Fast *and* Scalable Mutual Exclusion[⋆]

James H.Anderson and Yong-Jik Kim

Department of Computer Science
University of North Carolina at Chapel Hill

Abstract. We present an N-process algorithm for mutual exclusion under read/write atomicity that has $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ time complexity under contention, where "time" is measured by counting remote memory references. This is the first such algorithm to achieve these time complexity bounds. Our algorithm is obtained by combining a new "fast-path" mechanism with an arbitration-tree algorithm presented previously by Yang and Anderson.

1 Introduction

Recent work on mutual exclusion [3] 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. A shared variable is locally accessible on a distributed shared-memory multiprocessor if it is stored in a local memory module, and on a cache-coherent multiprocessor 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 [1, 4, 6]. In each of these algorithms, waiting processes form a "spin queue". Read-modify-write instructions are used to enqueue a blocked process on this queue. Performance studies presented in [1, 4, 6] have shown that these algorithms scale well as contention increases.

In subsequent work, Yang and Anderson showed that performance comparable to that of the queue-lock algorithms cited above could be achieved using only read and write operations [8]. In particular, they presented a read/write mutual exclusion algorithm with $\Theta(\log N)$ time complexity and experimentally 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 Fig. 1(a). The entry and exit sections associated with the two links connecting

⋆ Work supported by NSF grant CCR 9732916. The first author was also supported by an Alfred P. Sloan Research Fellowship.

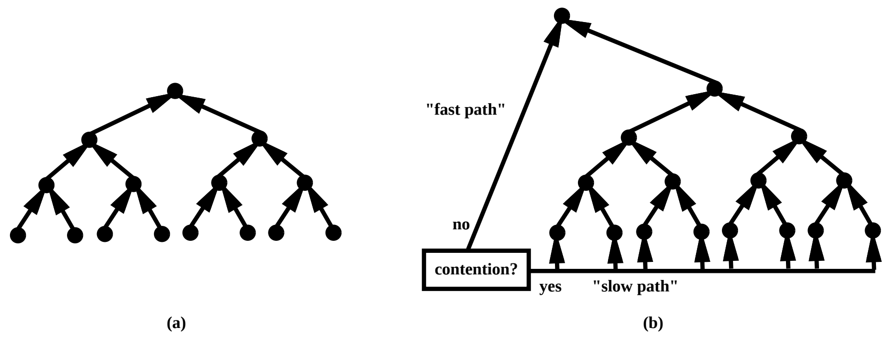

Fig. 1. Yang and Anderson's arbitration-tree algorithm (inset (a)) and its fast-path variant (inset (b)).

a given node to its sons constitute a two-process mutual exclusion algorithm. Initially, all processes start at the leaves of the tree. To enter its critical section, a process traverses the path from its leaf 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, executing the exit section of each link.

Although Yang and Anderson's algorithm exhibits scalable performance, in complexity-theoretic terms, 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 algorithm 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 Fig. 1(b). This algorithm has the desirable property that contention-free time complexity is $O(1)$. Unfortunately, it has the undesirable property that time complexity under contention is $\Theta(N)$ in the worst case, rather than $\Theta(\log N)$. In Yang and Anderson's fast-path algorithm, a process checks whether the fast path can be reopened after a period of contention ends 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 $\Theta(\log N)$ 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.

The rest of this paper is organized as follows. In Sec. 2, we present our fast-path algorithm. In Sec. 3, we prove that the algorithm is correct. We end the paper with concluding remarks in Sec. 4.

2 Fast-Path Algorithm

Our fast-path algorithm is shown in Fig. 2. In this section, we explain informally how the algorithm works. We begin with a brief overview of the code. We assume that each labeled sequence of statements in Fig. 2 is atomic; each such sequence reads or writes at most one shared variable. A process determines if it can access the fast path by executing statements 1-9. If a process p detects any other competing process while executing these statements, 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 10-20. A process that either acquires the fast path or is deflected to $SLOW2$ attempts to reopen the fast path by executing statements 13-20 or 29-37, respectively. A detailed explanation of how the fast path is reopened is given below.

