We defend against this possibility by ensuring that, when the fast path is reopened, any process that is "about to" acquire the fast path is deflected to the slow path.

The rest of this paper is organized as follows. In Section 2, we present our fast-path algorithm. In presenting the algorithm, we give proof sketches for several of the key properties that show that the algorithm is correct. Formal proofs of these properties are given in appendix. We end the paper with concluding remarks in Section 3.

2 Fast-Path Algorithm

Our fast-path algorithm is the last of three algorithms considered in turn in this section. The first of these algorithms is the original fast-path algorithm of Lamport [8]. The second is an algorithm that allows Lamport's fast-path mechanism to be efficiently "reset" when contention ends, at the expense of using unbounded variables. The final algorithm is a variant of the second that uses only bounded variables. In explaining these three 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 explaining the basic mechanisms of each algorithm in a way that is as intuitive as possible. A formal, assertional correctness proof for the final algorithm is presented in an appendix.

Lamport's fast-path mechanism is defined by the code fragment shown in Figure 2. (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.) This code ensures that at most one process will "take the fast path." To see this, suppose to the contrary that two processes p and q are at statement 6. Let p be the process that executed statement 5 last. Because p found that $X = p$ held at statement 5, X is not written by any process between p's execution of statement 1 and p's execution of statement 5. Thus, q executed statement 5 before p executed statement 1. This implies that q executed statement 4 before p executed statement 2. Thus, p must have read $Y = \mathit{false}$ at statement 2 and then taken the slow path at statement 3, which is a contradiction.

```
shared variable X: 0..N − 1; Y: boolean initially true
private variable y: boolean

process p::
0:  Noncritical Section;
1:  X := p;
2:  y := Y;
3:  if ¬y then "compete with other processes (slow path)"
    else
4:      Y := false;
5:      if X ≠ p then "compete with other processes (slow path )"
        else
6:          "take the fast path"
```

Figure 2: Lamport's fast-path mechanism.

It is straightforward to see that, with the stated initial conditions, if one process executes the code fragment in Figure 2 in isolation, then that process will take the fast path. The problem with using this code is that, after a period of contention ends, it is difficult to "reopen" the fast path so that it can be acquired by other processes. If a process does succeed in taking the fast path, then that process can reopen the fast path itself by simply assigning $Y := true$. On the other hand, if no process succeeds in taking the fast path, then the fast path ultimately must be reopened by one of the slow-path processes. Unfortunately, because processes are asynchronous and communicate only by means of atomic read and write operations, it can be difficult for a slow-path process to know whether the fast path has been acquired by some process, and if it has, which process has acquired it.

The algorithm shown in Figure 3 uses unbounded memory to solve this problem. Before describing how this algorithm works, we first examine its basic structure. Four shared variables and an infinite shared array are used in the algorithm: X, Y, *Reset*, *Name_Taken* (the array), and *Infast*. Variables X and Y are as in Lamport's algorithm, with the exception that Y now has an additional integer *indx* field. As explained below, the unbounded algorithm works by "renaming" any process that acquires the fast path with a new temporary process identifier. The *indx* field of Y is used in renaming processes. The variable *Reset* is used to reinitialize the *indx* field of Y after a period of contention ends. The variable *Name_Taken*$[k]$ is used to indicate whether name k has been assigned to some process. The variable *Infast* is set to record that the fast path has been acquired by some process.

A process determines if it can access the fast path by executing statements 1-8; of these, statements 1-4 comprise Lamport's fast-path mechanism. If a process p detects any other competing process while executing within statements 1-8, then p is "deflected" out of the fast path and invokes either $SLOW1$ or $SLOW2$. $SLOW1$ is invoked if p has not updated any variables that must be reset in order to reopen the fast path. Otherwise, $SLOW2$ is invoked. A detailed explanation of the deflection mechanism is given below. If a process is not deflected, then it successfully acquires the fast path, which consists of statements 9-15. A process that either acquires the fast path or is deflected to $SLOW2$ attempts to reopen the fast path by

```
type Ytype = record free: boolean; indx: 0..∞ end          /* stored in one word */

shared variable
    X: 0..N − 1;
    Y, Reset: Ytype initially (true, 0);
    Name_Taken: array[0..∞] of boolean initially false;
    Infast: boolean initially false

private variable  y: Ytype

process p::                                  /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  X := p;
2:  y := Y;
    if ¬y.free then SLOW1()
    else
3:      Y := (false, 0);
4:      if (X ≠ p ∨
5:          Infast) then SLOW2()
        else
6:          Name_Taken[y.indx] := true;
7:          if Reset ≠ y then
8:              Name_Taken[y.indx] := false;
                SLOW2()
            else
9:              Infast := true;
10:             ENTRY_2(0);                  /* fast path */
11:                 Critical Section;
12:                 Reset := (true, y.indx + 1);
13:                 Y := (true, y.indx + 1);
14:             EXIT_2(0);
15:             Infast := false
    fi  fi  fi
od

procedure SLOW1()
16: ENTRY_N(p);
17:     ENTRY_2(1);
18:         Critical Section;
19:     EXIT_2(1);
20: EXIT_N(p)

procedure SLOW2()
21: ENTRY_N(p);
22:     ENTRY_2(1);
23:         Critical Section;
24:         y := Reset;
25:         Reset := (false, y.indx);
26:         if ¬Name_Taken[y.indx] then
27:             Reset := (true, y.indx + 1);
28:             Y := (true, y.indx + 1)
            fi;
29:     EXIT_2(1);
30: EXIT_N(p)
```

Figure 3: Fast-path algorithm with unbounded memory.

executing statements 12-15 or 24-28, respectively. A detailed explanation of how the fast path is reopened is given below.

We assume that a fast-path process competes with other processes by placing a two-process mutual exclusion algorithm "on top" of an N-process mutual exclusion algorithm, as depicted in Figure 1(b). Although the N-process algorithm in Figure 1(b) is an arbitration tree, this algorithm could be implemented by some other means. In our discussion of time complexity given later, we will assume that the underlying two-process and N-process mutual exclusion algorithms are implemented using Yang and Anderson's algorithm, but this is not required for correctness. (As discussed later, for the overall algorithm to be starvation-free, the underlying algorithms must be starvation-free.) A fast-path process executes the "topmost" two-process mutual exclusion algorithm using 0 as a virtual process identifier. The entry code for this algorithm is denoted "`ENTRY_2`(0)" in Figure 3 (see statement 10). The corresponding two-process exit code is denoted "`EXIT_2`(0)" (statement 14). Each process p that is deflected to $SLOW1$ or $SLOW2$ must first compete within the N-process mutual exclusion algorithm (using its own process identifier). The entry and exit code for the N-process mutual exclusion algorithm are denoted "ENTRY_N(p)" and "EXIT_N(p)," respectively (statements 16, 20, 21, and 30). After competing within the N-process algorithm, a deflected process accesses the "topmost" two-process algorithm using 1 as a virtual process identifier. The entry and exit code for this are denoted "ENTRY_2(1)" and "EXIT_2(1)," respectively (statements 17, 19, 22, and 29).

One critical property of the algorithm bears mentioning before we consider some of the algorithm's more complicated properties. In particular, note that many of the accesses to shared variables actually occur within code sequences that execute as critical sections (statements 12-13 and 24-28). Thus, when reasoning about the algorithm, we can assume that these code sequences do not interleave with one another. In fact, as we shall see, this assumption is not merely a matter of convenience — the algorithm's correctness relies *crucially* on the fact that these code sequences execute as critical sections.

