A Generic Local-Spin Fetch-and-ϕ-based Mutual Exclusion Algorithm*

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
Email: anderson@cs.unc.edu

Yong-Jik Kim
Tmax Soft Research Center
272-6 Seohyeon-dong, Seongnam-si
Gyeonggi-do, Korea 463-824
Email: jick@tmax.co.kr

November 2005, Revised November 2006

Abstract

We present a generic *fetch-and-ϕ*-based local-spin mutual exclusion algorithm, with $O(1)$ time complexity under the remote-memory-references time measure. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$, where N is the number of processes. The *rank* of a *fetch-and-ϕ* primitive is a notion introduced herein; informally, it expresses the extent to which processes may "order themselves" using that primitive. This algorithm breaks new ground because it shows that $O(1)$ time complexity is possible using a wide range of primitives. In addition, by applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\max(1, \log_r N))$ algorithm using any primitive of rank r, where $2 \leq r < N$.

Keywords: Fetch-and-ϕ primitives, local spinning, shared-memory mutual exclusion, theory of concurrent algorithms, time complexity.

*Work supported by NSF grants CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289. This work was presented in preliminary form at the 23rd IEEE International Conference on Distributed Computing Systems [3].

1 Introduction

Recent work on shared-memory mutual exclusion has focused on the design of algorithms that minimize interconnection-network contention through the use of *local spinning*. In local-spin algorithms, all busy waiting is by means of read-only loops in which one or more "spin variables" are repeatedly tested. Such spin variables must be either locally cacheable or stored in a local memory module that can be accessed without an interconnection network traversal. The former is possible on cache-coherent (CC) machines, while the latter is possible on distributed shared-memory (DSM) machines. As explained later, it is generally more difficult to design local-spin algorithms for DSM machines than for CC machines.

In this paper, several results concerning the time complexity of local-spin mutual exclusion algorithms are presented. The notion of time complexity assumed is that given by the *remote-memory-references* (*RMR*) *measure* [4]. Under this measure, an algorithm's time complexity is defined as the total number of remote memory references (*i.e.*, references that require an interconnection network traversal) required in the worst case by one process to enter and then exit its critical section once. An algorithm may have different RMR time complexities on CC and DSM machines, because on CC machines, variable locality is dynamically determined, while on DSM machines, it is statically determined.

The main focus of this paper is mutual exclusion algorithms implemented using *fetch-and-*ϕ primitives. A *fetch-and-*ϕ primitive is characterized by a particular function ϕ (which we assume to be deterministic), accesses a single variable atomically, and has the effect of the following pseudo-code, where *var* is the variable accessed.

```
fetch-and-φ(var, input)
    old := var;
    var := φ(old, input);
    return(old)
```

In this paper, we distinguish between *fetch-and-*ϕ primitives that are comparison primitives and those that are not. A *comparison primitive* conditionally updates a shared variable after first testing that its value meets some condition; examples include *compare-and-swap* and *test-and-set*.[1] Non-comparison primitives update variables unconditionally; examples include *fetch-and-increment* and *fetch-and-store*.

In recent work [2], we established a time-complexity lower bound of $\Omega(\log N / \log \log N)$ remote memory references for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives. In contrast, several constant-time algorithms are known that are based on noncomparison *fetch-and-*ϕ primitives [5, 7, 12]. This suggests that noncomparison primitives may be the best choice to provide in hardware, if one is interested in implementing efficient blocking synchronization mechanisms.

Constant-time local-spin mutual algorithms that use noncomparison primitives have been proposed by T. Anderson [5], Graunke and Thakkar [7], Mellor-Crummey and Scott [12], Craig [6], and Landin and Hagersten [11]. In each of these algorithms, blocked processes wait within a "spin queue." A process enqueues itself by using a *fetch-and-*ϕ primitive to update a shared "tail" pointer; a process's predecessor (if any) in the queue is indicated by the primitive's return value. A process in the spin queue waits (if necessary) until released by its predecessor. Although these algorithms follow a common strategy, they vary in the primitives used and the progress properties ensured. Some important attributes of each algorithm are listed below.

- T. Anderson's algorithm uses *fetch-and-increment* and requires an underlying cache-coherence mechanism for spins to be local. Thus, it has $O(1)$ RMR time complexity only on CC machines.
- Graunke and Thakkar's algorithm uses *fetch-and-store*. This algorithm also requires an underlying cache-coherence mechanism and thus has $O(1)$ RMR time complexity only on CC machines.
- Craig [6] and Landin and Hagersten [11] independently proposed the same algorithm, which is based on *fetch-and-store*. While Landin and Hagersten considered only CC machines, Craig presented constant-time variants of the algorithm for both CC and DSM machines.

[1] *compare-and-swap* and *test-and-set* are ordinarily defined to return a boolean condition indicating if the comparison succeeded. In this paper, we instead assume that each returns the accessed variable's original value, as in [8]. It is straightforward to modify any algorithm that uses the boolean versions of these primitives to instead use the versions considered in this paper.

- Mellor-Crummey and Scott actually presented two variants of their algorithm, one that uses *fetch-and-store*, and a second that uses both *fetch-and-store* and *compare-and-swap*. In both, spins are local on both CC and DSM machines. However, the *fetch-and-store* variant is not starvation-free, and hence actually has unbounded RMR time complexity. The variant that also uses *compare-and-swap is* starvation-free and has $O(1)$ RMR time complexity on both CC and DSM machines.

The existence of these algorithms gives rise to a number of intriguing questions regarding mutual exclusion algorithms. Can $O(1)$ mutual exclusion algorithms be devised for both CC and DSM machines using primitives other than *fetch-and-increment* and *fetch-and-store*? Is it possible to automatically transform a local-spin algorithm for CC machines so that it has the same RMR time complexity on DSM machines? Given that the $\Omega(\log N/ \log\log N)$ lower bound mentioned above applies to algorithms that use comparison primitives, we know that there exist *fetch-and-ϕ* primitives that are not sufficient for constructing $O(1)$ algorithms. For such primitives, what is the most efficient algorithm that can be devised? Can we devise a *ranking* of synchronization primitives that indicates the singular characteristic of a primitive that enables a certain RMR time complexity (for mutual exclusion) to be achieved? Such a ranking would provide information relevant to the implementation of *blocking* synchronization mechanisms that is similar to that provided by Herlihy's wait-free hierarchy [8], which is relevant to *nonblocking* mechanisms.[2]

Contributions of this paper. While we are not yet able to fully answer all of these questions, we do take some initial steps towards their resolution in this paper. We begin by proposing a ranking of *fetch-and-ϕ* primitives. Informally, a primitive of rank r has sufficient symmetry-breaking power to linearly order up to r invocations of that primitive. Based on this notion of a rank, we then present a generic N-process *fetch-and-ϕ*-based local-spin mutual exclusion algorithm that has $O(1)$ RMR time complexity on both CC and DSM machines. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$. Our generic algorithm breaks new ground because it shows that $O(1)$ RMR time complexity is possible using a wide range of primitives, on both CC and DSM machines.

We present our generic algorithm by first giving a variant that is designed for CC machines. We then present a transformation, which may be of more general interest, that can be used to replace spin loops that are local on CC machines with ones that are local on DSM machines. This transformation is then applied to our algorithm. This transformation makes use of an underlying two-process mutual exclusion algorithm and is correct as long as the added two-process algorithms are invoked safely (*i.e.*, by at most two processes at any time). We show that our DSM algorithm is correct by showing that, within it, these invocations are safe.

Both the CC and DSM variants of our basic algorithm have $\Theta(N)$ space complexity (for N processes), which is the same as the prior constant-time algorithms cited above. In the DSM case, $\Theta(N)$ is asymptotically optimal because each process needs a dedicated spin variable, as explained later. The term *space complexity* refers to the number of variables used. In addition, variable sizes are of importance. In our algorithms, some of the variables that are employed have sizes that are dependent on the generic *fetch-and-ϕ* primitive that is used; all other variables require $O(\log N)$ bits.

By applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\max(1, \log_r N))$ algorithm using any primitive of rank r, where $2 \leq r < N$. For the case $r = \Theta(N)$, this algorithm is clearly asymptotically time-optimal. However, as shown in the second author's Ph.D. dissertation [9], there exists a class of primitives with constant rank for which $\Theta(\max(1, \log_r N))$ is *not* optimal. This is shown by presenting a $\Theta(\log N/ \log\log N)$ algorithm that can be implemented using any primitive that meets an additional condition. This algorithm is quite complicated, and therefore is not presented in this paper, due to space constraints. This additional condition arises due to the need to *reset* a variable that is repeatedly updated by *fetch-and-ϕ* primitive invocations. In the generic algorithm presented in this paper, variables are reset using simple writes. In the $\Theta(\log N/ \log\log N)$ algorithm presented in [9], the *fetch-and-ϕ* primitive used must be of rank at least three and also be *self-resettable*, which means that the primitive itself can be used to reset a variable that has

[2]Herlihy's hierarchy is concerned with *computability*: a primitive (or object) X is stronger than a primitive (or object) Y if X can be used to implement Y (in a non-blocking manner) but not vice versa. The ranking suggested here is not concerned with computability, but rather time complexity. Nonetheless, both rankings provide information concerning the usefulness of primitives. Herlihy's hierarchy indicates which primitives should be supported in hardware if one is interested in implementing nonblocking algorithms; the proposed ranking indicates which primitives should be supported in hardware if one is interested in implementing scalable spin locks.

been updated using that primitive, *i.e.*, it is not necessary to perform resets using simple write operations. In the $\Theta(\log N/\log\log N)$ algorithm, this self-resettable feature is used (in addition to sometimes resetting with simple writes), with a resulting asymptotic improvement in time complexity for primitives of rank $o(\log N)$. As explained in [9], it follows from the $\Omega(\log N/\log\log N)$ lower bound mentioned above that this algorithm is time-optimal for certain self-resettable primitives of constant rank.

Organization. The rest of this paper is organized as follows. In Section 2, we present needed definitions. Then, in Section 3, we present our generic algorithm. A formal proof of correctness for two versions of the generic algorithm is presented in Appendices A and B. We end the paper with concluding remarks in Section 4.

2 Definitions

In the mutual exclusion problem, each process cycles through four code sections, termed "noncritical," "entry," "critical," and "exit" sections, respectively. A process may halt within its noncritical section but not within its critical section. Furthermore, no variables (other than program counters) accessed within a process's entry or exit section may be accessed within its critical or noncritical section. The objective is to design the entry and exit sections so that the following requirements hold.[3]

- **Exclusion:** At most one process executes its critical section at any time.
- **Starvation-Freedom:** If some process is in its entry (exit) section, then that process eventually executes its critical (noncritical) section.

We hereafter let N denote the number of processes in the system, and assume that each process has a unique process identifier in the range $0, \ldots, N-1$.

We assume the existence of a *generic fetch-and-ϕ* primitive, as defined in Section 1. We will use "*Vartype*" to denote the type of the accessed variable *var*. (The accessed variable's type is part of the definition of such a primitive.) For example, for a *fetch-and-increment* primitive, *Vartype* would be **integer**, and for a *test-and-set* primitive, it would be **boolean**. In our algorithms, we use $\perp$ to denote the initial value of a variable accessed by a *fetch-and-ϕ* primitive (*e.g.*, if *Vartype* is **boolean**, then $\perp$ would denote either *true* or *false*). We now define the notion of a "rank," mentioned earlier.

Definition: The *rank* of a *fetch-and-ϕ* primitive is the largest integer r satisfying the following.

> For each process p, there exists a constant array $\alpha_p[0..\infty]$ of input values, such that, for any integer value a_p, if p performs the sequence of *fetch-and-ϕ* invocations given by
>
> $$\textbf{for } i := a_p \textbf{ to } \infty \textbf{ do } \textit{fetch-and-}\phi(v, \alpha_p[i]) \textbf{ od}$$
>
> on a variable v (of type *Vartype*) that is initially $\perp$ (for some choice of $\perp$), then in any interleaving of these invocations by the N different processes, **(i)** any two invocations among the first $r-1$ by different processes write different values to v, **(ii)** any two *successive* invocations among the first $r-1$ by the same process write different values to v, and **(iii)** of the first r invocations, only the first invocation returns $\perp$.