Before entering its critical section, a fast-path process must perform the entry code of the two-process mutual exclusion algorithm on top of the arbitration tree, as shown in Fig. 1(b). It executes this code using 0 as a virtual process identifier. This is denoted as "`ENTRY_2`(0)" in Fig. 2 (see statement 11). The corresponding two-process exit code is denoted "`EXIT_2`(0)" (statement 19). Each process p that is deflected to $SLOW1$ or $SLOW2$ must first compete within the N-process arbitration tree (using its own process identifier). The entry and exit code for the arbitration tree are denoted "`ENTRY_N`(p)" and "`EXIT_N`(p)", respectively (statements 21, 25, 26, and 39). After competing within the arbitration tree, a deflected process accesses the two-process algorithm on top of the tree 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 22, 24, 27, and 38).

We now explain our fast-path acquisition mechanism in detail. At the heart of this mechanism is the following code fragment from Lamport's fast mutual exclusion algorithm [5].

```
shared variable $X$: $0..N-1$; $Y$: boolean initially $true$
process $p$::
    Noncritical Section;
    $X := p$;
    if $\neg Y$ then "compete with other processes (slow path)"
    else $Y := false$;
        if $X \neq p$ then "compete with other processes (slow path)"
        else "take the fast path"
```

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

shared variable
    $X$: $0..N-1$;
    $Y$, $Reset$: $Ytype$ initially $(true, 0)$;
    $Slot$, $Proc$: array$[0..N-1]$ of boolean initially $false$;
    $Infast$: boolean initially $false$

private variable $y$: $Ytype$

process $p$::          /* $0 \leq p < N$ */
while $true$ do
0:  Noncritical Section;
1:  $X := p$;
2:  $y := Y$;
    if $\neg y.free$ then $SLOW1()$
    else
3:    $Y := (false, 0)$;
4:    $Proc[p] := true$;
5:    if $(X \neq p \vee$
6:        $Infast)$ then $SLOW2()$
      else
7:      $Slot[y.indx] := true$;
8:      if $Reset \neq y$ then
9:        $Slot[y.indx] := false$;
          $SLOW2()$
        else
10:       $Infast := true$;
          /* fast path */
11:       ENTRY_2(0);
12:         Critical Section;
13:         $Proc[p] := false$;
14:         $Reset := (false, y.indx)$;
15:         if $\neg Proc[y.indx]$ then
16:           $Reset :=$
                $(true, y.indx + 1 \bmod N)$;
17:           $Y :=$
                $(true, y.indx + 1 \bmod N)$
            fi;
18:         $Slot[y.indx] := false$;
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:     $Proc[p] := false$;
33:     $Reset := (false, y.indx)$;
34:     if $(\neg Slot[y.indx] \wedge$
35:         $\neg Proc[y.indx])$ then
36:       $Reset :=$
            $(true, y.indx + 1 \bmod N)$;
37:       $Y :=$
            $(true, y.indx + 1 \bmod N)$
        fi;
38:   EXIT_2(1);
39: EXIT_N($p$)
```

Fig. 2. Fast-path algorithm.

This code ensures that at most one process will "take the fast path". Moreover, with the stated initial conditions, if one process executes this code fragment 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 resetting Y to *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.

As a stepping stone towards our algorithm, consider the algorithm shown in Fig. 3, which uses unbounded memory to solve the problem. In this algorithm, Y has 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 properties 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 $Y = (true, 0)$ and assign Y, 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 $SLOW2$ will increment the *indx* field of Y and set the *free* field of Y to *true* (statement 28). This has the effect of reopening the fast path. In the latter case, we must argue that **(i)** the fast-path process p reopens the fast path after 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 incrementing the *indx* field of Y and setting the *free* field of Y to *true* (statement 13). Note that the *Infast* variable 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 q reopens the fast path by executing statement 28 while p is executing within statements 9-15. For this to happen, q must have read $Slot[0]$ at statement 26 before p assigned $Slot[0] := true$ at statement 6. This in turn implies that q executed statement 25 before p executed statement 7. 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 $Y.free = true$ and $Y.indx > 0$. This argument can be applied inductively

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

shared variable                     /* other variable declarations are as in Fig. 2 */
    $Slot$: array$[0..\infty]$ of boolean initially $false$

process $p$::                        /* $0 \leq p < N$ */
while $true$ do
0: Noncritical Section;
1: $X := p$;
2: $y := Y$;
   if $\neg y.free$ then $SLOW1()$
   else
3:     $Y := (false, 0)$;
4:     if $(X \neq p \lor$
5:         $Infast)$ then $SLOW2()$
       else
6:         $Slot[y.indx] := true$;
7:         if $Reset \neq y$ then
8:             $Slot[y.indx] := false$;
               $SLOW2()$
           else
9:             $Infast := true$;
               /* fast path */
10:            ENTRY_2(0);
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 $\neg Slot[y.indx]$ then
27:            $Reset := (true, y.indx + 1)$;
28:            $Y := (true, y.indx + 1)$
           fi;
29:    EXIT_2(1);
30:EXIT_N($p$)
```

Fig. 3. Fast-path algorithm with unbounded memory.

to show that the fast path is properly reopened after each period of contention ends.