As explained above, one of the problems with Lamport's fast-path code is that it is difficult for a slow-path process to know which (if any) process has acquired the fast path. In the unbounded algorithm in Figure 3, this problem is solved by including in Y an additional field, which is an identifier that is used to "rename" any process that acquires the fast path. This identifier will increase without bound over time, so we will never have to worry about the possibility that two processes are renamed with the same identifier. With this added field, a slow-path process has a way of identifying a process that has taken the fast path. To see how this works, consider what happens when, starting from the initial state, some set of processes execute their entry sections. At least one of these processes will read $Y = (true, 0)$ at statement 2 and assign $Y := (false, 0)$ at statement 3. By the correctness of Lamport's fast-path code, of the processes that assign Y, at most one will reach statement 6. A process that reaches statement 6 will either acquire the fast path by reaching statement 9, or will be deflected to $SLOW2$ at statement 8.

This gives us two cases to analyze: of the processes that read $Y = (true, 0)$ at statement 2 and assign Y at statement 3, either all are deflected to $SLOW2$, or one, say p, acquires the fast path. In the former case, at least one of the processes that executes $SLOW2$ finds $Name\_Taken[0]$ to be false at statement 26, and then reopens the fast path by executing statements 27 and 28, which establish $Y.free = true \ \wedge \ Y.indx > 0 \wedge Y = Reset$. To see why at least one process executes statements 27 and 28, note that each process under consideration reads $Y = (true, 0)$ at statement 2, and thus its $y.indx$ variable equals 0 while executing within statements 3-8. Note also that $Name\_Taken[0] = true$ is established only by statement 6. Furthermore, each process that is deflected to $SLOW2$ at statement 8 first assigns $Name\_Taken[0] := false$. Thus, at least one of the processes deflected to $SLOW2$ finds $Name\_Taken[0]$ to be false at statement 26.

In the case that some unique process p has acquired the fast path, we must argue that **(i)** the fast-path process p reopens the fast path upon leaving it, and **(ii)** no $SLOW2$ process "prematurely" reopens the fast path before p has left the fast path. Establishing (i) is straightforward. Process p will reopen the fast path by executing statements 12-15, which establish $Y = (true, 1) \ \wedge \ Infast = false \ \wedge \ Y = Reset$. Note that the assignment to *Infast* at statement 15 prevents the reopening of the fast path from actually taking effect until after p has finished executing `EXIT_2`(0).

To establish (ii), suppose, to the contrary, that some $SLOW2$ process reopens the fast path by executing statement 28 while p is executing within statements 9-15. Let q be the first such $SLOW2$ process to execute statement 28. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute statement 28 while p is executing within statements 11-13. Moreover, if p is enabled to execute statement 14 or 15, then *Infast* is false, and hence the fast path is closed. The remaining possibility is that p is enabled to execute statement 9 or 10. (Note that if p were enabled to execute statement 9, and if q were to reopen the fast path, then we could end up with two processes invoking `ENTRY_2` at statement 10 with a virtual process identifier of 0!) So, assume that q executes statement 28 while p is enabled to execute statement 9 or 10. For this to happen, q must have read $Name\_Taken[0] = \mathit{false}$ at statement 26 before p assigned $Name\_Taken[0] := \mathit{true}$ at statement 6. (Recall that all the processes under consideration read $Y = (\mathit{true}, 0)$ at statement 2. This is why p writes to $Name\_Taken[0]$ instead of some other element of $Name\_Taken$. q reads from $Name\_Taken[0]$ at statement 26 because it is the first process to attempt to reset the fast path, which implies that q reads $Reset = (\mathit{true}, 0)$ at statement 24.) Because q executes statement 26 before p executes statement 6, statement 25 is executed by q before statement 7 is executed by p. Thus, p must have found $Reset \neq y$ at statement 7, i.e., it was deflected to $SLOW2$, which is a contradiction. It follows from the explanation given here that after an initial period of contention ends, we must have $Y.\mathit{free} = \mathit{true} \wedge Y.\mathit{indx} > 0 \wedge Y = Reset \wedge Infast = \mathit{false}$. This argument can be applied inductively to show that the fast path is properly reopened after each period of contention ends.

From the discussion above, we have the following properties for the algorithm in Figure 3. (Corresponding properties for the bounded version of this algorithm are formally proved in an appendix.)

Property U1: If all processes are in their noncritical sections, then $Y.\mathit{free} = \mathit{true} \wedge Infast = \mathit{false} \wedge Y = Reset$ holds. □

Property U2: If some process is executing within statements 9-15 (i.e., its program counter is least 9 and at most 15), then no other process is executing within these statements. □

By Property U1, if some process p executes its entry section in the absence of any contention, i.e., all other processes remain in their noncritical sections, then p "takes the fast path," i.e., it executes its critical section at statement 11. By Property U2, at most one process may "take the fast path" at any time. Thus, the `ENTRY` and `EXIT` routines are invoked properly, and presuming their correctness, the algorithm in Figure 3 ensures that at most one process executes its critical section at any time.

Of course, the problem with this algorithm is that the *indx* field of Y that is used for renaming will continue to grow without bound. The algorithm of Figure 4 solves this problem by requiring $Y.\mathit{indx}$ to be incremented modulo-N. With $Y.\mathit{indx}$ being updated in this way, the following potential problem arises. A

```
type Ytype = record free: boolean; indx: 0..N − 1 end        /* stored in one word */

shared variable
    X: 0..N − 1;
    Y, Reset: Ytype initially (true, 0);
    Name_Taken, Obstacle: array[0..N − 1] of boolean initially false;
    Infast: boolean initially false

private variable y: Ytype

process p::                              /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  X := p;
2:  y := Y;
    if ¬y.free then SLOW1()
    else
3:      Y := (false, 0);
4:      Obstacle[p] := true;
5:      if (X ≠ p ∨
6:          Infast) then SLOW2()
        else
7:          Name_Taken[y.indx] := true;
8:          if Reset ≠ y then
9:              Name_Taken[y.indx] := false;
                SLOW2()
            else
10:             Infast := true;
11:             ENTRY_2(0);              /* fast path */
12:                 Critical Section;
13:                 Obstacle[p] := false;
14:                 Reset := (false, y.indx);
15:                 if ¬Obstacle[y.indx] then
16:                     Reset := (true, y.indx + 1 mod N);
17:                     Y := (true, y.indx + 1 mod N)
                    fi;
18:                 Name_Taken[y.indx] := false;
19:             EXIT_2(0);
20:             Infast := false
    fi  fi  fi
od

procedure SLOW1()
21: ENTRY_N(p);
22:     ENTRY_2(1);
23:         Critical Section;
24:     EXIT_2(1);
25: EXIT_N(p)

procedure SLOW2()
26: ENTRY_N(p);
27:     ENTRY_2(1);
28:         Critical Section;
29:         Y := (false, 0);
30:         X := p;
31:         y := Reset;
32:         Obstacle[p] := false;
33:         Reset := (false, y.indx);
34:         if (¬Name_Taken[y.indx] ∧
35:                 ¬Obstacle[y.indx]) then
36:             Reset := (true, y.indx + 1 mod N);
37:             Y := (true, y.indx + 1 mod N)
            fi;
38:     EXIT_2(1);
39: EXIT_N(p)
```

Figure 4: Fast-path algorithm.

process p may reach statement 7 in Figure 4 with $y.indx = k$ and then get delayed. While delayed, other processes may repeatedly increment $Y.indx$ (in $SLOW2$) until it "cycles back" to k. At this point, another process q may reach statement 7 with $y.indx = k$. This is a problem because p and q may interfere with each other in updating $Name\_Taken[k]$.