A *fetch-and-ϕ* primitive has *infinite rank* if the condition above is satisfied for arbitrarily large values of r. □

As our generic algorithm shows, a *fetch-and-ϕ* primitive with rank r has enough power to linearly order r invocations by possibly different processes unambiguously. Note that it is not necessary for the primitive to fully order invocations by the same process, since each process can keep its own execution history.

[3] Progress properties other than starvation-freedom may be of interest as well.

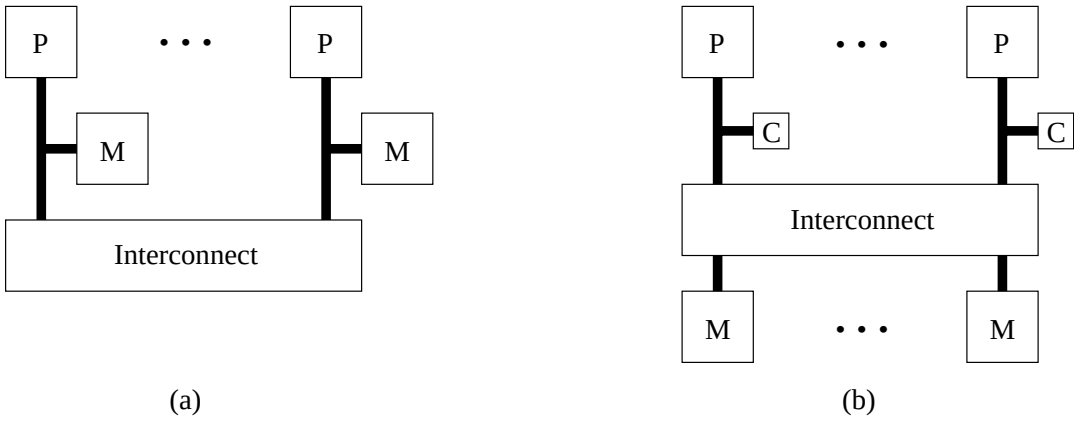

Figure 1: **(a)** DSM model. **(b)** CC model. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

Examples. An r-bounded *fetch-and-increment* primitive on a variable v with range $0, \ldots, r-1$ is defined by $\phi(old, input) = \min(r-1, old+1)$. (In this primitive, the input parameter is not used, and hence we may simply assume $\alpha_p[j] = \perp$ for all p and j.) If v is initially 0, then any r consecutive invocations on v return distinct values, $0, 1, \ldots, r-1$. Moreover, any further invocation (after the r^{th}) returns $r-1$, which is the same as the return value of the r^{th} invocation. Therefore, an r-bounded *fetch-and-increment* primitive has rank r, and an unbounded *fetch-and-increment* primitive has infinite rank.

For *fetch-and-increment* primitives, the input parameter α is extraneous. However, this is not the case for other primitives. Consider a *fetch-and-store* primitive on a variable with $2N+1$ distinct values ($2N$ pairs $(p, 0)$ and $(p, 1)$, where p is a process, and an additional initial value $\perp$). By defining $\alpha_p[j] = (p, j \bmod 2)$, it is easily shown that *fetch-and-store* has infinite rank. (Informally, each process may write the information "this is an (even/odd)-indexed invocation by process p" each time.) It also follows that an unbounded *fetch-and-store* primitive has infinite rank.

Finally, *test-and-set* has rank two: only the first *test-and-set* invocation on a variable initially *false* returns its initial value. *compare-and-swap* also has rank two.

3 A Constant-Time Generic Algorithm

In this section, we present an $O(1)$ mutual exclusion algorithm that uses a generic *fetch-and-ϕ* primitive, which is assumed to have rank at least $2N$. Two variants of the algorithm are presented, one for CC machines and one for DSM machines. These two architectural paradigms have been considered extensively in work on local-spin algorithms. Both are illustrated in Figure 1. In a DSM machine, each processor has its own memory module that can be accessed without accessing the global interconnection network. On such a machine, a shared variable can be made locally accessible by storing it in a local memory module. In a CC machine, each processor has a private cache, and some hardware protocol is used to enforce cache consistency (*i.e.*, to ensure that all copies of the same variable in different local caches are consistent). On such a machine, a shared variable becomes locally accessible by migrating to a local cache line. In this paper, we consider a DSM machine with caches that are kept coherent to be a CC machine.[4] We also assume that there is a unique process executing the algorithm on each processor and that these processes do not migrate.

In local-spin algorithms for DSM machines, each process must have its own dedicated spin variables (which must be stored in its local memory module). In contrast, in algorithms for CC machines, processes may *share* spin variables, because each process can read a different cached copy. Because of this flexibility, algorithms

[4] Although virtually every modern multiprocessor is cache-coherent, non-cache-coherent DSM systems are still used in embedded applications, where cheaper computing technology often must be used due to cost limitations. Thus, the DSM model is of relevance for reasons other than historical interest.

for CC machines tend to be a bit simpler than those for DSM machines. This is why we present separate algorithms. Our CC algorithm, denoted G-CC, is presented first, and then its DSM counterpart, denoted G-DSM, is obtained by means of a fairly simple transformation. The two algorithms are shown in Figures 2 and 5. In both algorithms, "**await** B," where B is a boolean expression, is used as a shorthand for the busy-waiting loop "**while** $\neg B$ **do** /* null */ **od**."

As noted earlier, we assess time complexity using the *RMR* (*remote-memory-references*) *time complexity measure*. As its name suggests, only remote memory references that cause an interconnect traversal are counted under this measure. We will assess the RMR time complexity of an algorithm by counting the total number of remote memory references required by one process to enter and then exit its critical section once. An algorithm may have different RMR time complexities under the CC and DSM models because the notion of a remote memory reference differs under these two models. In the CC model, we assume that, once a spin variable has been cached, it remains cached until it is either updated or invalidated as a result of being modified by another process on a different processor. (Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable because of associativity or capacity limitations.)

3.1 Algorithm G-CC: A Generic Algorithm for CC Machines

In this section, we present our generic algorithm for CC machines. Our intent here is to explain the various mechanisms used in the algorithm in an intuitive way. In so doing, we state and explain several formal properties of the algorithm. However, these explanations cannot be taken as a formal proof of correctness. For that, we refer the reader to Appendix A. At the end of this section, we illustrate some of the algorithm's properties by considering a fairly comprehensive example execution.

When trying to implement a mutual exclusion algorithm using a generic *fetch-and-ϕ* primitive — of which only its rank r is known — the primary problem that arises is the following.

> If the primitive is invoked more than r times to access a variable, then it may not be able to provide enough information for processes to order themselves. Therefore, *the algorithm must provide a means of resetting such a variable before it is accessed r times*.

Because we are using a primitive of rank $2N$ in Algorithm G-CC, we need to reset a variable accessed by the primitive before it is accessed $2N$ times. We do this by using two "waiting queues," indexed 0 and 1. Associated with each queue j is a "tail pointer," $Tail[j]$. In its entry section, a process enqueues itself onto one of these two queues by using the *fetch-and-ϕ* primitive to update its tail pointer, and waits on its predecessor, if necessary. At any time, one of the queues is designated as the "current" queue, which is indicated by the shared variable *CurrentQueue*. The other queue is called the "old" queue. The algorithm switches between the two queues over time in a way that ensures that each tail pointer is reset before being accessed $2N$ times. We now describe the reset mechanism in detail.

When a process p begins its entry section, it determines which queue is the current queue by reading the variable *CurrentQueue* (statement 3 of Figure 2), and then enqueues itself onto that queue using the *fetch-and-ϕ* primitive (statement 5). If p is not at the head of its queue ($p.prev \neq \perp$),[5] then it waits until its predecessor in the queue updates the spin variable $Signal[p.idx][p.prev]$ (statement 6), which p then resets (statement 7). Note that the local variable $p.prev$ (if not $\perp$) indicates a spin variable written by p's *predecessor* in the queue. The spin variable that a potential successor to p may wait on is indicated, from p's perspective, by the local variable $p.self$.

As explained below, it is possible for a process q to read *CurrentQueue* before another process updates *CurrentQueue* to switch to the other queue. Such a process q will then enqueue itself onto the old queue. Thus, *both* queues may possibly hold waiting processes. To arbitrate between processes in the two queues, an extra two-process mutual exclusion algorithm is used. A process competes in this two-process algorithm (after reaching the head of its waiting queue) using the routines $\texttt{Entry}_2$ and $\texttt{Exit}_2$, with the index of its queue as a "process identifier" (statements 8 and 12), as illustrated in Figure 3(a). Note that this extra two-process algorithm can be implemented from reads and writes in $O(1)$ time [13].

[5] We use $s.p$ to denote the statement with label s of process p, and $p.v$ to represent p's private variable v.

```
shared variables
    CurrentQueue: 0..1;
    Tail: array[0..1] of Vartype initially ⊥;
    Position: array[0..1] of 0..2N − 1 initially 0;
    Signal: array[0..1][Vartype] of boolean initially false;
    Active: array[0..N − 1] of boolean initially false;
    QueueIdx: array[0..N − 1] of (⊥, 0..1)

private variables
    idx: 0..1;
    counter: integer;
    prev, self, tail: Vartype;
    pos: 0..2N − 1

process p ::    /* 0 ≤ p < N */

while true do
0:  Noncritical Section;

1:  QueueIdx[p] := ⊥;
2:  Active[p] := true;
3:  idx := CurrentQueue;
4:  QueueIdx[p] := idx;
5:  prev := fetch-and-φ(Tail[idx], α_p[counter]);
    self := φ(prev, α_p[counter]);
    counter := counter + 1;
    if prev ≠ ⊥ then
6:      await Signal[idx][prev];
7:      Signal[idx][prev] := false
    fi;
8:  Entry_2(idx);

9:  Critical Section;

10: pos := Position[idx];
11: Position[idx] := pos + 1;
12: Exit_2(idx);

13: if (pos < N) ∧ (pos ≠ p) then
14:     await ¬Active[pos] ∨
15:           (QueueIdx[pos] = idx)
    elseif pos = N then
16:     tail := Tail[1 − idx];
17:     Signal[1 − idx][tail] := false;
18:     Tail[1 − idx] := ⊥;
19:     Position[1 − idx] := 0;
20:     CurrentQueue := 1 − idx
    fi;

21: Signal[idx][self] := true;
22: Active[p] := false
od
```

Figure 2: Algorithm G-CC: Generic *fetch-and-φ*-based mutual exclusion algorithm for CC machines.

As explained above, some process must reset the current queue before it is accessed $2N$ times. To facilitate this, each queue j has an associated shared variable $Position[j]$. This variable indicates the relative position of the current head of the queue, starting from 0. For example, in Figure 3(a), the head of queue 0 is at position 2, and hence $Position[0]$ equals 2. A process in queue j updates $Position[j]$ while still effectively in its critical section (statements 10 and 11). Thus, $Position[j]$ cannot be concurrently updated by different processes.

A process exchanges the role of the two queues in its exit section if it is at position N in the current queue (statements 16–20). (These statements will be explained in detail shortly.) Figures 3(b) and 3(c) show the state of the two queues before and after such an exchange. In order to exchange the queues, we must ensure the following property.

Property 1 If a process executes its critical section after having acquired position N of the current queue, then no process is in the old queue.

(A process is considered to be "in" the old queue if it read the index of that queue from *CurrentQueue*. In particular, that process may not yet have updated the queue's tail pointer.) Given Property 1, a process at position N may safely reset the old queue and exchange the queues. (More formally, the fact that such an exchange is safe follows from invariant (I13) in Appendix A.) Property 1 is a direct consequence of the following property.

Property 2 If a process executes its critical section after having acquired position *pos* of the current queue, and if $pos > q$, then, by the time process p reaches statement 21, process q is not in the old queue.