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 Fig. 2 solves this problem by requiring $Y.indx$ to be incremented modulo-N. With $Y.indx$ being updated in this way, the following potential problem arises. A process p may reach statement 7 in Fig. 2 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 $Slot[k]$. The algorithm in Fig. 2 prevents such a scenario from happening by preventing $Y.indx$ from cycling while some process executes within statements 7-18. To see how this is prevented, note that before reaching statement 7, a process p must first assign $Proc[p] := true$ 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 $Proc[n]$ (statement 15 or 35) and find it to be false. This check prevents $Y.indx$ from cycling while p executes within statements 7-18. As shown in the next section, the correctness of the code that reopens the fast path (statements 13-18 and 29-37) rests heavily on the fact that this code is executed within a critical section.

3 Correctness Proof

In this section, we prove that the algorithm in Fig. 2 is correct. Specifically, we prove that the mutual exclusion property (at most one process executes 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, given the correctness of `ENTRY` and `EXIT` calls.) 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 iff the program counter for process i equals some value in S. $\Box$

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

$$\begin{aligned} F(i) \ \equiv\ & i@\{3..8, 10..20\} \ \wedge \\ & (i@\{3..5\} \ \Rightarrow\ X = i) \ \wedge\ (i@\{3..8\} \ \Rightarrow\ Reset = i.y) \end{aligned} \qquad \Box$$

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 a process i is at $\{3..9\}$ and is not FAST-possible, then we define it to be *FAST-disabled*. We will later show that a FAST-disabled process cannot acquire the fast path. We now turn our attention to the mutual exclusion property.

3.1 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 numbered (I1) through (I22) and are stated on the following pages. Informally, invariants (I1) through (I4) give conditions that must hold if a process is FAST-possible. Invariants (I5) through (I9) 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. Invariants (I10) through (I15) show that certain regions of code are mutually exclusive. Invariants (I16) through (I21) are all simple invariants that follow almost directly from the code. Invariant (I22) is the mutual exclusion property, our goal.

In establishing these 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 $i@\{3..8\}$), $1.i$ (which establishes $X = i$), and $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, where j is any arbitrary process (which may establish $Reset = i.y$). 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. But, by (I5), this implies that $Reset = i.y$ 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 $1.i$ and $31.i$ establish $i@\{2, 32\}$. Thus, 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 $F(i)$. $\Box$

We now prove each invariant listed above. It is easy to see that each invariant hold initially, so we will not bother to prove this. For each invariant I, we show that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. In proving this, we do not consider statements that trivially don't affect I.

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

Proof: 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 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 both hold. Hence, statement $37.j$ also cannot be executed while the antecedent holds. □

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

Proof: 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 $2.i$, $31.i$, $9.j$, and $18.j$, 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 $i.y.\mathit{indx} = j.y.\mathit{indx}$ holds. By (I15) (with i and j exchanged), $j@\{9\} \wedge i.y.\mathit{indx} = j.y.\mathit{indx}$ 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). □

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

Proof: The antecedent may be established only by statement $8.i$, which does so only if $\mathit{Reset} = i.y$ holds. Therefore, statement $8.i$ preserves (I3).

The consequent may be falsified only by statements $2.i$, $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, 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 $\mathit{Reset}.\mathit{indx}$. Hence, it does not falsify (I3). □

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

Proof: 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 both hold. Hence, the antecedent is false after the execution of statement $20.j$. □

$$\textbf{invariant} \quad i@\{3..5\} \wedge X = i \Rightarrow \mathit{Reset} = i.y \tag{I5}$$

Proof: The antecedent may be established only by statements $1.i$ (which establishes $X = i$) and $2.i$ (which may establish $i@\{3..5\}$). However, $1.i$ establishes $i@\{2\}$ and hence cannot establish the antecedent. Also, by (I19), statement $2.i$ establishes the consequent.