The algorithm in Figure 4 prevents such a scenario from happening by preventing $Y.indx$ from cycling while some process p executes within statements 7-18. Informally, cycling is prevented by requiring process p to erect an "obstacle" that prevents $Y.indx$ from being incremented beyond the value p. More precisely, note that before reaching statement 7, p must first assign $Obstacle[p] := true$ at statement 4. Note further that before a process can increment $Y.indx$ from n to $n+1$ **mod** N (statement 17 or 37), it must first check

$Obstacle[n]$ (statement 15 or 35) and find it to be false. This check prevents $Y.indx$ from being incremented beyond the value p while p executes within statements 7-18. The correctness of the code that reopens the fast path rests heavily on the fact that most of this code (statements 13-18 and 29-37) is executed within a critical section. Note that this obstacle-placement strategy might prevent the reopening of the fast path when it is actually safe to do so. For example, if p is the only process with $Obstacle[p] = true$ and if p is delayed at statement 5 with $X \neq p$, then p's obstacle may prevent other processes from reopening the fast path even though p is destined to be deflected to $SLOW2$. However, there is no harm in not reopening the fast path in a case like this, because the presence of an obstacle means there is contention.

To this point, we have explained every difference between the algorithms in Figures 3 and 4 except one: in Figure 4, there are added assignments to Y and X (statements 29 and 30) in $SLOW2$. The reason for these assignments is as follows. Suppose some process p is about to assign $Obstacle[p] := true$ at statement 4 but gets delayed. (In other words, p is "about to" erect an obstacle to prevent $Y.indx$ from cycling.) We must ensure that if p ultimately reaches statement 7, then $Y.indx$ does not get incremented beyond the value p. Let k be the value read from $Y.indx$ by p at statement 2. For $Y.indx$ to be incremented beyond p, some other process q that reads $Y.indx = k$ must attempt to reopen the fast path. If such a process q reopens the fast path by executing statement 17 while process p is delayed at statement 4, then by the correctness of Lamport's fast-path mechanism, process p will find $X \neq p$ at statement 5 and will be deflected to $SLOW2$.

Suppose instead that process q reopens the fast path by executing statement 37. As before, we assume this happens while process p is delayed at statement 4. If process q executes statement 30 after p executes statement 1, then p will find $X \neq p$ at statement 5 and will be deflected to $SLOW2$. So, assume that q executes statement 30 before p executes statement 1. This implies that q establishes $Y.free = false$ by executing statement 29 before p reads Y at statement 2. Note that $Y.free = true$ is only established within critical sections (statements 17 and 37). Also, note that we have established the following sequence of statement executions (perhaps interleaved with statement executions of other processes): q executes statements 29 and 30; p executes statements 1-4; q executes statement 37 (q's execution of statements 31-36 may interleave arbitrarily with p's execution of statements 1-4). Because statements 28-37 are executed as a critical section, this implies that p reads $Y.free = false$ at statement 2, and thus does not reach statement 4, which is a contradiction. We conclude from this reasoning that if p is delayed at statement 4, and if p ultimately reaches statement 7, then $Y.indx$ does not get incremented beyond the value p.

From the discussion above, we have the following properties for the algorithm in Figure 4.

Property B1: If all processes are in their noncritical sections, then $Y.free = true \wedge Infast = false \wedge Y = Reset$ holds. □

Property B2: If some process is executing within statements 10-20, then no other process is executing within these statements. □

Property B3: If some process p that read $Y.indx = k$ at statement 2 is executing within statements 6-9, and if $Y.free$ is true, then either $k \leq Y.indx \leq p$, or $Y.indx \leq p \leq k$, or $p \leq k \leq Y.indx$. Informally, $Y.indx$ is "trapped" between k and p (modulo-N) and cannot cycle. □

These properties follow from invariants that are formally established in the appendix. In particular, B1 follows from invariant (I26), B2 follows from invariant (I11), and B3 follows from invariants (I6) and (I19).

We now establish the time complexity of the algorithm. If the calls to `ENTRY_2`, `EXIT_2`, `ENTRY_N`, and `EXIT_N` are implemented using Yang and Anderson's algorithm [11], then each call to `ENTRY_2` and `EXIT_2` requires a constant number of remote memory operations, and each call to `ENTRY_N` and `EXIT_N` requires $O(\log N)$ remote memory operations. From the code in Figure 4, it should be clear that each process performs a constant number of memory operations, and hence a constant number of remote memory operations, outside of calls to these `ENTRY` and `EXIT` routines. (Note that there are no loops or recursive calls outside of these routines.) Thus, each process performs a constant number of remote memory operations in the absence of contention, and $O(\log N)$ remote memory operations under contention. Moreover, because Yang and Anderson's algorithm is starvation-free, our algorithm is too. Thus, we have the following theorem.

Theorem 1 *The algorithm in Figure 4 is a correct N-process mutual exclusion algorithm, and is starvation-free, provided the two-process and N-process mutual exclusion algorithms used to implement the* `ENTRY` *and* `EXIT` *routines are starvation-free. Moreover, if the* `ENTRY` *and* `EXIT` *routines are implemented using Yang and Anderson's algorithm, then each critical section access requires a constant number of remote memory operations in the absence of contention, and $O(\log N)$ remote memory operations in the presence of contention.* □

3 Concluding Remarks

We have presented a fast-path algorithm for mutual exclusion under read/write atomicity. When used in conjunction with Yang and Anderson's arbitration tree algorithm, each critical section access requires $O(1)$ remote memory references in the absence of contention, and $O(\log N)$ remote memory references in the presence of contention, where N is the number of processes. This is the first read/write mutual exclusion algorithm to achieve these time complexity bounds.

In presenting our fast-path algorithm, we have abstracted away from the details of the underlying algorithms used to implement the `ENTRY` and `EXIT` calls. With the `ENTRY_2`/`EXIT_2` calls in Figure 4 implemented using Yang and Anderson's two-process algorithm, our fast-path algorithm can be simplified slightly. In particular, the writes to *Infast* can be removed, and the test of *Infast* in statement 6 can be replaced by a test of a similar variable (specifically, the variable $C[0]$) used in Yang and Anderson's algorithm [11].

Results by Cypher have shown that read/write atomicity is too weak for implementing mutual exclusion with a constant number of remote memory references per critical section access [5]. The actual lower bound established by him is a slow growing function of N. We suspect that $\Omega(\log N)$ is probably a tight lower bound for this problem. At the very least, we know from Cypher's work that time complexity under contention must be a function of N. Thus, mechanisms for achieving constant time complexity in the absence of contention should remain of interest even if algorithms with better time complexity under contention are developed.

The problem of implementing a fast-path mechanism bears some resemblance to the wait-free long-lived renaming problem [10]. 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 invoke 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. By using such renaming operations, it is possible to speed up computations with time complexity that depends on name space size.

Thinking about connections to renaming actually led us to discover the fast-path algorithm described in this paper. In principle, a fast-path mechanism could be implemented by associating a name with the fast path and by having each process attempt to acquire that name in its entry section; a process that successfully acquires the fast-path name would release it in its exit section. Despite this rather obvious connection, the problem of implementing a fast-path mechanism is actually a much easier problem than the long-lived renaming problem. In particular, while a renaming algorithm must be wait-free, most of the steps involved in releasing a "fast-path name" can be done within a process's critical section. Our algorithm heavily exploits this fact.