To maintain Property 2, each process p has two associated variables, $Active[p]$ and $QueueIdx[p]$, which indicate (respectively) whether process p is active, and if so, which queue it is executing in (statements 1, 2, 4, and 22). If a process p executes at position $q < N$ in the current queue, then in its exit section, p waits

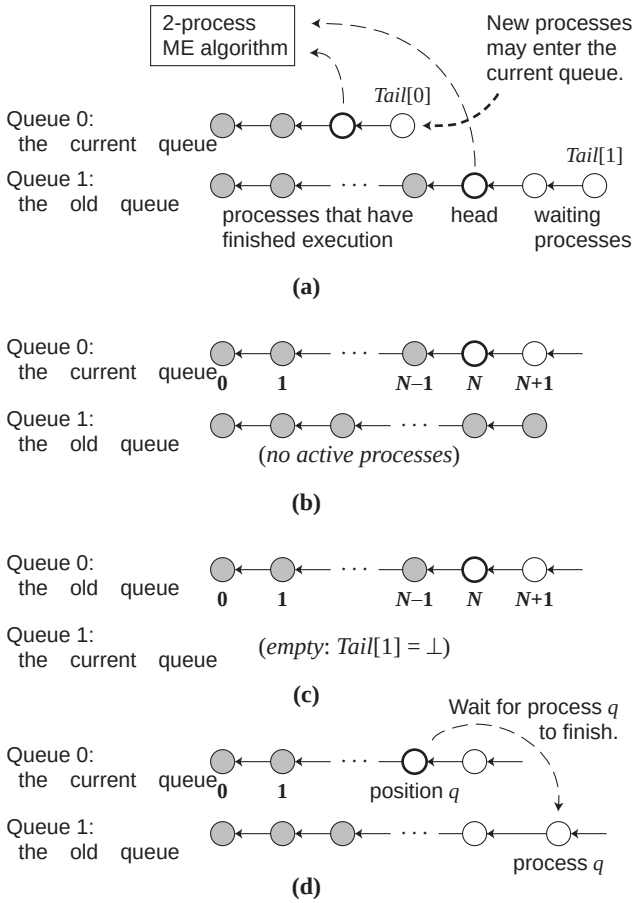

Figure 3: The structure of Algorithm G-CC. **(a)** The overall structure. This figure shows a possible state of execution when the current queue is queue 0. (The "finished" processes may be duplicated, because a process may execute its critical section multiple times.) **(b)** A state just before *CurrentQueue* is updated. **(c)** A state just after *CurrentQueue* is updated. **(d)** A process (in its exit section) in the current queue waiting for another in the old queue.

until either q finishes its exit-section execution (*i.e.*, $Active[p] = false$; statement 14) or enters the current queue (statement 15). p thus ensures that process q does not execute in the old queue, and then signals a possible successor (*i.e.*, a process at position $q + 1$ in the current queue) that it is now at the head of the current queue (statement 21). This situation is depicted in Figure 3(d).

Although p waits for q, starvation-freedom is guaranteed, because q is in the old queue, and hence makes progress independently of the current queue. Only the current queue is stalled until q finishes execution. (The fact that p may have to wait for a significant duration in its exit section may be a cause for concern. However, such waiting can be eliminated, if process p instructs process q to signal p's successor *after* q finishes its critical section. Thus, p may finish execution without waiting for q. For simplicity, this handshake has not been added to Algorithm G-CC.)

We now explain statements 16–20, which are executed in order to exchange the role of the two queues. Without loss of generality, suppose that a process p executes these statements with $p.idx = 0$. (See Figure 3(b) and Figure 3(c).) Variables $Tail[1]$ and $Position[1]$ are initialized by statements 18 and 19, respectively. In addition, we must ensure that each entry of the $Signal[1][\ldots]$ array is reset to *false*. Note that, if a process q in queue 1 establishes $Signal[1][x] = true$ (where $x = q.self$) by executing statement 21, then its successor r (which spins on $Signal[1][x]$ at statement 6) resets $Signal[1][x]$ by executing statement 7. Thus, by the time p executes statements 16–20, Property 1 ensures that every entry of $Signal[1][\ldots]$ is reset to *false*, *except for the one set by the last process in queue 1.* (Clearly, the last process does not have a successor, so this entry is not

reset.)

Property 1 again ensures that this last entry of queue 1 is indicated by *Tail*[1]. Therefore, statements 16 and 17 properly reset this entry and thereby complete the reinitialization of *Signal*[1][. . .]. Finally, statement 20 exchanges the two queues.

We still must show that using a *fetch-and-*ϕ primitive of rank $2N$ is sufficient. Suppose that process p acquires position N of queue 0 when it is the current queue. We claim that at most $N-1$ processes may be enqueued onto queue 0 after p and before the queues are exchanged again. For a process q to enqueue itself onto queue 0 after p, it must have read the value of *CurrentQueue* before it was updated by p. For q to enqueue itself a *second* time onto queue 0, it must read $CurrentQueue = 0$ again, *after* $CurrentQueue = 1$ was established by p. This implies that the two queues have been exchanged again. (We remind the reader that, by the explanation above, the queues will not be exchanged again until there are no processes in queue 0.) Thus, after p establishes that queue 1 is current, and while queue 0 continues to be the old queue, at most $N-1$ processes may be enqueued (after p) onto queue 0. Thus, we have the following property.

Property 3 Each process's position is between 0 and $2N-1$ (inclusive).

(Formally, this property follows from invariants (I15) and (I22) in Appendix A.) It follows that a rank of $2N$ is sufficient.

Example: To better illustrate the various mechanisms used in the algorithm, we will construct an example execution, which is illustrated in Figure 4. In this example, each *Tail* variable is assumed to range over $0, \ldots, 2N-1$, with its initial value defined to be 0. (That is, the type *Vartype* is taken to be $0..2N-1$, and in the declaration of *Tail* and in statements 5 and 18, the symbol $\perp$ is taken to be 0.) Furthermore, we assume that the *fetch-and-*ϕ primitive being used is a $2N$-bounded *fetch-and-increment* primitive.

The execution is constructed as follows. Starting from the initial state, processes 0 and 1 alternately enter and exit their critical sections while $CurrentQueue = 0$ continues to hold until process 0 obtains $0.prev = N$ (statement 5) and establishes $0.pos = N$ (statement 10). Assume at this point in the execution that process 0 is about to execute statement 13 and all other processes are in their noncritical sections.

Before continuing, note that, in the execution described so far, had one of processes 0 and 1 waited on the other at statement 6, then when its waiting ceased, it would have immediately reset the correspond *Signal* variable to *false* in statement 7. Thus, at this point in the execution, all *Signal* variables in queue 0 are *false*. The situation so far is illustrated in Figure 4(a).

To continue, suppose that process 1 executes its entry section until it busy waits at statement 6, then process 2 does the same, followed in the same way (in order) by processes 3 through $N-1$. Then, prior to waiting at statement 6, each process k, where $1 \leq k < N$, obtains $k.prev = N + k$. That is, the spin variable that process k's predecessor in queue 0 updates is at position $N + k$. (Note that, for our particular choice of *fetch-and-*ϕ primitive, we have $p.pos = p.prev$ for each process p, but in general these two values may have no relation with each other.)

Now, suppose that process 0 executes statements 16–20 to switch the queues. This establishes the following:

- $0.tail = 0$ (statement 16; recall that $\perp = 0$ is the initial value of *Tail*[1]);
- *Signal*[1][0] = *false* (statement 17; this actually has no impact because all *Signal* variables are initially *false*);
- *Tail*[1] = 0 (statement 18; again, $\perp$ is defined to be 0);
- *Position*[1] = 0 (statement 19); and
- *CurrentQueue* = 1 (statement 20).

Next, suppose that process 0 establishes *Signal*[0][$N+1$] = *true* (statement 21; note that $0.prev = N$ and $0.self = N+1$) and *Active*[0] = *false* (statement 22) and returns to its noncritical section. Before continuing with the execution, note that, at this point, because *Signal*[0][$N+1$] = *true* holds, process 1 is enabled to transit from statement 6 to statement 7. When it later executes statement 7, it will re-establish *Signal*[0][$N+1$] = *false*.

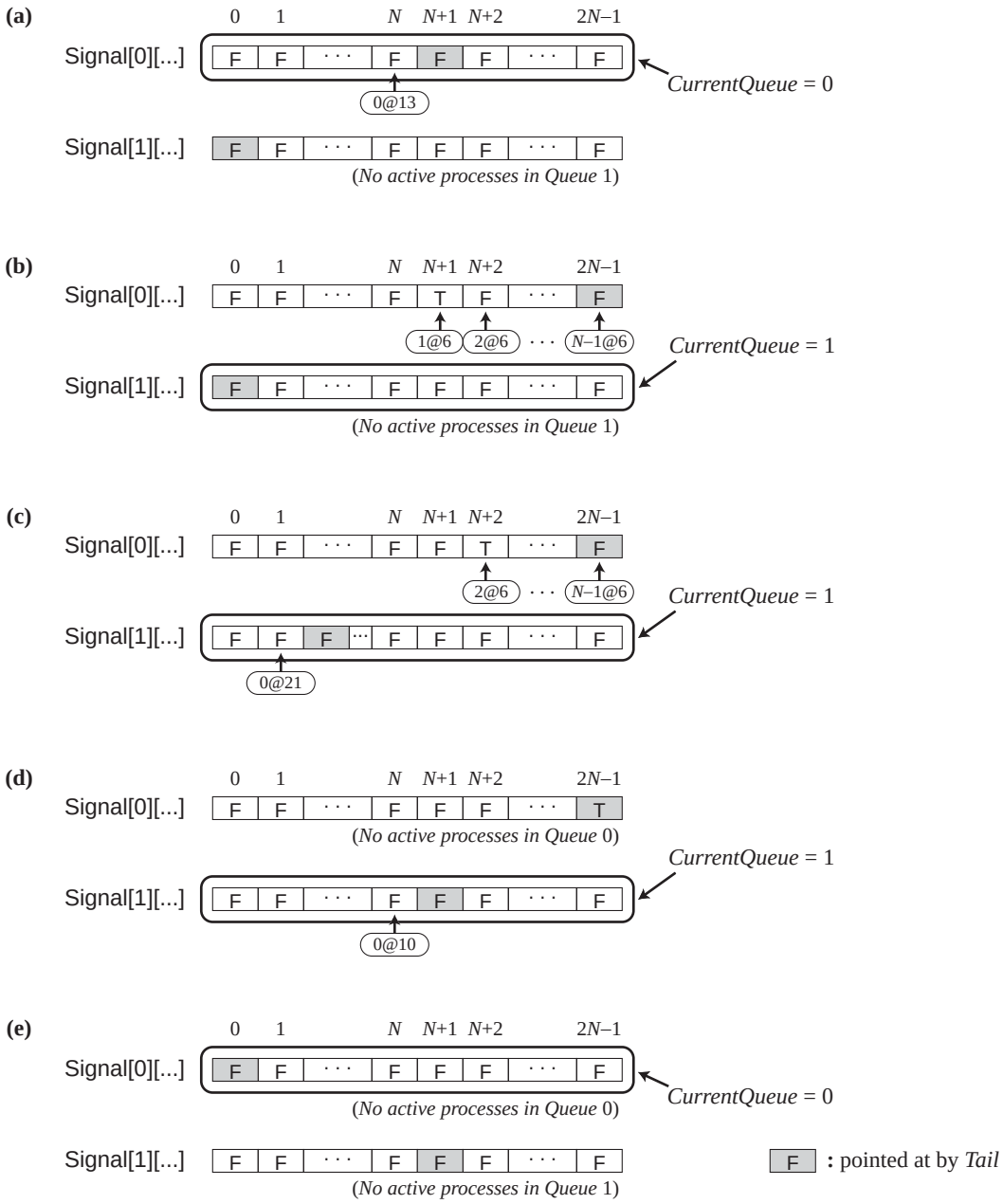

Figure 4: Example execution of of ALGORITHM G-CC. The notation $p@s$ means that process p is enabled to execute statement s. In each inset, the value of each active process p's *prev* variable is indicated; this value indicates a spin variable that p's predecessor updates and that p may potentially block on. For example, in inset (a), $0.prev = N$.