The consequent may be falsified only by statements $2.i$, $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, 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). □

$$\textbf{invariant} \quad i@\{6..9\} \Rightarrow \begin{array}{l} (i.y.\mathit{indx} \le \mathit{Reset}.\mathit{indx} \le i) \vee \\ (\mathit{Reset}.\mathit{indx} \le i \le i.y.\mathit{indx}) \vee \\ (i \le i.y.\mathit{indx} \le \mathit{Reset}.\mathit{indx}) \end{array} \tag{I6}$$

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

The consequent may be falsified only by statements $2.i$, $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, 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 $\mathit{Reset}.\mathit{indx}$ by 1 modulo-N. Therefore, these statements may falsify the consequent only if $\mathit{Reset}.\mathit{indx} = i$ holds before execution. However, in this case, by (I7), the antecedent of (I6) is false. Thus, statements $16.j$ and $36.j$ cannot falsify (I7).

Finally, consider $14.j$ and $33.j$, where $j \neq i$. By (I3) and (I16), $14.j$ and $33.j$ don't change $\mathit{Reset}.\mathit{indx}$. Hence, they can't falsify the consequent. □

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

Proof: The antecedent may be established only by statements $5.i$, $14.k$, $16.k$, $33.k$, and $36.k$, where k is any arbitrary process. Statement $5.i$ establishes the antecedent only if executed when $X = i$ holds. In this case, by (I5), $\mathit{Reset} = i.y$, and hence $F(i)$, holds as well. 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 $36.k$. If $k \neq i$, then 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 by statements $15.j$ and $35.j$, which may do so only if $Proc[j.y.indx] = false$. 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 $Proc[i] = true$, by (I21). Taken together, these assertions imply that $Proc[j.y.indx] = true$. Therefore, statements $15.j$ or $35.j$ 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: The antecedent may be established only by statements $2.i$, $5.i$, and $31.i$. However, statements $2.i$ and $31.i$ establish $i@\{3,32\}$, which implies that the antecedent is false. Furthermore, by (I5), statement $5.i$ preserves the consequent.

The consequent may be falsified only by statements $2.i$, $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, 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 $16.j$, $33.j$, 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: (I9) may be falsified only by statements $2.i$, $8.i$, $31.i$, $14.j$, $16.j$, $33.j$, and $36.j$, 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 $i@\{9\} \wedge Reset.indx = (i.y.indx - 1) \textbf{ mod } N$ 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: 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 $Y.\mathit{free} = \mathit{true}$. 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), $Y.\mathit{free} = \mathit{false}$. In either case, statement $2.i$ cannot establish $F(i) \ \wedge \ i@\{3..6\}$. $\Box$

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: 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 $6.j$. 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 $F(i) \ \wedge \ \mathit{Infast} = \mathit{false}$ 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), $\mathit{Infast} = \mathit{true}$. Hence, statement $6.i$ cannot establish $i@\{7..20\}$. $\Box$

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

Proof: 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 $30.j$.

Statement $2.i$ may falsify (I12) only if executed when $Y.\mathit{free} = \mathit{true} \ \wedge \ j@\{31..37\}$ 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). $\Box$

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

Proof: 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 $33.j$.

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 $i.y.\mathit{free} = \mathit{true}$ holds. Because statement $33.j$ establishes $\mathit{Reset.free} = \mathit{false}$, $\mathit{Reset} \neq i.y$ holds after its execution, which implies that $F(i) \ \wedge \ i@\{6, 7\}$ is false. Therefore, statement $33.j$ preserves (I13). $\Box$

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

Proof: 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 $F(i) \wedge i@\{8, 10..19\} \wedge \mathit{Slot}[j.y.\mathit{indx}] = \mathit{false}$ holds. By (I22), $i@\{17..19\}$ and $j@\{34\}$ cannot hold simultaneously. Thus, $34.j$ could potentially falsify (I14) only if executed when $F(i) \wedge i@\{8, 10..16\} \wedge \mathit{Slot}[j.y.\mathit{indx}] = \mathit{false}$ holds. In this case, $\mathit{Slot}[i.y.\mathit{indx}] = \mathit{true}$ holds as well, by (I2), as does $\mathit{Reset}.\mathit{indx} = i.y.\mathit{indx}$, by the definition of $F(i)$ and (I3). In addition, by (I17), $j@\{34\}$ implies that $\mathit{Reset}.\mathit{indx} = j.y.\mathit{indx}$ holds. Combining these assertions, we have $\mathit{Slot}[j.y.\mathit{indx}] = \mathit{false} \wedge \mathit{Slot}[j.y.\mathit{indx}] = \mathit{true}$, which is a contradiction. Hence, statement $34.j$ cannot falsify (I14). □