In 1993, Choy and Singh initiated work on adaptive mutual exclusion under read/write atomicity [4]. The problem here is to design a mutual exclusion algorithm with time complexity that is a function of the number of contending processes. Unfortunately, the algorithms presented by Choy and Singh use busy-waiting loops that generate an unbounded number of remote memory references in the worst case, and thus they fail to qualify as adaptive algorithms under the time complexity measure adopted in this paper. Under this measure, an algorithm is adaptive if each critical section access requires $O(k)$ remote memory references, where k is the number of contending processes. Clearly, under this definition, $O(k)$ time complexity can be achieved only if all busy waiting is by means of local spinning. Since the publication of Choy and Singh's paper, the problem of designing an adaptive, local-spin mutual exclusion algorithm has remained open.

In recent work [1], we have closed this problem by showing that such an algorithm can be obtained by incorporating the fast-path mechanism of this paper within the grid-based, long-lived renaming algorithm of Moir and Anderson [10]. In their algorithm, processes move among "points" in a grid, where each point is defined by a small number of shared variables. The grid points are denoted (i, j), where $0 \leq i \leq N-1$, $0 \leq j \leq N-1$, and N is the number of processes. Each such point corresponds to a name. A process acquires a new name by moving among the grid points, starting with (0,0). If multiple processes concurrently access the same grid point, then they may "collide" and be deflected to other grid points. A process continues moving among the grid points until it successfully acquires the name associated with some grid point (for brevity, we will not describe the algorithm in any more detail than this; an inductive argument is used to show that each process eventually visits a grid point without colliding and successfully acquires the corresponding name [10]). To release a name, a process must attempt to "reopen" each of the grid points it visited in acquiring its name.

The main idea behind our adaptive algorithm is to allow an arbitration tree to form dynamically within the renaming grid of Moir and Anderson's algorithm. The tree grows when new grid points are visited, and shrinks when previously-visited grid points are reopened. In Moir and Anderson's algorithm, it takes $O(N)$ time to reopen a point in the grid. Using a mechanism that is similar to that used in this paper to reopen the fast path, we can reopen a grid point in only $O(1)$ time. As in this paper, the correctness of this mechanism relies crucially on the fact that much of the release code executes within a critical section, so a similar mechanism would not work in a *wait-free* renaming algorithm. In our adaptive algorithm, the arbitration tree may not remain balanced, but its height is bounded by the number of contending processes. Each node of the tree is a three-process version of Yang and Anderson's mutual exclusion algorithm [11], so time complexity is $O(k)$, where k is the number of contending processes.

Acknowledgement: We are grateful to the anonymous referees for their helpful suggestions.

Appendix: Correctness Proof

In this appendix, we formally prove that the algorithm in Figure 4 is correct. Specifically, we prove that the mutual exclusion property (at most one process executes it critical section at any time) holds and that the fast path is always open in the absence of contention. (The algorithm is easily seen to be starvation-free if the underlying algorithms used to implement the `ENTRY` and `EXIT` calls are starvation-free.) The following notational conventions will be used in the proof.

Notational Conventions: Unless stated otherwise, we assume i, j, and k range over $\{0..N-1\}$. We use $n.i$ to denote the statement with label n of process i, and $i.y$ to represent i's private variable y. Let S be a subset of the statement labels in process i. Then, $i@\{S\}$ holds if and only if the program counter for process i equals some value in S. (Note that if l is a statement label, then $i@\{l\}$ means that process i is *enabled* to execute statement l, i.e., it hasn't executed statement l yet.) As stated earlier, we assume that each labeled sequence of statements in Figure 4 is atomic (in particular, note that statement $5.i$ establishes either $i@\{6\}$ or $i@\{26\}$, depending on whether X equals i, and that statement $34.i$ establishes either $i@\{35\}$ or $i@\{38\}$, depending on the value of $Name\_Taken[i.y.indx]$). □

Definition: We define a process i to be *FAST-possible* if the condition $F(i)$, defined below, is true.

$$F(i) \;\equiv\; i@\{3..8, 10..20\} \;\wedge\; (i@\{3..5\} \;\Rightarrow\; X = i) \;\wedge\; (i@\{3..8\} \;\Rightarrow\; Reset = i.y) \qquad \square$$

Informally, this condition indicates that process i may *potentially* acquire the fast path. It does not necessarily mean that i is *guaranteed* to acquire the fast path: if $F(i)$ holds, then process i still can be deflected to $SLOW1$ or $SLOW2$. If $i@\{3..9\}$ holds and $F(i)$ is false, then then we define process i to be *FAST-disabled*. We will later show that it is impossible for a FAST-disabled process to acquire the fast path. We now turn our attention to establishing the mutual exclusion property.

Mutual Exclusion

We will establish the mutual exclusion property by proving that the conjunction of a number of assertions is an invariant. This proves that each of these assertions individually is an invariant. These invariants are listed below. For each invariant I, we prove **(i)** I holds initially, and **(ii)** for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. For convenience, we list here each invariant that is established in the proof of mutual exclusion.

I) Invariants that give conditions that must hold if a process is FAST-possible.

invariant $F(i) \;\wedge\; i@\{4..8, 10..17\} \;\Rightarrow\; Y = (false, 0)$ (I1)

invariant $F(i) \;\wedge\; i@\{8, 10..17\} \;\Rightarrow\; Name\_Taken[i.y.indx] = true$ (I2)

invariant $i@\{10..16\} \;\Rightarrow\; Reset.indx = i.y.indx$ (I3)

invariant $i@\{11..20\} \;\Rightarrow\; Infast = true$ (I4)

II) Invariants that prevent "cycling." These invariants are used to show that if $i@\{6..9\}$ holds and process i is FAST-disabled, then $Reset.indx$ must be "trapped" between $i.y.indx$ and i. Therefore, there is no way $Reset$ can cycle back, erroneously making process i FAST-enabled again.

invariant $i@\{3..5\} \;\wedge\; X = i \;\Rightarrow\; Reset = i.y$ (I5)

invariant $i@\{6..9\} \;\Rightarrow\; (i.y.indx \le Reset.indx \le i) \;\vee$
$(Reset.indx \le i \le i.y.indx) \;\vee\; (i \le i.y.indx \le Reset.indx)$ (I6)

invariant $i@\{6..9\} \;\wedge\; Reset.indx = i \;\Rightarrow\; \neg(\exists j :: j@\{16, 36\})$ (I7)

invariant $i@\{6..9\} \;\wedge\; i.y.indx = i \;\Rightarrow\; Reset.indx = i.y.indx$ (I8)

invariant $i@\{9\} \;\wedge\; Reset.indx = i.y.indx \;\Rightarrow\; Reset.free = false$ (I9)

III) Invariants showing that certain regions of code are mutually exclusive.

invariant $F(i) \;\wedge\; F(j) \;\wedge\; i \ne j \;\Rightarrow\; \neg(i@\{3..6\} \;\wedge\; j@\{3..8, 10..17\})$ (I10)

invariant $F(i) \;\wedge\; F(j) \;\wedge\; i \ne j \;\Rightarrow\; \neg(i@\{7, 8, 10..20\} \;\wedge\; j@\{7, 8, 10..20\})$ (I11)

invariant $F(i) \;\Rightarrow\; \neg(i@\{3..5\} \;\wedge\; j@\{31..37\})$ (I12)

invariant $F(i) \Rightarrow \neg(i@\{6,7\} \wedge j@\{34..37\})$ (I13)

invariant $F(i) \Rightarrow \neg(i@\{8,10..19\} \wedge j@\{35..37\})$ (I14)

invariant $i@\{6..9\} \wedge j@\{6..17\} \wedge i \neq j \Rightarrow i.y.indx \neq j.y.indx$ (I15)