In a similar way, when processes 2 through $N-1$ later execute statement 7, each of $Signal[0][N+2]$ through $Signal[0][2N-1]$ will be reset to *false*. However, process $N-1$ later will establish $Signal[0][2N-1] = true$ when it executes statement 21. (A $2N$-bounded *fetch-and-increment* operation on a variable with the value $2N-1$ leaves the variable's value unchanged, so $(N-1).prev = 2N-1$ and $(N-1).self = 2N-1$.) We must make sure that, as the execution progresses, $Signal[0][2N-1] = false$ is re-established before queue 0 becomes the current queue again. Note that, at this point in the execution, $Tail[0] = 2N-1$ holds. (This fact will be used later.) The situation so far is illustrated in Figure 4(b).

To continue the execution, note that, to the point described so far, process 0 is in its noncritical section, and

each other process is in queue 0, with process 1 stalled at statement 6 and processes 2 through $N-1$ blocked there. Now, assume that process 0 enters its entry section once again. Then, it will read $\mathit{CurrentQueue} = 1$ (statement 3) and obtain $0.\mathit{prev} = 0$ (statement 5). Assuming that each of the remaining processes remains at statement 6 (so none acquires the two-process lock), process 0 may progress to statement 10, where it establishes $0.\mathit{pos} = 0$, and then to statement 13. Because $0.\mathit{pos} = 0$ holds, process 0 next transits to statement 21. After executing statements 21 and 22, suppose that process 0 returns to its noncritical section.

Now, suppose that process 0 repeats the above actions while the other processes remain at statement 6. Then, the scenario is similar, but process 0 now obtains $0.\mathit{prev} = 1$ (statement 5) and then establishes $0.\mathit{pos} = 1$ (statement 10). Thus, when process 0 reaches statement 13, it will transit to statement 14, where it will be blocked until process 1 is either no longer active or has accessed queue 1. Suppose that process 1 completes its entry, critical, and exit sections, and returns to its noncritical section, after which, process 0 (because it is no longer blocked) transits to statement 21, as illustrated in Figure 4(c). At this point in the execution, process 1 is no longer in queue 0 (see Property 2). Suppose now that process 0 completes its exit section and also returns to its noncritical section.

Suppose that we repeat this same scenario until process 0 establishes $0.\mathit{pos} = N$ by executing statement 10. Then, by this point in the execution, no processes can be in queue 0 (see Property 1). This is illustrated in Figure 4(d). Thus, when process 0 switches the queues again by subsequently executing statements 16–20, it is safe to switch the two queues. Further, note that, when process 0 performs statement 16, it will obtain $0.\mathit{tail} = \mathit{Tail}[0] = 2N-1$. (This is the most recent value assigned to $\mathit{Tail}[0]$ in the execution.) Thus, when it performs statement 17, it will establish $\mathit{Signal}[0][2N-1] = \mathit{false}$, which as noted earlier, must be done before queue 0 becomes the current queue again. The execution at this point is depicted in Figure 4(e). □

The busy-waiting loops at statements 6, 14, and 15 in Figure 2 are read-only loops in which variables are read that may be updated by a unique process. On a CC machine, each such loop incurs $O(1)$ RMR time complexity. It follows that Algorithm G-CC has $O(1)$ RMR time complexity on CC machines.

As noted earlier, a detailed correctness proof of Algorithm G-CC is given in Appendix A. From the discussion so far, we have the following lemma.

Lemma 1 *If the underlying fetch-and-ϕ primitive has rank at least $2N$, then* Algorithm G-CC *is a correct, starvation-free mutual exclusion algorithm with $O(1)$ RMR time complexity in CC machines.* □

3.2 Algorithm G-DSM: A Generic Algorithm for DSM Machines

We now explain how to convert Algorithm G-CC into Algorithm G-DSM, which is illustrated in Figure 5. The key idea of this conversion is a simple transformation of each busy-waiting loop, which we examine here in isolation. In Algorithm G-CC, all busy waiting is by means of statements of the form "**await** B," where B is some boolean condition. Moreover, if a process p is waiting for condition B to hold, then there is a unique process that can establish B, and once B is established, it remains true, until p's "**await** B" statement terminates.

In Algorithm G-DSM, each statement of the form "**await** B" has been replaced by the code fragment on the left below (see statements 10–17 and 25–33 in Figure 5), and each statement of the form "$B := \mathit{true}$" by the code fragment on the right (see statements 4–8, 40–44, and 45–49).

```
a: Entry_2(J, 0);
b:     flag := B;
c:     Waiter[J] := if flag then ⊥ else p;
d:     Spin[p] := false;
e: Exit_2(J, 0);
f: if ¬flag then
g:     await Spin[p];
h:     Waiter[J] := ⊥
   fi
```

```
i: Entry_2(J, 1);
j:     B := true;
k:     next := Waiter[J];
l: Exit_2(J, 1);
m: if next ≠ ⊥ then Spin[next] := true fi
```

The variable $\mathit{Waiter}[\mathcal{J}]$ is assumed to be initially $\perp$, and $\mathit{Spin}[p]$ is a spin variable used exclusively by process

```
/* all variable declarations are as defined in Figure 2 except as noted here */

shared variables
    Waiter1: array[0..N − 1] of (⊥, 0..N − 1);
    Waiter2: array[0..1][Vartype] of (⊥, 0..N − 1);
    Spin: array[0..N − 1] of boolean initially false

private variables
    next: (⊥, 0..N − 1);
    flag: boolean;
    q: 0..N − 1

process p ::    /* 0 ≤ p < N */

while true do
0:  Noncritical Section;

1:  QueueIdx[p] := ⊥;
2:  Active[p] := true;
3:  idx := CurrentQueue;
4:  Entry_2(p, 1);
5:      QueueIdx[p] := idx;
6:      q := Waiter1[p];
7:  Exit_2(p, 1);
8:  if q ≠ ⊥ then Spin[q] := true fi;

9:  prev := fetch-and-φ(Tail[idx], α_p[counter]);
    self := φ(prev, α_p[counter]);
    counter := counter + 1;
    if prev ≠ ⊥ then

10:     Entry_2((idx, prev), 0);
11:         flag := Signal[idx][prev];
12:         Waiter2[idx][prev] :=
                        if flag then ⊥ else p;
13:         Spin[p] := false;
14:     Exit_2((idx, prev), 0);
15:     if ¬flag then
16:         await Spin[p];
17:         Waiter2[idx][prev] := ⊥
        fi;

18:     Signal[idx][prev] := false
    fi;
19: Entry_2(idx)

20: Critical Section;

21: pos := Position[idx];
22: Position[idx] := pos + 1;
23: Exit_2(idx);

24: if (pos < N) ∧ (pos ≠ p) then
        q := pos;

25:     Entry_2(q, 0);
26:         flag :=¬Active[q] ∨
27:                 (QueueIdx[q] = idx);
28:         Waiter1[q] :=
                        if flag then ⊥ else p;
29:         Spin[p] := false;
30:     Exit_2(q, 0);
31:     if ¬flag then
32:         await Spin[p];
33:         Waiter1[q] := ⊥
        fi

    elseif pos = N then
34:     temp := Tail[1 − idx];
35:     Signal[1 − idx][temp] := false;
36:     Tail[1 − idx] := ⊥;
37:     Position[1 − idx] := 0;
38:     CurrentQueue := 1 − idx
    fi;

39: if pos < 2N − 1 then
40:     Entry_2((idx, self), 1);
41:         Signal[idx][self] := true;
42:         next := Waiter2[idx][self];
43:     Exit_2((idx, self), 1);
44:     if next ≠ ⊥ then Spin[next] := true fi
    fi;

45: Entry_2(p, 1);
46:     Active[p] := false;
47:     next := Waiter1[p];
48: Exit_2(p, 1);
49: if next ≠ ⊥ then Spin[next] := true fi
od
```

Figure 5: Algorithm G-DSM: Generic *fetch-and-φ*-based mutual exclusion algorithm for DSM machines. Statements different from Figure 2 are shown with **boldface** line numbers.

p (and, hence, it can be stored in memory local to p). $\texttt{Entry}_2$ and $\texttt{Exit}_2$ represent an instance of a two-process mutual exclusion algorithm, indexed by $\mathcal{J}$, *i.e.*, $\mathcal{J}$ is used to identify a particular instance of the two-process mutual exclusion algorithm. To see that this transformation is correct, assume that a process p executes lines a–h while another process q executes lines i–m. Since lines b–d and j–k execute within a critical section, lines b–d precede lines j–k, or vice versa. If b–d precede j–k, and if $B = \mathit{false}$ holds before the execution of b–d, then p assigns $\mathit{Waiter}[\mathcal{J}] := p$ at line c, and initializes its spin variable at line d. Process q subsequently reads $\mathit{Waiter}[\mathcal{J}] = p$ at line k, and establishes $\mathit{Spin}[p] = \mathit{true}$ at line m, which ensures that p is not blocked. On the other hand, if lines j–k precede lines b–d, then process q reads $\mathit{Waiter}[\mathcal{J}] = \perp$ (the initial value) at line k, and does not update any spin variable at line m. Since process p executes line b after q executes line j, p preserves $\mathit{Waiter}[\mathcal{J}] = \perp$, and does not execute lines g and h.

Note that for the above correctness argument to be valid, it is crucial that, at any time, at most one process executes within lines a–h and within lines i–m. Otherwise, the correctness of the underlying two-process mutual exclusion algorithm cannot be guaranteed. In order to satisfy this property, we have to apply a minor modification to Algorithm G-CC before applying the transformation shown above. Namely, we change statement 21 of Algorithm G-CC as follows. (See also statement 39 in Figure 5.)

```
21: if pos < 2N − 1 then
        Signal[idx][self] := true
    fi;
```

We hereafter denote this modified algorithm as Algorithm G-CC′. We now informally argue that this modification does not affect the algorithm's correctness. Consider a process p that executes statement 21 while at position $2N - 1$. Let $i = p.\mathit{idx}$ and $x = p.\mathit{self}$. By Property 3, p cannot possibly have a successor. Moreover, a process may read $\mathit{Signal}[i][x]$ (at statement 6) only if its predecessor has written (or will eventually write) $\mathit{Signal}[i][x] := \mathit{true}$ by executing statement 21. Hence, in Algorithm G-CC, the value of $\mathit{Signal}[i][x]$ written by p at statement 21 is never read by any process until the queues are exchanged again, at which point the process that performs the exchange overwrites $\mathit{Signal}[i][x]$ by executing statement 17.

From the preceding discussion, it follows that Algorithm G-CC′ is a correct, starvation-free mutual exclusion algorithm, to which we can then apply the transformation shown above and obtain Algorithm G-DSM. Moreover, all the invariants stated in Appendix A also remain valid for Algorithm G-CC′, except for those that directly refer to the *Signal* array. (We will explain shortly why we cannot apply the above transformation to Algorithm G-CC directly; see the reasoning for statements 40–43 below.) A formal version of the argument given here regarding Algorithm G-CC′ is given in Appendix B.

We now argue that the various two-process mutual exclusion algorithms added to Figure 5 are safe by considering each pair of `Entry`/`Exit` calls in this figure in turn. By *safe*, we mean that, if some process is executing within a code fragment that begins with $\texttt{Entry}_2(\mathcal{J}, b)$ and ends with $\texttt{Exit}_2(\mathcal{J}, b)$, where $\mathcal{J}$ identifies this particular two-process mutual exclusion instance and b is 0 or 1, then no other process may be concurrently executing within a code fragment that begins with $\texttt{Entry}_2(\mathcal{J}, b)$ and ends with $\texttt{Exit}_2(\mathcal{J}, b)$. (The arguments below are also presented more formally in Appendix B.)

Statements 4–7 and statements 45–48: The `Entry`/`Exit` calls here are clearly safe because of the usage of the identifier 'p', which uniquely corresponds to process p.

Statements 10–14: Because the *fetch-and-*ϕ primitive used in Algorithm G-CC′ has rank at least $2N$, by Property 3, each process that executes statements 5–7 of Algorithm G-CC′ within a particular queue must have a distinct value of *prev*. This implies that the `Entry`/`Exit` pair at statements 10–15 in Figure 5 is safe.

Statements 25–30: Since the queues are switched after a process at position N executes its critical section, if a process executes statements 14 and 15 of Algorithm G-CC′, then at that time, no process with the same value of *pos* may be executing within the other queue. Since the algorithm clearly ensures that each process executing within the same queue has a distinct value of *pos*, this implies that the `Entry`/`Exit` pair at statements 25–30 in Figure 5 is safe.