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

Proof: The only statements that may falsify the consequent are $2.i$, $31.i$, $2.j$, and $31.j$. 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 $\mathit{Reset} = i.y$ holds, which in turn implies that $F(i)$ is true. So, assume that $X = i \wedge \mathit{Reset} = i.y \wedge F(i)$ 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 $\mathit{Reset} \neq j.y$. Because $\mathit{Reset} = i.y$, this implies that $i.y \neq j.y$. In addition, by (I20), we have $i.y.\mathit{free} = \mathit{true} \wedge j.y.\mathit{free} = \mathit{true}$. 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 $i.y.\mathit{indx} = j.y.\mathit{indx}$ holds before $5.i$ is executed. Then, because $\mathit{Reset} = i.y$ holds, we have $j.y.\mathit{indx} = \mathit{Reset}.\mathit{indx}$. By (I9), this implies that $\mathit{Reset}.\mathit{free} = \mathit{false}$. Because $\mathit{Reset} = i.y$ holds, this implies that $i.y.\mathit{free} = \mathit{false}$ holds. However, by (I20), we have $i.y.\mathit{free} = \mathit{true}$, 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. □

The following invariants are straightforward and are stated without proof.

invariant $i@\{32, 33\} \Rightarrow \mathit{Reset} = i.y$ (I16)
invariant $i@\{15, 16, 34..36\} \Rightarrow \mathit{Reset} = (\mathit{false}, i.y.\mathit{indx})$ (I17)
invariant $i@\{30..37\} \Rightarrow Y = (\mathit{false}, 0)$ (I18)

invariant $\mathit{Y.free} = \mathit{true} \;\Rightarrow\; Y = \mathit{Reset}$ (I19)
invariant $i@\{3..20\} \;\Rightarrow\; \mathit{i.y.free} = \mathit{true}$ (I20)
invariant $(i@\{5..13, 26..32\}) = (\mathit{Proc}[i] = \mathit{true})$ (I21)

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

3.2 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 present three other invariants; two of these are quite straightforward and are stated without proof.

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

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

Proof: The only statements that can establish the antecedent are $15.i$, $17.i$, $34.i$, $35.i$, and $37.i$. Both $17.i$ and $37.i$ establish the consequent.

Statements $15.i$ and $35.i$ can establish the antecedent only if $\mathit{Proc}[k] = \mathit{true}$, where $k = \mathit{i.y.indx}$. By (I21), $\mathit{Proc}[k] = \mathit{true}$ 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 $\mathit{Slot}[\mathit{i.y.indx}] = \mathit{true}$. By (I23), this implies that $(\exists j :: j@\{8..18\} \wedge \mathit{i.y.indx} = \mathit{j.y.indx})$ holds. By (I22), $j@\{12..18\} \wedge i@\{34\}$ is false. It follows that $(\exists j :: j@\{8..11\} \wedge \mathit{i.y.indx} = \mathit{j.y.indx})$ holds, which implies that the antecedent is false.

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

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

invariant **(Fast path is open in the absence of contention)**
$(\forall i :: i@\{0\}) \;\Rightarrow\; \mathit{Y.free} = \mathit{true} \wedge \mathit{Infast} = \mathit{false} \wedge Y = \mathit{Reset}$ (I26)

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

4 Concluding Remarks

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 Fig. 2 implemented using Yang and Anderson's two-process algorithm, our fast-path algorithm can be simplified slightly. In particular, the writes to the variable *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 $C[0]$ — see [8]) used in Yang and Anderson's algorithm.

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 [2]. 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 [7]. Indeed, thinking about connections to renaming led us to discover our fast-path algorithm. 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.

References

1. T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Sys.*, 1(1):6–16, 1990.
2. R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1995.
3. E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
4. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.
5. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Computer Sys.*, 5(1):1–11, 1987.
6. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. on Computer Sys.*, 9(1):21–65, 1991.
7. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, 1995.
8. 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. 1993.

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