IV) Miscellaneous invariants that are either trivial or follow almost directly from the mutual exclusion property (I22).

invariant $i@\{32,33\} \Rightarrow Reset = i.y$ (I16)

invariant $i@\{15,16,34..36\} \Rightarrow Reset = (false, i.y.indx)$ (I17)

invariant $i@\{30..37\} \Rightarrow Y = (false, 0)$ (I18)

invariant $Y.free = true \Rightarrow Y = Reset$ (I19)

invariant $i@\{3..20\} \Rightarrow i.y.free = true$ (I20)

invariant $(i@\{5..13, 26..32\}) = (Obstacle[i] = true)$ (I21)

V) Mutual exclusion (our goal).

invariant (Mutual exclusion) $|\{i :: i@\{12..19, 23, 24, 28..38\}\}| \leq 1$ (I22)

In establishing the above invariants, statements that might potentially establish $F(i)$ must be repeatedly considered. The following lemma shows that only one such statement must be considered.

Lemma 1: *If t and u are consecutive states such that $F(i)$ is false at t but true at u, and if each of* (I1) *through* (I22) *holds at t, then u is reached from t via the execution of statement $2.i$.*

Proof: The only statements that could potentially establish $F(i)$ are $2.i$ (which establishes $i@\{3..8, 10..20\}$ and may establish $Reset = i.y$), $5.i$ (which falsifies $i@\{3..5\}$), $8.i$ (which falsifies $i@\{3..8\}$), $1.i$ and $30.i$ (which establish $X = i$), and $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$ (which may establish $Reset = i.y$), where j is any arbitrary process. We now show that none of these statements other than $2.i$ can establish $F(i)$.

Statement $5.i$ can establish $i@\{6\}$, and hence $F(i)$, only if $X = i$ holds at t. (If $X \neq i$ holds at t, then $5.i$ establishes $i@\{26\}$, which implies that $F(i)$ is false.) By (I5), if $X = i$ holds at t, then $Reset = i.y$ holds at t as well. By the definition of $F(i)$, this implies that $F(i)$ holds at t, a contradiction.

Statement $8.i$ can establish $i@\{10\}$, and hence $F(i)$, only if $Reset = i.y$ holds at t. But this implies that $F(i)$ holds at t, a contradiction.

Statements $1.i$, $30.i$, and $31.i$ establish $i@\{2, 31, 32\}$. Therefore, they cannot establish $F(i)$.

Statements $14.j$ and $33.j$ could establish $F(i)$ only if $i@\{3..8\} \wedge Reset \neq i.y$ holds at t, and upon executing $14.j$ or $33.j$, $Reset = i.y$ is established. However, by (I3) and (I16), $14.j$ and $33.j$ can change the value of $Reset$ only by changing the value of $Reset.free$ from $true$ to $false$. By (I20), if $i@\{3..8\}$ holds at t, then $i.y.free = true$ holds as well. Thus, statements $14.j$ and $33.j$ cannot possibly establish $Reset = i.y$, and hence cannot establish $F(i)$.

Statements $16.j$ and $36.j$ likewise can establish $F(i)$ only if $i@\{3..8\} \wedge Reset \neq i.y$ holds at t. We consider two cases, depending on whether $i@\{3..5\}$ or $i@\{6..8\}$ holds at t. If $i@\{3..5\} \wedge Reset \neq i.y$ holds at t, then by (I5), $X \neq i$ holds at t. This implies that $X \neq i$ holds at u as well, i.e., $F(i)$ is false at u.

Now, suppose that $i@\{6..8\} \wedge Reset \neq i.y$ holds at t. By (I17), statements $16.j$ and $36.j$ increment $Reset.indx$ by 1 modulo-N. Therefore, they may establish $F(i)$ only if $Reset.indx = (i.y.indx - 1) \textbf{ mod } N$ holds at t. By (I6), this implies that $i = Reset.indx$ or $i = i.y.indx$ holds at t. By (I8), the latter implies that $i = Reset.indx$ holds at t. Hence, in either case, $i = Reset.indx$ holds at t. Because we have assumed that $i@\{6..8\} \wedge j@\{16, 36\}$ holds at t, by (I7), we have a contradiction. Therefore, statements $16.j$ and $36.j$ cannot establish $F(i)$. □

We now prove each invariant listed above. Recall that our strategy is to show, for each invariant I, **(i)** I holds initially, and **(ii)** for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. In proving (ii), we do not consider statement executions that trivially do not affect I.

invariant $F(i) \wedge i@\{4..8, 10..17\} \Rightarrow Y = (false, 0)$ (I1)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I1) is true. To prove that (I1) is not falsified, it suffices to consider only those statements that may establish the antecedent or falsify the consequent. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. However, $2.i$ establishes $i@\{3\}$ and thus cannot establish the antecedent. The condition $i@\{4..8, 10..17\}$ may be established only by statement $3.i$, which also establishes the consequent.

The consequent may be falsified only by statements $17.j$ or $37.j$, where j is any arbitrary process. If $j = i$, then both $17.j$ and $37.j$ establish $i@\{18, 38\}$, which implies that the antecedent is false.

Suppose that $j \neq i$. By (I10) and (I11), the antecedent and $j@\{17\}$ cannot both hold simultaneously (recall that $j@\{17\}$ implies $F(j)$, by definition). Hence, statement $17.j$ cannot be executed while the antecedent holds. Similarly, by (I12), (I13), and (I14), the antecedent and $j@\{37\}$ cannot hold simultaneously. Hence, statement $37.j$ also cannot be executed while the antecedent holds. □

invariant $F(i) \wedge i@\{8, 10..17\} \Rightarrow Name\_Taken[i.y.indx] = true$ (I2)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I2) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. However, $2.i$, establishes $i@\{3\}$ and hence cannot establish the antecedent. The condition $i@\{8, 10..17\}$ may be established only by statement $7.i$, which also establishes the consequent.

The consequent may be falsified only by statements $2.i$ and $31.i$ (which may change the value of $i.y.indx$) and $9.j$ and $18.j$ (which assign the value *false* to an element of the $Name\_Taken$ array), where j is any arbitrary process. Statements $2.i$ and $31.i$ establish $i@\{3, 21, 32\}$, which implies that the antecedent is false. If $j = i$, then $9.j$ and $18.j$ establish $i@\{19, 26\}$, which implies that the antecedent is false.

Suppose that $j \neq i$. In this case, statement $9.j$ may falsify the consequent only if $i.y.indx = j.y.indx$ holds. By (I15) (with i and j exchanged), $j@\{9\} \wedge i.y.indx = j.y.indx$ implies that the antecedent of (I2) is false. Thus, $9.j$ cannot falsify (I2). Similarly, by (I11), if $j@\{18\}$ holds (which implies that $F(j)$ holds), then the antecedent of (I2) is false. Thus, $18.j$ also cannot falsify (I2). $\square$

invariant $i@\{10..16\} \Rightarrow Reset.indx = i.y.indx$ (I3)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I3) is true. The antecedent may be established only by statement $8.i$, which does so only if $Reset = i.y$ holds. Therefore, statement $8.i$ preserves (I3).