Statements 40–43: Because the *fetch-and-ϕ* primitive has rank at least $2N$, each process at queue i (for some i) whose position is between 0 and $2N-2$ (inclusive) writes a distinct value to $Tail[i]$ at statement 6 of ALGORITHM G-CC′, and hence, has a distinct value of *self*. This implies that the `Entry/Exit` pair at statements 40–43 in Figure 5 is safe.

Note that, if process p is at position $2N-1$ of queue i, then p is allowed to write any value to $Tail[i]$, by the definition of rank. (By Property 3, p cannot have a successor.) Hence, in ALGORITHM G-CC, p's execution of statement 21 may write the same element of *Signal* as some other process residing at an earlier position. In particular, in ALGORITHM G-CC, if some other process q has acquired position h ($< 2N-1$) of queue i, then both statements 21.p and 21.q may write the same element of *Signal*.

This does not pose a problem for ALGORITHM G-CC, because the execution of statement 21 is atomic, and hence p may execute statements 8–21 only after all of its predecessors have finished executing statement 21. However, this *would* be a problem if we were to implement statement 21 with statements i–m as in ALGORITHM G-DSM. This is why we introduce an additional "**if**" clause in ALGORITHM G-CC′.[6]

The above transformation can also be applied within other algorithms, as long as a safety proof like that above can be established. For example, this transformation can be applied to convert the algorithm of Graunke and Thakkar [7] to a variant that locally spins on DSM machines.[7] Given the correctness of the above transformation, we have the following.

Lemma 2 *If the underlying fetch-and-ϕ primitive has rank at least $2N$, then* ALGORITHM G-DSM *is a correct, starvation-free mutual exclusion algorithm with $O(1)$ RMR time complexity in DSM machines.* □

If we have a *fetch-and-ϕ* primitive with rank r ($4 \leq r < 2N$), then we can arrange instances of ALGORITHM G-DSM in an arbitration tree, where each process is statically assigned a leaf node and each non-leaf node consists of an $\lfloor r/2 \rfloor$-process mutual exclusion algorithm, implemented using ALGORITHM G-DSM. Because this arbitration tree is of $\Theta(\log_r N)$ height, we have the following theorem. (Note that for $r = 2$ or 3, a $\Theta(\log_r N)$ algorithm is possible without even using the *fetch-and-ϕ* primitive [13].)

Theorem 1 *Using any fetch-and-ϕ primitive of rank $r \geq 2$, starvation-free mutual exclusion can be implemented with $\Theta(\max(1, \log_r N))$ RMR time complexity on either CC or DSM machines.* □

It is possible to combine the arbitration-tree algorithm just described with an adaptive mutual exclusion algorithm presented previously by us [10]. The adaptive algorithm utilizes two trees, called *renaming* and *overflow trees*, respectively, as illustrated in Figure 6. A "name" is associated with each node within the renaming tree. A process begins execution by attempting to acquire one of these names. It does so by entering the renaming tree at its root and descending. As a process descends this tree, it may either stop, move left, or move right at a node. If a process stops at a node, then it begins moving upwards within the renaming tree — in this phase of its execution, the renaming tree is just an arbitration tree. It is possible for a process to fail to acquire a name within the renaming tree, in which case it enters the overflow tree at a designated leaf node. A process can fail to acquire a name only if the point contention [8] it experiences exceeds (asymptotically) the renaming tree's height. The overflow tree is simply a second arbitration tree. An extra two-process mutual exclusion algorithm is used to arbitrate between the "winning" processes coming from these two trees. As shown in [10], if a process experiences point contention k, then it will acquire a name in the renaming tree in $O(k)$ time (provided the renaming tree is of sufficient height). In this case, its overall RMR time complexity for entering and exiting its critical section is $O(k)$. By defining the renaming tree's height to be $\Theta(\log N)$, the

[6] In the conference version of this paper [3], we presented a DSM algorithm as a direct transformation of ALGORITHM G-CC. We later discovered that an additional "**if**" clause is necessary, as explained here.

[7] In [3], we stated that the above transformation could also be applied to the algorithm of T. Anderson [5]. However, Hyonho Lee of the University of Toronto later found that this assertion is not correct because in that algorithm it is possible for more than two processes to invoke one of the added two-process mutual exclusion algorithms at the same time. Nonetheless, a similar transformation can be applied to that algorithm where the availability of the *fetch-and-increment* primitive is exploited to avoid the `Entry` and `Exit` calls.

[8] The *point contention* experienced by a process p is the maximum number of processes that are *simultaneously* active (*i.e.*, outside of their noncritical sections) over a computation that starts when p becomes active and ends when it once again becomes inactive [1].

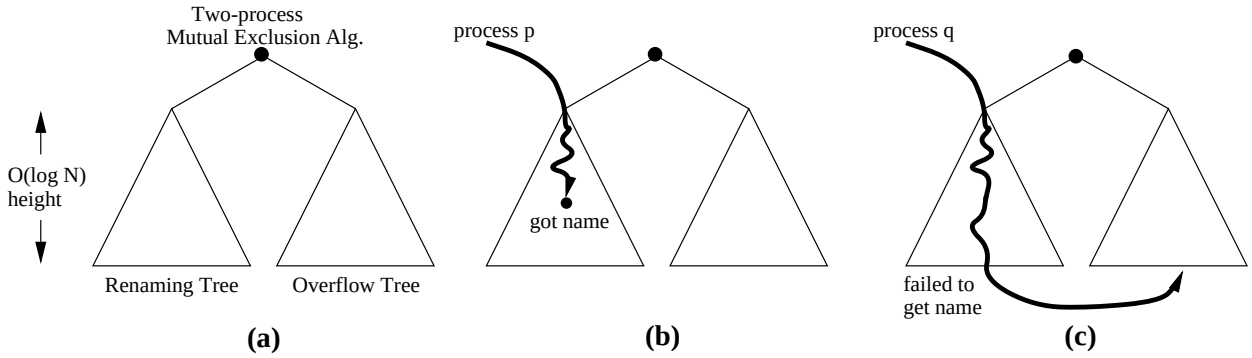

Figure 6: **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

overall RMR time complexity becomes $O(\min(k, \log N))$. The renaming tree is implemented using only atomic reads and writes. If we change its height to $\Theta(\max(1, \log_r N))$ and use the *fetch-and-ϕ*-based arbitration-tree algorithm described above to implement the overflow tree, then the overall RMR time complexity becomes $O(\min(k, \max(1, \log_r N)))$. Thus, we have the following theorem.

Theorem 2 *Using any fetch-and-ϕ primitive of rank $r \geq 2$, starvation-free adaptive mutual exclusion can be implemented with $O(\min(k, \max(1, \log_r N)))$ RMR time complexity on either CC or DSM machines, where k is point contention.* □

In presenting our algorithms, we have assumed that process identifiers range over $0, \ldots, N-1$. With one exception, this assumption is simply a matter of convenience. That one exception is the code sequence in lines 13–15 of Algorithm G-CC in Figure 2, where a process p with $p.pos = k$, where $0 \leq k < N$, waits until process k is either inactive or is accessing the same queue as p. This code sequence, which is part of the reset mechanism, creates a linkage between the range of process identifiers and the range of queue positions. This linkage can be eliminated by introducing a third queue, which each process uses only for its *first* critical-section execution. Because each process uses this queue only once, it does not need to be reset. If a process p obtains $p.pos = k$ in this queue, then it uses the process identifier k in its subsequent critical-section executions, which are implemented using the two queues considered earlier. Because three queues are being used now, a three-process mutual exclusion algorithm must be used instead of a two-process algorithm in order to arbitrate among the "winning" processes from these queues.

4 Concluding Remarks

We have presented a ranking of *fetch-and-ϕ* primitives based on the time complexity with which such primitives can be used to implement mutual exclusion. We have also shown that any *fetch-and-ϕ* primitive of rank r can be used to implement a $\Theta(\max(1, \log_r N))$ mutual exclusion algorithm, on either DSM or CC machines. $\Theta(\max(1, \log_r N))$ is clearly optimal for $r = \Omega(N)$. However, as remarked earlier, it follows from work appearing in the second author's Ph.D. dissertation [9] that $\Theta(\max(1, \log_r N))$ is *not* optimal for certain "self-resettable" primitives of rank at least three. Devising asymptotically optimally algorithms for primitives of arbitrary ranks remains as future work. It is important to note that, in designing these algorithms, our main goal was to achieve certain asymptotic time complexities. In particular, we have *not* concerned ourselves with designing algorithms that can be practically applied. Indeed, it is difficult to design practical algorithms when assuming so little of the *fetch-and-ϕ* primitives being used. It is likely that by exploiting the semantics of a particular primitive, our algorithms could be optimized considerably.

We believe that the notion of rank defined in this paper may be a suitable way of characterizing the "power" of primitives from the standpoint of blocking synchronization, much like the notion of a *consensus number*, which is used in Herlihy's wait-free hierarchy [8], reflects the "power" of primitives from the standpoint of

nonblocking synchronization. Interestingly, primitives like *compare-and-swap* that are considered to be powerful according to Herlihy's hierarchy are weak from a blocking synchronization standpoint (since they are subject to our $\Omega(\log N / \log \log N)$ lower bound [2]). Also, primitives like *fetch-and-increment* and *fetch-and-store* that are considered to be powerful from a blocking synchronization standpoint are considered quite weak according to Herlihy's hierarchy. (They have consensus number two.) This difference arises because in nonblocking algorithms, the need to reach consensus is fundamental (as shown by Herlihy), while in blocking algorithms, the need to order competing processes is important.

References

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. ACM, May 1999.

[2] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2003.

[3] J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 538–547. IEEE, May 2003.

[4] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[5] 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.

[6] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 148–156. IEEE, December 1993.

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

[8] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[9] Y.-J. Kim. *Time Complexity Bounds for Shared-memory Mutual Exclusion.* Ph.D. thesis, University of North Carolina, Chapel Hill, NC, 2003.

[10] Y.-J. Kim and J. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, to appear.

[11] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171. IEEE, April 1994.

[12] 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.

[13] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

Appendix A: Correctness Proof for Algorithm G-CC

In this appendix, we formally prove that Algorithm G-CC, presented in Section 3.1, satisfies the Exclusion and Starvation-Freedom properties. We use invariants and *leads-to* and *unless* properties in our proof. To define such notions, some additional definitions are required. We define a *state* of a program to be an assignment of values to the variables of the program, including process program counters. An *event* is a particular execution of a statement. A *history* of a program is a sequence $t_1,\ e_1,\ t_2,\ e_2,\ \ldots$, where each t_j is a state, each e_j is an event, t_1 is an initial state, and state t_{j+1} can be reached from state t_j via the execution of the statement corresponding to e_j. A history is *fair* if and only if each continuously enabled statement is eventually executed. A non-**await** statement is enabled if the program counter of the process in which it appears equals the label of that statement. (We further comment on statement labels below.) To ease the reasoning a bit, we assume that each **await** statement induces a single state transition, even though it actually represents a busy-waiting loop. That is, we assume that a statement of the form "**await** B" is enabled if the above condition for non-**await** statements holds, and in addition, the boolean condition B is true; when executed, such a statement causes a single transition to the next executable statement. An assertion is an *invariant* for a program if and only if it is true at each state of every history of that program. Letting A and B denote assertions, A *leads-to* B is true for a program if and only if the following holds for any *fair* history of the program: if A holds at some state in the history, then B holds at either the same state or some later state. A *unless* B is true for a program if and only if, for any history of the program, if $A \wedge \neg B$ holds at some state in the history, then $A \vee B$ holds at the next state. Informally, A is not falsified until B is established.

The following notational conventions are used in the remainder of this appendix.

Notational Conventions: We use $s.p$ to denote the statement with label s of process p, and $p.v$ to represent p's private variable v (see Footnote 5). Let S be a subset of the statement labels in process p. Then, $p@S$ holds if and only if the program counter for process p equals some value in S. (Note that if s is a statement label, then $p@\{s\}$ means that statement s of process p is *enabled*, *i.e.*, p has not yet executed s.)

We assume that each labeled sequence of statement(s) is atomic. For example, consider statement 15 of Figure 2. When executed by process p, this statement (atomically) updates $p.prev$, $p.self$, and $p.counter$, and establishes $p@\{6\}$ if the value assigned to $p.prev$ differs from $\perp$, and establishes $p@\{8\}$, otherwise. We number statements in this manner to reduce the number of cases that must be considered in the correctness proof. It can be seen that each labeled statement accesses at most one non-auxiliary (see below) shared variable, and does so via a single read, write, or *fetch-and-ϕ* operation. □

Our proof also makes use of a number of auxiliary variables. In Figure 7, Algorithm G-CC is shown with these added auxiliary variables, which are accessed only by statements 5 and 18. We begin with a description of these variables.

Private auxiliary variable $p.\mathsf{position}$ keeps track of p's position in the queue; as shown by invariant (I49) below, when p is in its exit section, $p.\mathsf{position}$ equals the non-auxiliary variable $p.pos$.

For $i = 0$ or 1, $\mathsf{HistLen}[i]$, $\mathsf{Hist}[i][0..\infty]$, and $\mathsf{Param}[i][0..\infty]$ keep the history of queue i since the last time it was initialized. (As explained in Section 3.1, each queue is accessed at most $2N$ times before it is re-initialized. Thus, only finite prefixes of $\mathsf{Hist}[i][\ldots]$ and $\mathsf{Param}[i][\ldots]$ are actually used.)

Example. Assume that queue i is initially empty. Then, initially we have the following.

$$\begin{aligned} \mathit{Tail}[i] &= \perp; \\ \mathsf{HistLen}[i] &= 0; \\ \mathsf{Hist}[i] &= (\perp,\ \perp,\ \perp,\ \ldots); \\ \mathsf{Param}[i] &= (\perp,\ \perp,\ \perp,\ \ldots). \end{aligned}$$

Now suppose that processes p, q, r, and p execute statement 5 (in that order), and that the private variables $p.counter$, $q.counter$, and $r.counter$ initially equal 0, 3, and 5, respectively.

If the underlying *fetch-and-ϕ* primitive is *fetch-and-store* with $2N+1$ distinct values (as defined in Section 2),

shared variables
CurrentQueue: 0..1;
Tail: **array**[0..1] **of** *Vartype* **initially** $\perp$;
Position: **array**[0..1] **of** $0..2N-1$ **initially** 0;
Signal: **array**[0..1][*Vartype*] **of boolean initially** *false*;
Active: **array**[$0..N-1$] **of boolean initially** *false*;
QueueIdx: **array**[$0..N-1$] **of** $(\perp, 0..1)$

type
ParamType = **record**
proc: $0..N-1$;
counter: **integer**
end

shared auxiliary variables
HistLen: **array**[0..1] **of** $0..\infty$ **initially** 0;
Hist: **array**[0..1][$0..\infty$] **of** *Vartype* **initially** $\perp$;
Param: **array**[0..1][$0..\infty$] **of** ParamType **initially** $\perp$

private variables
idx: 0..1;
counter: **integer**;
prev, *self*, *tail*: *Vartype*;
pos: $0..2N-1$

private auxiliary variable
position: $0..\infty$

process p :: /* $0 \le p < N$ */

while *true* **do**
0: Noncritical Section;
1: $QueueIdx[p] := \perp$;
2: $Active[p] := true$;
3: $idx := CurrentQueue$;
4: $QueueIdx[p] := idx$;
5: /* atomically execute lines 5a–5h */
5a: $prev := Tail[idx]$;
5b: $self := \phi(prev, \alpha_p[counter])$;
5c: $Tail[idx] := self$;
5d: position := HistLen[idx];
5e: Param[idx][position] := (p, *counter*);
5f: Hist[idx][position + 1] := *self*;
5g: HistLen[idx] := position + 1;
5h: *counter* := *counter* + 1;
if $prev \neq \perp$ **then**
6: **await** $Signal[idx][prev]$;
7: $Signal[idx][prev] := false$
fi;
8: $\texttt{Entry}_2(idx)$;
9: Critical Section;
10: $pos := Position[idx]$;
11: $Position[idx] := pos + 1$;
12: $\texttt{Exit}_2(idx)$;
13: **if** $(pos < N) \wedge (pos \neq p)$ **then**
14: **await** $\neg Active[pos] \vee$
15: $(QueueIdx[pos] = idx)$
elseif $pos = N$ **then**
16: $tail := Tail[1 - idx]$;
17: $Signal[1 - idx][tail] := false$;
18: /* atomically execute lines 18a–18d */
18a: $Tail[1 - idx] := \perp$;
18b: HistLen[$1 - idx$] := 0;
18c: **forall** j **do** Param[$1 - idx$][j] := $\perp$ **od**;
18d: **forall** j **do** Hist[$1 - idx$][j] := $\perp$ **od**;
19: $Position[1 - idx] := 0$;
20: $CurrentQueue := 1 - idx$
fi;
21: $Signal[idx][self] := true$;
22: $Active[p] := false$
od

Figure 7: Algorithm G-CC with auxiliary variables added.

then we have the following after these four *fetch-and-ϕ* invocations take place.

$$\begin{aligned} \mathit{Tail}[i] &= (p,1); \\ \mathsf{HistLen}[i] &= 4; \\ \mathsf{Hist}[i] &= \big(\bot,\ (p,0),\ (q,1),\ (r,1),\ (p,1),\ \bot,\ \bot,\ \ldots\big); \\ \mathsf{Param}[i] &= \big((p,0),\ (q,3),\ (r,5),\ (p,1),\ \bot,\ \bot,\ \ldots\big). \end{aligned}$$

On the other hand, if the underlying *fetch-and-ϕ* primitive is *fetch-and-increment* (where we define $\bot = 0$), then $\mathit{Tail}[i]$ and $\mathsf{Hist}[i]$ have the following final value, while the other two variables hold the same values as above.

$$\begin{aligned} \mathit{Tail}[i] &= 4; \\ \mathsf{Hist}[i] &= \big(\bot\,(=0),\ 1,\ 2,\ 3,\ 4,\ \bot,\ \bot,\ \ldots\big). \end{aligned}$$

List of Invariants

We will establish the 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. Unless stated otherwise, we assume the following: i ranges over 0 and 1; h, j, k, and l range over $0..\infty$; x and y range over *Vartype*; p, q, and r range over $0..N-1$.

invariant **(Exclusion)** $\big|\{p :: p@\{9..12\}\}\big| \le 1$ (I1)

invariant $\big|\{p :: p@\{7..21\} \ \wedge\ p.idx = i\}\big| \le 1$ (I2)

invariant $p@\{6..22\} \ \wedge\ p.idx = i \ \wedge\ p.\mathsf{position} = h \ \Rightarrow$
$\mathsf{HistLen}[i] > h \ \wedge\ p.prev = \mathsf{Hist}[i][h] \ \wedge\ p.self = \mathsf{Hist}[i][h+1] \ \wedge\ \mathsf{Param}[i][h].proc = p$ (I3)

invariant $0 < \mathsf{HistLen}[i] < 2N \ \Rightarrow\ \mathit{Tail}[i] \neq \bot$ (I4)

invariant $p@\{8..21\} \ \wedge\ p.idx = i \ \Rightarrow\ (\forall x :: \mathit{Signal}[i][x] = \mathit{false})$ (I5)

invariant $\mathit{Signal}[i][x] = \mathit{true} \ \Rightarrow\ x = \mathsf{Hist}[i][\mathit{Position}[i]]$ (I6)

invariant $\mathit{Signal}[i][x] = \mathit{Signal}[i][y] = \mathit{true} \ \Rightarrow\ x = y$ (I7)

invariant $p@\{7\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Signal}[i][p.prev] = \mathit{true}$ (I8)

invariant $\big|\{p :: p@\{6,7\} \ \wedge\ p.idx = i \ \wedge\ p.prev = x\}\big| \le 1$ (I9)

invariant $\mathit{Position}[i] = q+1 \ \Rightarrow$
$q@\{0..3\} \ \vee\ q.idx = i \ \vee\ (\exists p :: p@\{12..20\} \ \wedge\ p.idx = i)$ (I10)

invariant $q+1 < \mathit{Position}[i] \le N \ \Rightarrow\ q@\{0..3\} \ \vee\ q.idx = i$ (I11)

invariant $p@\{11..13\} \ \wedge\ p.idx = i \ \wedge\ p.pos = N \ \Rightarrow$
$\mathit{CurrentQueue} = i \ \wedge\ (\forall q :: q@\{0..3\} \ \vee\ q.idx = i)$ (I12)

invariant $p@\{16..20\} \ \wedge\ p.idx = i \ \Rightarrow$
$\mathit{CurrentQueue} = i \ \wedge\ (\forall q :: q@\{0..3\} \ \vee\ q.idx = i)$ (I13)

invariant $p@\{11\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[i] = p.pos$ (I14)

invariant $p@\{12..21\} \ \wedge\ p.idx = i \ \Rightarrow\ \mathit{Position}[i] = p.pos + 1$ (I15)

invariant $\mathsf{HistLen}[i] > h \ \wedge\ \mathsf{Param}[i][h] = (p,c) \ \Rightarrow$
$\mathsf{Hist}[i][h+1] = \phi(\mathsf{Hist}[i][h],\ \alpha_p[c])$ (I16)

invariant $\mathit{Tail}[i] = \mathsf{Hist}[i][\mathsf{HistLen}[i]]$ (I17)

invariant $\mathsf{Hist}[i][0] = \bot$ (I18)

invariant $0 \le j < k < \mathsf{HistLen}[i] \ \wedge$
$\mathsf{Param}[i][j] = (p, c_1) \ \wedge\ \mathsf{Param}[i][k] = (p, c_2) \ \wedge$
$(\forall l : j < l < k :: \mathsf{Param}[i][l].proc \neq p) \ \Rightarrow$

$$c_2 = c_1 + 1 \tag{I19}$$

invariant $0 \le j < \mathsf{HistLen}[i] \ \wedge$
$\mathsf{Param}[i][j] = (p, c) \ \wedge$
$(\forall k : j < k < \mathsf{HistLen}[i] :: \mathsf{Param}[i][k].proc \neq p) \ \Rightarrow$
$(p.counter = c + 1 \ \wedge \ p@\{4..22\} \ \wedge \ p.idx = i) \ \vee$
$(p.counter = c + 1 \ \wedge \ p@\{0..3\}) \ \vee$
$(p@\{0..3\} \ \wedge \ CurrentQueue = 1 - i) \ \vee$
$(p@\{4..22\} \ \wedge \ p.idx = CurrentQueue = 1 - i)$ (I20)

invariant $0 \le \mathsf{HistLen}[i] \le 2N$ (I21)

invariant $0 \le Position[i] \le 2N$ (I22)

invariant $\mathsf{HistLen}[i] = 0 \ \Rightarrow \ (\forall x :: Signal[i][x] = false)$ (I23)

invariant $p@\{18\} \ \wedge \ p.idx = i \ \Rightarrow \ (\forall x :: Signal[1 - i][x] = false)$ (I24)

invariant $p@\{17\} \ \wedge \ p.idx = i \ \Rightarrow \ p.tail = Tail[1 - i]$ (I25)

invariant $p@\{7..11\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] = p.\mathsf{position}$ (I26)

invariant $p@\{12..21\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] = p.\mathsf{position} + 1$ (I27)

invariant $\big[\ \mathsf{HistLen}[i] - Position[i] = \big|\{p :: p@\{6..11\} \ \wedge \ p.idx = i\}\big| \ \big] \ \vee$
$(\exists p :: p@\{19\} \ \wedge \ p.idx = 1 - i)$ (I28)

invariant $\big|\{p :: p@\{6..22\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h\}\big| \le 1$ (I29)

invariant $CurrentQueue = i \ \Rightarrow$
$\big|\{p :: p@\{4, 5\} \ \wedge \ p.idx = 1 - i\}\big| \le 2N - \mathsf{HistLen}[1 - i]$ (I30)

invariant $Position[i] = N + 1 \ \Rightarrow$
$CurrentQueue = 1 - i \ \vee$
$(\exists p :: p@\{12..20\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N)$ (I31)