The consequent may be falsified only by statements $2.i$ and $31.i$ (which may change the value of $i.y.indx$) and $14.j$, $16.j$, $33.j$, and $36.j$ (which update $Reset$), where j is any arbitrary process. The antecedent is false after the execution of $2.i$ and $31.i$ and also after the execution of $16.j$, $33.j$, and $36.j$ if $j = i$. If $j = i$, then statement $14.j$ preserves the consequent.

Consider $14.j$, $16.j$, $33.j$, and $36.j$, where $j \neq i$. By (I11), the antecedent of (I3) and $j@\{14, 16\}$ cannot hold simultaneously (recall that $i@\{10..16\} \Rightarrow F(i)$ and $j@\{14, 16\} \Rightarrow F(j)$). Similarly, by (I14), the antecedent and $j@\{36\}$ cannot hold simultaneously. Hence, statements $14.j$, $16.j$, and $36.j$ can be executed only when the antecedent is false, and thus do not falsify (I3). By (I16), statement $33.j$ cannot change the value of $Reset.indx$. Hence, it does not falsify (I3). $\square$

invariant $i@\{11..20\} \Rightarrow Infast = true$ (I4)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I4) is true. The antecedent may be established only by statement $10.i$, which also establishes the consequent. The consequent may be falsified only by statement $20.j$, where j is any arbitrary process. If $j = i$, then statement $20.j$ also falsifies the antecedent. If $j \neq i$, then by (I11), the antecedent and $j@\{20\}$ cannot hold simultaneously. Hence, the antecedent is false after the execution of statement $20.j$. $\square$

invariant $i@\{3..5\} \wedge X = i \Rightarrow Reset = i.y$ (I5)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I5) is true. The antecedent may be established only by statements $1.i$ and $30.i$ (which establish $X = i$) and $2.i$ (which may establish $i@\{3..5\}$). However, $1.i$ and $30.i$ establish $i@\{2, 31\}$ and hence cannot establish the antecedent. Also, by (I19), if statement $2.i$ establishes $i@\{3..5\}$, then it also establishes $Reset = i.y$.

The consequent may be falsified only by statements $2.i$ and $31.i$ (which may change the value of $i.y$) and $14.j$, $16.j$, $33.j$, and $36.j$ (which update $Reset$), where j is any arbitrary process. However, statement $2.i$ preserves (I5) as shown above. Furthermore, the antecedent is false after the execution of $31.i$ and also after the execution of each of $14.j$, $16.j$, $33.j$, and $36.j$ if $j = i$.

Consider $14.j$, $16.j$, $33.j$, and $36.j$, where $j \neq i$. If the antecedent and consequent of (I5) both hold, then $F(i)$ holds by definition. If $j \neq i$, then by (I10) and (I12), $j@\{14, 16, 33, 36\}$ cannot hold as well. Hence, these statements cannot falsify (I5). □

invariant $i@\{6..9\} \Rightarrow (i.y.indx \leq Reset.indx \leq i) \vee$
$$(Reset.indx \leq i \leq i.y.indx) \vee (i \leq i.y.indx \leq Reset.indx) \tag{I6}$$

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I6) is true. The antecedent may be established only if statement $5.i$ is executed when $X = i$ holds. In this case, by (I5), $Reset = i.y$ holds, so the consequent is preserved.

The consequent may be falsified only by statements $2.i$ and $31.i$ (which may change the value of $i.y.indx$) and $14.j$, $16.j$, $33.j$, and $36.j$ (which may change the value of $Reset.indx$), where j is any arbitrary process. The antecedent is false after the execution of $2.i$ and $31.i$ and also after the execution of each of $14.j$, $16.j$, $33.j$, and $36.j$ if $j = i$.

Consider statements $16.j$ and $36.j$, where $j \neq i$. By (I17), these statements increment $Reset.indx$ by 1 modulo-N. Therefore, these statements may falsify the consequent only if $Reset.indx = i$ holds before execution (this can be seen by considering each disjunct of the consequent of (I6) in turn; note that if the third disjunct holds, then incrementing $Reset.indx$ by 1 modulo-N either preserves this disjunct or establishes the second disjunct). However, if $Reset.indx = i$ holds before the execution of $16.j$ or $36.j$, then by (I7), the antecedent of (I6) is false. Thus, statements $16.j$ and $36.j$ cannot falsify (I7).

Finally, consider statements $14.j$ and $33.j$, where $j \neq i$. By (I3) and (I16), these statements do not change $Reset.indx$. Hence, they cannot falsify the consequent. □

invariant $i@\{6..9\} \wedge Reset.indx = i \Rightarrow \neg(\exists j :: j@\{16, 36\})$ (I7)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I7) is true. The antecedent may be established only by statements $5.i$ (which may establish $i@\{6..9\}$) and $14.k$, $16.k$, $33.k$, and $36.k$ (which may change the value of $Reset.indx$), where k is any arbitrary process. Statement $5.i$ establishes the antecedent only if executed when $X = i$ holds. In this case, by (I5), $Reset = i.y$ holds, and hence $F(i)$ is true. By (I10) and (I12), this implies that $\neg(\exists j :: j@\{16, 36\})$ also holds. This implies that statement $5.i$ cannot falsify (I7).

If $k = i$, then the antecedent is false after the execution of each of $14.k$, $16.k$, $33.k$, and $36.k$. Suppose that $k \neq i$. By (I22), $(\forall j :: k@\{16, 36\} \wedge j@\{16, 36\} \Rightarrow k = j)$ holds. Therefore, $16.k$ and $36.k$ both establish $(\forall j :: \neg j@\{16,36\})$, which is equivalent to the consequent. Now, consider statements $14.k$ and $33.k$, where $k \neq i$. By (I3) and (I16), these statements do not change $Reset.indx$. It follows that, although statements $14.k$ and $33.k$ may preserve the antecedent, they do not establish it.

The consequent may be falsified only if either statement $15.j$ or $35.j$ is executed when $Obstacle[j.y.indx] = false$ holds. However, if the antecedent of (I7) and $j@\{15,35\}$ both hold, then the following hold: $Reset.indx = j.y.indx$, by (I17), $Reset.indx = i$, by the antecedent, and $Obstacle[i] = true$, by (I21). Taken together, these assertions imply that $Obstacle[j.y.indx] = true$. From this, we conclude that statements $15.j$ and $35.j$ cannot falsify the consequent while the antecedent holds. □

invariant $i@\{6..9\} \wedge i.y.indx = i \Rightarrow Reset.indx = i.y.indx$ (I8)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I8) is true. The antecedent may be established only by statements $5.i$ (which may establish $i@\{6..9\}$) and $2.i$ and $31.i$ (which may change the value of $i.y.indx$). However, statements $2.i$ and $31.i$ establish $i@\{3,32\}$, which implies that the antecedent is false. Furthermore, by (I5), if statement $5.i$ establishes $i@\{6..9\}$, then the consequent holds.

The consequent may be falsified only by statements $2.i$ and $31.i$ (which may change the value of $i.y.indx$) and $14.j$, $16.j$, $33.j$, and $36.j$ (which update $Reset$), where j is any arbitrary process. However, the antecedent is false after the execution of $2.i$ and $31.i$ and also after the execution of each of $14.j$, $16.j$, $33.j$, and $36.j$ if $j = i$.