invariant $Position[i] > N + 1 \ \Rightarrow \ CurrentQueue = 1 - i$ (I32)

invariant $Active[p] = p@\{3..22\}$ (I33)

invariant $p@\{5..22\} \ \wedge \ p.idx = i \ \Rightarrow \ QueueIdx[p] = i$ (I34)

invariant $p@\{2..4\} \ \Rightarrow \ QueueIdx[p] = \bot$ (I35)

invariant $Position[i] \le N \ \Rightarrow$
$CurrentQueue = i \ \vee$
$(\exists p :: p@\{20\} \ \wedge \ p.idx = 1 - i) \ \vee$
$\big(Position[i] = 0 \ \wedge \ (\forall q :: q@\{0..3\} \ \vee \ q.idx = 1 - i)\big)$ (I36)

invariant $1 \le Position[i] \le N \ \Rightarrow \ CurrentQueue = i$ (I37)

invariant $p@\{4, 5\} \ \wedge \ p.idx = i \ \Rightarrow \ \mathsf{HistLen}[i] < 2N$ (I38)

invariant $p@\{19, 20\} \ \wedge \ p.idx = i \ \Rightarrow \ \mathsf{HistLen}[1 - i] = 0$ (I39)

invariant $p@\{20\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[1 - i] = 0$ (I40)

invariant $Position[i] \le h < \mathsf{HistLen}[i] \ \Rightarrow$
$(\exists p :: p@\{6..11\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h)$ (I41)

invariant $Position[i] = h > 0 \ \Rightarrow$
$Signal[i][\mathsf{Hist}[i][h]] = true \ \vee$
$\big(\exists p :: p@\{12..21\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h - 1\big) \ \vee$
$\big(\exists q :: q@\{8..11\} \ \wedge \ q.idx = i \ \wedge \ q.\mathsf{position} = h\big)$ (I42)

invariant $0 < j < k \le \mathsf{HistLen}[i] \ \wedge \ k < 2N \ \wedge$
$\mathsf{Param}[i][j - 1].proc \neq \mathsf{Param}[i][k - 1].proc \ \Rightarrow$
$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I43)

invariant $0 < j \le \mathsf{HistLen}[i] \ \wedge \ j < 2N \ \Rightarrow \ \mathsf{Hist}[i][j] \neq \bot$ (I44)

invariant $0 < j < k \le \mathsf{HistLen}[i] \ \wedge \ k < 2N \ \wedge$
$\mathsf{Param}[i][j - 1] = (p, c) \ \wedge \ \mathsf{Param}[i][k - 1] = (p, c + 1) \ \Rightarrow$

$$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k] \tag{I45}$$

invariant $Position[i] \leq h < \mathsf{HistLen}[i] \ \wedge \ \mathsf{Param}[i][h].proc = p \ \Rightarrow$
$p@\{6..11\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h$ (I46)

invariant $p@\{6\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] \leq p.\mathsf{position}$ (I47)

invariant $Position[i] \leq j < k < \mathsf{HistLen}[i] \ \Rightarrow \ \mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I48)

invariant $p@\{11..22\} \ \Rightarrow \ p.pos = p.\mathsf{position}$ (I49)

Proof of the Exclusion Property

We now prove that each of (I1)–(I48) is an invariant. For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.) If I is an implication (which is the case for most of our invariants), then it suffices to check only those program statements that may establish the antecedent of I, or that may falsify the consequent if executed while the antecedent holds.

invariant (Exclusion) $\big|\{p :: p@\{9..12\}\}\big| \leq 1$ (I1)

Proof: Since the $\texttt{Entry}_2$ and $\texttt{Exit}_2$ routines (statements 8 and 12) are assumed to be correct, (I1) follows easily from (I2). □

invariant $\big|\{p :: p@\{7..21\} \ \wedge \ p.idx = i\}\big| \leq 1$ (I2)

Proof: The only statements that may potentially falsify (I2) are $5.p$ and $6.p$. Statement $5.p$ may falsify (I2) by establishing $p@\{7..21\} \ \wedge \ p.idx = i$ only if executed when

$$p.idx = i \ \wedge \ Tail[i] = \bot \ \wedge \ q@\{7..21\} \ \wedge \ q.idx = i$$

holds, where q is any arbitrary process, different from p. (In this case, process p transits from statement 5 to statement 8.) By (I4) and (I38), $p@\{5\} \wedge p.idx = i \wedge Tail[i] = \bot$ implies $\mathsf{HistLen}[i] = 0$. However, by applying (I3) with 'p' $\leftarrow q$, we have $\mathsf{HistLen}[i] > q.\mathsf{position} \geq 0$, a contradiction. Thus, statement $5.p$ cannot falsify (I2).

Statement $6.p$ may falsify (I2) by establishing $p@\{7..21\} \ \wedge \ p.idx = i$ only if executed when

$$p.idx = i \ \wedge \ Signal[i][p.prev] = true \ \wedge \ q@\{7..21\} \ \wedge \ q.idx = i,$$

holds, where q is any arbitrary process, different from p. By (I5), we have $q@\{7\}$. Thus, by (I8), we have $Signal[i][q.prev] = true$, which in turn implies $p.prev = q.prev$ by (I7). However, by (I9), this implies $p = q$, a contradiction. Thus, statement $6.p$ cannot falsify (I2). □

invariant $p@\{6..22\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \Rightarrow$
$\mathsf{HistLen}[i] > h \ \wedge \ p.prev = \mathsf{Hist}[i][h] \ \wedge \ p.self = \mathsf{Hist}[i][h+1] \ \wedge \ \mathsf{Param}[i][h].proc = p$ (I3)

Proof: The only statement that may establish the antecedent is $5.p$, which may do so only if executed when $p.idx = i \ \wedge \ \mathsf{HistLen}[i] = h$ holds. In this case, $5.p$ establishes $\mathsf{HistLen}[i] = h + 1 \ \wedge \ p.self = \mathsf{Hist}[i][h+1] \ \wedge \ \mathsf{Param}[i][h].proc = p$. Moreover, by (I17), $Tail[i] = \mathsf{Hist}[i][h]$ holds before $5.p$ is executed, and hence, $p.prev = \mathsf{Hist}[i][h]$ holds afterward.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. As shown above, statement $5.p$ cannot falsify (I3). For $q \neq p$, if statement $5.q$ is executed when $\mathsf{HistLen}[i] > h$ holds, then it preserves $\mathsf{HistLen}[i] > h$, and does not update $\mathsf{Hist}[i][h]$, $\mathsf{Hist}[i][h+1]$, or $\mathsf{Param}[i][h]$. Thus, statement $5.q$ preserves (I3).

Statement $18.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds. However, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, this implies $p@\{0..3\} \ \vee \ p.idx = 1 - i$. Thus, in this case, the antecedent is false before and after the execution of $18.q$. □

invariant $0 < \mathsf{HistLen}[i] < 2N \;\Rightarrow\; \mathit{Tail}[i] \neq \bot$ (I4)

Proof: Invariant (I4) follows easily by applying (I44) with 'j' $\leftarrow \mathsf{HistLen}[i]$, and using (I17). □

invariant $p@\{8..21\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; (\forall x :: \mathit{Signal}[i][x] = \mathit{false})$ (I5)

Proof: The only statements that may establish the antecedent are $5.p$ and $7.p$. Statement $5.p$ may establish the antecedent only if executed when $p.\mathit{idx} = i \;\wedge\; \mathit{Tail}[i] = \bot$ holds. In this case, by (I4) and (I38), we have $\mathsf{HistLen}[i] = 0$. Hence, by (I23), the consequent is true before and after the execution of $5.p$.

Statement $7.p$ may establish the antecedent only if executed when $p@\{7\} \;\wedge\; p.\mathit{idx} = i$ holds. In this case, by (I7) and (I8), $\mathit{Signal}[i][p.\mathit{prev}]$ is the only entry among $\mathit{Signal}[i][\ldots]$ that is *true*. Thus, statement $7.p$ establishes the consequent.

The only statement that may falsify the consequent is $21.q$, where q is any arbitrary process. Statement $21.q$ may potentially falsify (I5) only if executed when $p@\{8..21\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; q@\{21\} \;\wedge\; q.\mathit{idx} = i$ holds, which implies $p = q$ by (I2). Thus, $21.q$ falsifies the antecedent in this case. □

invariant $\mathit{Signal}[i][x] = \mathit{true} \;\Rightarrow\; x = \mathsf{Hist}[i][\mathit{Position}[i]]$ (I6)

Proof: The only statement that may establish the antecedent is $21.p$, which may do so only if $p.\mathit{idx} = i \;\wedge\; p.\mathit{self} = x$ holds. In this case, before $21.p$ is executed, $\mathit{Position}[i] = p.\mathsf{position}+1$ and $p.\mathit{self} = \mathsf{Hist}[i][p.\mathsf{position}+1]$ hold, by (I27) and (I3), respectively. Thus, the consequent is true before and after the execution of $21.p$.

The only statements that my falsify the consequent are $5.p$ and $18.p$ (which may update $\mathsf{Hist}[i][\mathit{Position}[i]]$) and $11.p$ (which may update $\mathit{Position}[i]$), where p is any arbitrary process. Statement $5.p$ may update $\mathsf{Hist}[i][\mathit{Position}[i]]$ only if executed when

$$p@\{5\} \;\wedge\; p.\mathit{idx} = i \tag{1}$$

holds. In this case, $5.p$ updates only one entry of $\mathsf{Hist}[i][\ldots]$, namely, $\mathsf{Hist}[i][h+1]$, where h is the value of $\mathsf{HistLen}[i]$ before its execution. Moreover, by (I28), either $h \geq \mathit{Position}[i]$ holds, or there exists a process q satisfying $q@\{19\} \;\wedge\; q.\mathit{idx} = 1 - i$. In the former case, statement $5.p$ does not update $\mathsf{Hist}[i][\mathit{Position}[i]]$, and hence preserves (I5). In the latter case, by (I13), we have $p@\{0..3\} \;\vee\; p.\mathit{idx} = 1 - i$, which contradicts (1). It follows that the latter case in fact cannot arise.

Statement $18.p$ may update $\mathsf{Hist}[i][\mathit{Position}[i]]$ only if executed when $p.\mathit{idx} = 1 - i$ holds, in which case, by (I24), the antecedent of (I6) is false before and after its execution. Similarly, statement $11.p$ may update $\mathit{Position}[i]$ only if executed when $p.\mathit{idx} = i$ holds, in which case, by (I5), the antecedent is false before and after its execution. □

invariant $\mathit{Signal}[i][x] = \mathit{Signal}[i][y] = \mathit{true} \;\Rightarrow\; x = y$ (I7)

Proof: This invariant follows trivially from (I6). □

invariant $p@\{7\} \;\wedge\; p.\mathit{idx} = i \;\Rightarrow\; \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}$ (I8)

Proof: The only statement that may establish the antecedent is $6.p$, which may do so only if the consequent holds.

The only statements that may falsify the consequent are $5.p$ (which assigns $p.\mathit{prev}$) and $7.q$ and $17.q$, where q is any arbitrary process. After the execution of $5.p$, $p@\{7\}$ is false. Statement $7.q$ may potentially falsify (I8) only if executed when $p@\{7\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; q@\{7\} \;\wedge\; q.\mathit{idx} = i$ holds. In this case, by (I2), we have $p = q$, and hence $7.q$ falsifies the antecedent. Statement $17.q$ may falsify the consequent only if executed when $q.\mathit{idx} = 1 - i$ holds, which implies that $p@\{0..3\} \;\vee\; p.\mathit{idx} = 1 - i$ holds, by (I13). Thus, the antecedent is false before and after the execution of $17.q$. □

invariant $\big|\{p :: p@\{6,7\} \;\wedge\; p.\mathit{idx} = i \;\wedge\; p.\mathit{prev} = x\}\big| \leq 1$ (I9)

Proof: Assume the following.

$$p@\{6,7\} \ \wedge \ q@\{6,7\} \ \wedge \ p.idx = q.idx = i \ \wedge \ p.prev = q.prev = x.$$

Our proof obligation is to show $p = q$. Define $j = p.\mathsf{position}$ and $k = q.\mathsf{position}$. By (I3), we have $(p.prev = \mathsf{Hist}[i][j] = x) \wedge (j < \mathsf{HistLen}[i])$. Similarly, we also have $(q.prev = \mathsf{Hist}[i][k] = x) \wedge (k < \mathsf{HistLen}[i])$. Moreover, by (I26) and (I47), we have $j \geq Position[i]$ and $k \geq Position[i]$.