Consider statements $14.j$, $16.j$, $33.j$, and $36.j$, where $j \neq i$. By (I3) and (I16), statements $14.j$ and $33.j$ do not change $Reset.indx$, and hence cannot falsify the consequent. Note also that, by (I7), the antecedent, the consequent, and $j@\{16,36\}$ cannot all hold simultaneously. Hence, statements $16.j$ and $36.j$ cannot falsify the consequent when the antecedent holds. □

invariant $i@\{9\} \wedge Reset.indx = i.y.indx \Rightarrow Reset.free = false$ (I9)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I9) is true. (I9) may be falsified only by statements $8.i$ (which may establish $i@\{9\}$), $2.i$ and $31.i$ (which may change the value of $i.y.indx$), and $14.j$, $16.j$, $33.j$, and $36.j$ (which update $Reset$), where j is any arbitrary process. Statements $2.i$ and $31.i$ establish $i@\{3,32\}$, which implies that the antecedent is false. Statement $8.i$ establishes the antecedent only if executed when $Reset \neq i.y \wedge Reset.indx = i.y.indx$ holds, which implies that $Reset.free \neq i.y.free$. However, by (I20), $i.y.free = true$. Thus, $Reset.free = false$.

If $j = i$, then each of $14.j$, $16.j$, $33.j$, and $36.j$ establishes $i@\{15,17,34,37\}$, which implies that the antecedent is false.

Consider statements $14.j$, $16.j$, $33.j$, and $36.j$, where $j \neq i$. Statements $14.j$ and $33.j$ trivially establish or preserve the consequent. By (I17), statements $16.j$ and $36.j$ increment $Reset.indx$ by 1 modulo-N. Therefore, these statements may establish the antecedent of (I9) only if executed when $i@\{9\} \wedge Reset.indx = (i.y.indx - 1) \ \mathbf{mod}\ N$ holds. In this case, by (I6), $i = Reset.indx$ or $i = i.y.indx$ holds. By (I8), the latter implies that $i = Reset.indx$ holds. In either case, $i = Reset.indx$ holds. By (I7), this implies that $i@\{9\}$ is false. It follows that statements $16.j$ and $36.j$ cannot falsify (I9). □

invariant $F(i) \wedge F(j) \wedge i \neq j \Rightarrow \neg(i@\{3..6\} \wedge j@\{3..8, 10..17\})$ (I10)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and thus (I10) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. Therefore, the only statements that may falsify (I10) are $2.i$ and $2.j$. Without loss of generality, it suffices to consider only statement $2.i$.

Statement $2.i$ may establish $F(i) \wedge i@\{3..6\}$ only if $Y.free = true$. We consider two cases. First, suppose that $(\exists j : j \neq i :: F(j) \wedge j@\{3..5\})$ holds before $2.i$ is executed. In this case, $X = j$ holds by the definition of $F(j)$. Hence, $X \neq i$, which implies that $2.i$ does not establish $F(i)$. Second, suppose that $(\exists j : j \neq i :: F(j) \wedge j@\{6..8, 10..17\})$ holds before $2.i$ is executed. In this case, by (I1), $Y.free = false$. In either case, statement $2.i$ cannot establish $F(i) \wedge i@\{3..6\}$. □

invariant $F(i) \wedge F(j) \wedge i \neq j \Rightarrow \neg(i@\{7, 8, 10..20\} \wedge j@\{7, 8, 10..20\})$ (I11)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I11) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. However, $2.i$ establishes $i@\{3\}$ and hence cannot falsify (I11). The only other statements that could potentially falsify (I11) are $6.i$ and $6.j$, which may falsify the consequent. Without loss of generality, it suffices to consider only statement $6.i$.

By Lemma 1, statement $6.i$ may establish $F(i) \wedge i@\{7, 8, 10..20\}$ only if $F(i) \wedge Infast = false$ holds before execution. We consider two cases. First, suppose that $(\exists j : j \neq i :: F(j) \wedge j@\{7, 8, 10..17\})$ holds before the execution of $6.i$. In this case, by (I10), $F(i) \wedge i@\{6\}$ is false. This implies that $6.i$ cannot establish $F(i) \wedge i@\{7, 8, 10..20\}$. Second, suppose that $(\exists j : j \neq i :: F(j) \wedge j@\{18..20\})$ holds before $6.i$ is executed. In this case, by (I4), $Infast = true$. Hence, statement $6.i$ cannot establish $i@\{7..20\}$. □

invariant $F(i) \Rightarrow \neg(i@\{3..5\} \wedge j@\{31..37\})$ (I12)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I12) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. Therefore, the only statements that may falsify (I12) are $2.i$ and $30.j$, which may falsify the consequent.

Statement $2.i$ may falsify (I12) only if executed when $Y.free = true \ \wedge\ j@\{31..37\}$ holds, but this is precluded by (I18). Statement $30.j$ may falsify (I12) only if executed when $F(i) \ \wedge\ i@\{3..5\} \ \wedge\ i \neq j$ holds. Because statement $30.j$ falsifies $X = i$, it also falsifies $F(i) \ \wedge\ i@\{3..5\}$. Thus, it preserves (I12). □

invariant $F(i) \ \Rightarrow\ \neg(i@\{6,7\} \ \wedge\ j@\{34..37\})$ (I13)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I13) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. However, $2.i$ establishes $i@\{3\}$ and hence cannot falsify (I13). The only other statements that may potentially falsify (I13) are $5.i$ and $33.j$, which may falsify the consequent.

Statement $5.i$ may falsify (I13) only if executed when $F(i) \ \wedge\ j@\{34..37\}$ holds, but this is precluded by (I12). Statement $33.j$ may falsify (I13) only if executed when $F(i) \ \wedge\ i@\{6,7\}$ holds, which, by (I20), implies that $i.y.free = true$ holds. Because statement $33.j$ establishes $Reset.free = false$, $Reset \neq i.y$ holds after its execution, which implies that $F(i) \ \wedge\ i@\{6,7\}$ is false. Therefore, statement $33.j$ preserves (I13). □

invariant $F(i) \ \Rightarrow\ \neg(i@\{8,10..19\} \ \wedge\ j@\{35..37\})$ (I14)

Proof: Initially $(\forall i :: i@\{0\})$ holds, and hence (I14) is true. By Lemma 1, the only statement that can establish $F(i)$ is $2.i$. However, $2.i$ establishes $i@\{3\}$ and hence cannot falsify (I14). The only other statements that could potentially falsify (I14) are $7.i$ and $34.j$. Statement $7.i$ may falsify (I14) only if executed when $F(i) \ \wedge\ j@\{35..37\}$ holds, but this is precluded by (I13).

Statement $34.j$ may falsify (I14) only if executed when $F(i) \ \wedge\ i@\{8,10..19\} \ \wedge\ Name\_Taken[j.y.indx] = false$ holds. By (I22), $i@\{12..19\}$ and $j@\{34\}$ cannot hold simultaneously. Thus, $34.j$ could potentially falsify (I14) only if executed when $F(i) \ \wedge\ i@\{8,10,11\} \ \wedge\ Name\_Taken[j.y.indx] = false$ holds. In this case, $Name\_Taken[i.y.indx] = true$ holds as well, by (I2), as does $Reset.indx = i.y.indx$, by the definition of $F(i)$ and (I3). In addition, by (I17), $j@\{34\}$ implies that $Reset.indx = j.y.indx$ holds. Combining these assertions, we have $Name\_Taken[j.y.indx] = false \wedge Name\_Taken[j.y.indx] = true$, which is a contradiction. Hence, statement $34.j$ cannot falsify (I14). □

invariant $i@\{6..9\} \ \wedge\ j@\{6..17\} \ \wedge\ i \neq j \ \Rightarrow\ i.y.indx \neq j.y.indx$ (I15)

Proof: Initially $(\forall i :: i@\{0\})$ holds, so (I15) is true. The only statements that may falsify the consequent are $2.i$ and $31.i$ (which may change the value of $i.y.indx$) and $2.j$ and $31.j$ (which may change the value of $j.y.indx$). However, the antecedent is false after the execution of each of these statements.

The only statements that can establish the antecedent are $5.i$ and $5.j$. We show that $5.i$ does not falsify (I15); the reasoning for $5.j$ is similar. Statement $5.i$ can establish the antecedent only if executed when

$X = i$ holds. By (I5), this implies that $Reset = i.y$ holds, which in turn implies that $F(i)$ is true. So, assume that $X = i \wedge Reset = i.y \wedge F(i)$ holds before $5.i$ is executed. We analyze three cases, which are defined by considering the value of process j's program counter.

- **Case 1:** $j@\{6..8\}$ holds before $5.i$ is executed. In this case, because $F(i)$ is true, by (I10), $F(j)$ does not hold. Thus, we have $j@\{6..8\} \wedge \neg F(j)$, which implies that $Reset \neq j.y$. Because $Reset = i.y$, this implies that $i.y \neq j.y$. In addition, by (I20), we have $i.y.free = true \wedge j.y.free = true$. Hence, because $i.y \neq j.y$, we have $i.y.indx \neq j.y.indx$. Thus, the consequent of (I15) holds before, and hence after, $5.i$ is executed.

- **Case 2:** $j@\{9\}$ holds before $5.i$ is executed. In this case, we show that the consequent of (I15) holds before, and hence after, $5.i$ is executed. Assume to the contrary that $i.y.indx = j.y.indx$ holds before $5.i$ is executed. Then, because $Reset = i.y$ holds, we have $j.y.indx = Reset.indx$. By (I9), this implies that $Reset.free = false$. Because $Reset = i.y$ holds, this implies that $i.y.free = false$ holds. However, by (I20), we have $i.y.free = true$, which is a contradiction.

- **Case 3:** $j@\{10..17\}$ holds before $5.i$ is executed. In this case, by (I10), $F(i) \wedge i@\{5\}$ is false, which is a contradiction. $\Box$

invariant $i@\{32, 33\} \Rightarrow Reset = i.y$ (I16)

invariant $i@\{15, 16, 34..36\} \Rightarrow Reset = (false, i.y.indx)$ (I17)

invariant $i@\{30..37\} \Rightarrow Y = (false, 0)$ (I18)

invariant $Y.free = true \Rightarrow Y = Reset$ (I19)

Proof Sketch: By (I22), all writes to $Reset$ (statements 14, 16, 33, 36) and to Y, except for statement 3 (statements 17, 29, 37), are within mutually exclusive regions of code. Given this and the initial condition $Y = (true, 0) \wedge Reset = (true, 0)$, each of these invariants easily follows. Note that statement 3 establishes $Y = (false, 0)$, and hence cannot falsify either (I18) or (I19). Note also that (I1) and (I18) imply that statements 14, 16, 33, and 36 (which update $Reset$) cannot falsify (I19). $\Box$

invariant $i@\{3..20\} \Rightarrow i.y.free = true$ (I20)

invariant $(i@\{5..13, 26..32\}) = (Obstacle[i] = true)$ (I21)

Proof: These invariants follow easily from the program text and the initial condition $(\forall i :: Obstacle[i] = false)$. $\Box$

invariant **(Mutual exclusion)** $|\{i :: i@\{12..19, 23, 24, 28..38\}\}| \leq 1$ (I22)

Proof: From the specification of `ENTRY_2/EXIT_2` and `ENTRY_N/EXIT_N`, (I22) may fail to hold only if two processes simultaneously execute within statements 10-20. However, this is precluded by (I11). $\Box$

Fast Path is Always Open in the Absence of Contention

Having shown that the mutual exclusion property holds, we now prove that when all processes are within their noncritical sections, the fast path is open. This property is formally captured by (I26) given below. Before proving (I26), we first establish three other invariants.

invariant $\mathit{Name\_Taken}[k] = \mathit{true} \Rightarrow (\exists i :: i@\{8..18\} \wedge k = i.y.\mathit{indx})$ (I23)

Proof: Initially $(\forall k :: \mathit{Name\_Taken}[k] = \mathit{false})$ holds, and hence (I23) is true. The antecedent can only be established if, for some process i, statement $7.i$ is executed, where $i.y.\mathit{indx} = k$. However, in this case, $7.i$ also establishes the consequent. The only statements that may falsify the consequent are $2.i$ and $31.i$ (which may change the value of $i.y.\mathit{indx}$) and $9.i$ and $18.i$ (which may falsify $i@\{8..18\}$). Statements $2.i$ and $31.i$ cannot be executed when $i@\{8..18\} \wedge k = i.y.\mathit{indx}$ holds. Statements $9.i$ and $18.i$ falsify the antecedent. $\Box$

invariant $(\forall i :: i@\{0..2, 18..25, 38, 39\}) \Rightarrow Y.\mathit{free} = \mathit{true}$ (I24)

Proof: Initially $Y.\mathit{free} = \mathit{true}$ holds, so (I24) is true. The only statements that can establish the antecedent are $15.i$, $17.i$, $34.i$, $35.i$, and $37.i$, where i is any process. Both $17.i$ and $37.i$ establish the consequent.

Statements $15.i$ and $35.i$ can establish the antecedent only if $\mathit{Obstacle}[k] = \mathit{true}$, where $k = i.y.\mathit{indx}$. By (I21), $\mathit{Obstacle}[k] = \mathit{true}$ implies that $k@\{5..13, 26..32\}$ holds, which implies that the antecedent is false.

Similarly, statement $34.i$ can establish the antecedent only if $\mathit{Name\_Taken}[i.y.\mathit{indx}] = \mathit{true}$. By (I23), this implies that $(\exists j :: j@\{8..18\} \wedge i.y.\mathit{indx} = j.y.\mathit{indx})$ holds. By (I22), $j@\{12..18\} \wedge i@\{34\}$ is false. It follows that $(\exists j :: j@\{8..11\} \wedge i.y.\mathit{indx} = j.y.\mathit{indx})$ holds, which implies that the antecedent is false.

The only statements that can falsify the consequent are $3.i$ and $29.i$, where i is any process. Both establish $i@\{4, 30\}$, which implies that the antecedent is false. $\Box$

invariant $\mathit{Infast} = \mathit{true} \Rightarrow (\exists i :: i@\{11..20\})$ (I25)

Proof: Initially $\mathit{Infast} = \mathit{false}$ holds, so (I25) is true. The only statement that can establish the antecedent is $10.i$, which also establishes the consequent. The only statement that can falsify the consequent is $20.i$, which also falsifies the antecedent. $\Box$

invariant (Fast path is open in the absence of contention)

$$(\forall i :: i@\{0\}) \Rightarrow Y.free = true \wedge Infast = false \wedge Y = Reset \tag{I26}$$

Proof: If $(\forall i :: i@\{0\})$ holds, then $Y.free = true$ holds by (I24), and $Infast = false$ holds by (I25). By (I19), $Y.free = true$ implies that $Y = Reset$ holds as well. □

References

[1] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. Unpublished manuscript.

[2] J. Anderson and Y.-J. Kim. Fast *and* scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, Springer-Verlag, September 1999.

[3] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[4] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 183–194. ACM, August 1993.

[5] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.

[6] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[7] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[8] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[9] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[10] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.

[11] J.-H. Yang and J. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–182. ACM, August 1993.