Combining these assertions, by (I48), we have $j = k$. By (I29), this implies $p = q$. □

invariant $Position[i] = q + 1 \ \Rightarrow$
$$q@\{0..3\} \ \vee \ q.idx = i \ \vee \ (\exists p :: p@\{12..20\} \ \wedge \ p.idx = i) \tag{I10}$$

Proof: The only statements that may establish the antecedent are $11.r$ and $19.r$, where r is any arbitrary process. Statement $11.r$ may establish the antecedent only if executed when $r.idx = i$ holds, in which case it establishes the last disjunct of the consequent. If $19.r$ updates $Position[i]$, then it establishes $Position[i] = 0$, and hence cannot establish the antecedent. (Recall that q is assumed to range over $0..N-1$.)

The only statement that may falsify $q@\{0..3\} \ \vee \ q.idx = i$ is $3.q$, which may do so only if executed when $CurrentQueue = 1 - i$ holds, which in turn implies that $Position[i] = 0 \ \vee \ Position[i] > N$ holds, by (I37). Thus, since $0 \leq q < N$, the antecedent is false before and after $3.q$ is executed.

The only statements that may falsify $p@\{12..20\} \ \wedge \ p.idx = i$ are $13.p$, $14.p$, $15.p$, and $20.p$. Assume that the antecedent holds before the execution of each of these statements. By (I15), we have $p.pos = q < N$, and hence statement $13.p$ establishes $p@\{14\}$, and statement $20.p$ cannot be executed (since $p.pos = N$ holds when it is executed).

Statement $14.p$ may falsify $p@\{12..20\}$ only if executed when $Active[q] = false$ holds. By (I33), this implies that $q@\{0..2\}$ holds, and hence the consequent of (I10) holds after $14.p$ is executed. Statement $15.p$ may falsify $p@\{12..20\} \ \wedge \ p.idx = i$ only if executed when $QueueIdx[q] = i$ holds. By (I34) and (I35), this implies $q@\{0,1\} \ \vee \ q.idx = i$, and hence the consequent of (I10) holds after its execution. □

invariant $q + 1 < Position[i] \leq N \ \Rightarrow \ q@\{0..3\} \ \vee \ q.idx = i$ (I11)

Proof: The only statements that may establish the antecedent are $11.r$ and $19.r$, where r is any arbitrary process. Statement $11.r$ updates $Position[i]$ only if executed when $r.idx = i$ holds, in which case, by (I14), it increments $Position[i]$ by one. Therefore, statement $11.r$ may establish the antecedent only if executed when $Position[i] = q+1$ holds. In this case, by (I10), either the consequent holds, or there exists a process p satisfying $p@\{12..20\} \ \wedge \ p.idx = i$, before the execution of $11.r$. However, since we have $r@\{11\} \ \wedge \ r.idx = i$, the latter is precluded by (I2). Therefore, the consequent is true before $11.r$ is executed, and hence it is also true afterward.

If statement $19.r$ updates $Position[i]$, then it establishes $Position[i] = 0$, and hence cannot establish the antecedent.

The only statement that may falsify the consequent is $3.q$, which may do so only if $CurrentQueue \neq i$ holds. In this case, by (I37), the antecedent is false before and after the execution of $3.q$. □

invariant $p@\{11..13\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N \ \Rightarrow$
$$CurrentQueue = i \ \wedge \ (\forall q :: q@\{0..3\} \ \vee \ q.idx = i) \tag{I12}$$

Proof: The only statement that may establish the antecedent is $10.p$, which may do so only if executed when

$$p@\{10\} \ \wedge \ p.idx = i \ \wedge \ Position[i] = N \tag{2}$$

holds. By (I37), this implies that $CurrentQueue = i$ holds before and after $10.p$ is executed.

In order to prove that $(\forall q :: q@\{0..3\} \ \vee \ q.idx = i)$ holds after the execution of $10.p$, we consider two cases, depending on the value of q. If $0 \leq q < N - 1$, then by (I11), $q@\{0..3\} \ \vee \ q.idx = i$ holds before and after the execution of $10.p$. On the other hand, if $q = N - 1$, then since $Position[i] = N$, by (I10), either

$q@\{0..3\} \ \vee \ q.idx = i$ holds, or there exists a process r (different from p) satisfying $r@\{12..20\} \ \wedge \ r.idx = i$. However, the latter is precluded by (2) and (I2).

The only statement that may falsify $CurrentQueue = i$ is $20.r$ (where r is any arbitrary process), which may do so only if executed when $r.idx = i$ holds. Taken together with the antecedent, this implies that $r = p$ holds, by (I2). Thus, statement $20.r$ falsifies the antecedent in this case.

The only statement that may falsify $q@\{0..3\} \ \vee \ q.idx = i$ is $3.q$. However, if $3.q$ is executed when the consequent is true, then $3.q$ establishes $q.idx = i$, and hence preserves the consequent. □

invariant $p@\{16..20\} \ \wedge \ p.idx = i \ \Rightarrow$
$$CurrentQueue = i \ \wedge \ (\forall q :: q@\{0..3\} \ \vee \ q.idx = i) \tag{I13}$$

Proof: The only statement that may establish the antecedent is $13.p$, which may do so only if executed when $p.idx = i \ \wedge \ p.pos = N$ holds. In this case, by (I12), the consequent holds before and after the execution of statement $13.p$.

The only statements that may falsify the consequent are $3.q$ and $20.q$ (where q is any arbitrary process). However, each preserves the consequent as shown in the proof of (I12). □

invariant $p@\{11\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] = p.pos$ (I14)
invariant $p@\{12..21\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] = p.pos + 1$ (I15)

Proof: The only statement that may establish the antecedent of (I14) (respectively, (I15)) is $10.p$ (respectively, $11.p$), which clearly establishes the corresponding consequent.

The only other statements that may falsify either consequent are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify either consequent only if executed when $q.idx = i$ holds. Taken together with either antecedent, this implies that $q = p$ holds, by (I2). Thus, statement $11.q$ falsifies the antecedent of (I14), and establishes the antecedent and consequent of (I15).

Statement $19.q$ may falsify either consequent only if executed when $q.idx = 1 - i \ \wedge \ q.pos = N$ holds. By (I13), this implies that $p@\{0..3\} \ \vee \ p.idx = 1 - i$ holds. Hence, the antecedents of (I14) and (I15) are false before and after the execution of $19.q$. □

invariant $\mathsf{HistLen}[i] > h \ \wedge \ \mathsf{Param}[i][h] = (p, c) \ \Rightarrow$
$$\mathsf{Hist}[i][h+1] = \phi(\mathsf{Hist}[i][h], \ \alpha_p[c]) \tag{I16}$$

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (Note that statement $18.q$ assigns $\mathsf{HistLen}[1 - q.idx] := 0$, and hence cannot establish the antecedent.)

Statement $5.q$ may establish the antecedent only if executed when $\mathsf{HistLen}[i] = h \ \wedge \ q.idx = i$ holds. In this case, by (I17), $5.q$ establishes

$$\mathsf{Hist}[i][h+1] = q.self = \phi(\mathsf{Hist}[i][h], \ \alpha_q[c']) \quad \text{and} \quad \mathsf{Param}[i][h] = (q, c'),$$

where c' is the value of $q.counter$ before the execution of $5.q$. Thus, the antecedent is established only if $q = p$ and $c = c'$, in which case the consequent easily follows.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. Statement $5.q$ may falsify the consequent only if executed when $q.idx = i \ \wedge \ (\mathsf{HistLen}[i] = h - 1 \ \vee \ \mathsf{HistLen}[i] = h)$ holds. If $\mathsf{HistLen}[i] = h - 1$ holds before its execution, then $\mathsf{HistLen}[i] = h$ holds after its execution, and hence the antecedent is false. On the other hand, if $\mathsf{HistLen}[i] = h$ holds before its execution, then statement $5.q$ preserves (I16) as shown above.

Statement $18.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds, in which case it establishes $\mathsf{HistLen}[i] = 0$, and hence the antecedent is false after its execution. □

invariant $Tail[i] = \mathsf{Hist}[i][\mathsf{HistLen}[i]]$ (I17)
invariant $\mathsf{Hist}[i][0] = \perp$ (I18)

Proof: These invariants follow trivially from inspecting the code. In particular, lines 5c, 5f and 5g, as well as lines 18a and 18d, ensure that (I17) is maintained. Also, since $\mathsf{HistLen}[i]$ is always nonnegative (by (I21)), line 5f cannot update $\mathsf{Hist}[i][0]$, and hence (I18) is maintained. □

invariant $0 \le j < k < \mathsf{HistLen}[i] \ \wedge$
$\mathsf{Param}[i][j] = (p, c_1) \ \wedge \ \mathsf{Param}[i][k] = (p, c_2) \ \wedge$
$(\forall l : j < l < k :: \mathsf{Param}[i][l].proc \neq p) \ \Rightarrow$
$c_2 = c_1 + 1$ (I19)

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (Note that statement $18.q$ assigns $\mathsf{HistLen}[1-1.idx] := 0$ and does not update any entry of Param.) Since $5.q$ increments $\mathsf{HistLen}[q.idx]$ by one, it may establish the antecedent only if executed when $q.idx = i \ \wedge \ \mathsf{HistLen}[i] = k$ holds. Note that $5.q$ establishes $\mathsf{Param}[i][k] = (q, c)$ in this case, where c is the value of $q.counter$ before the execution of $5.q$. Thus, $5.q$ may establish the antecedent only if executed when the following holds.

$$q = p \ \wedge \ c_2 = q.counter \ \wedge \ q@\{5\} \ \wedge \ q.idx = i \ \wedge$$
$$0 \le j < k = \mathsf{HistLen}[i] \ \wedge$$
$$\mathsf{Param}[i][j] = (p, c_1) \ \wedge$$
$$(\forall l : j < l < k :: \mathsf{Param}[i][l].proc \neq p).$$

Thus, by applying (I20) with 'c' $\leftarrow c_1$, the first disjunct of the consequent of (I20) follows, and hence we have $c_2 = q.counter = c_1 + 1$. □

invariant $0 \le j < \mathsf{HistLen}[i] \ \wedge$
$\mathsf{Param}[i][j] = (p, c) \ \wedge$
$(\forall k : j < k < \mathsf{HistLen}[i] :: \mathsf{Param}[i][k].proc \neq p) \ \Rightarrow$
$(p.counter = c + 1 \ \wedge \ p@\{4..22\} \ \wedge \ p.idx = i) \ \vee$ $\mathcal{A}$
$(p.counter = c + 1 \ \wedge \ p@\{0..3\}) \ \vee$ $\mathcal{B}$
$(p@\{0..3\} \ \wedge \ CurrentQueue = 1 - i) \ \vee$ $\mathcal{C}$
$(p@\{4..22\} \ \wedge \ p.idx = CurrentQueue = 1 - i)$ $\mathcal{D}$
(I20)

Proof: The only statement that may establish the antecedent is $5.q$, where q is any arbitrary process. (As with (I19), $18.q$ need not be considered here.) Since $5.q$ increments $\mathsf{HistLen}[q.idx]$ by one, it may establish the antecedent only if executed when $q.idx = i \wedge \mathsf{HistLen}[i] = j$ holds. Note that $5.q$ establishes $\mathsf{Param}[i][k] = (q, c')$, where c' is the value of $q.counter$ before the execution of $5.q$. Thus, the antecedent may be established only if $q = p \ \wedge \ c' = c$ holds. It follows that statement $5.q$ establishes disjunct $\mathcal{A}$ in this case.

Disjunct $\mathcal{A}$ may be falsified only by statements $5.p$ (which may update $p.counter$) and $22.p$ (which may falsify $p@\{4..22\}$). If statement $5.p$ is executed while the antecedent holds, then it establishes $\mathsf{Param}[i][\mathsf{HistLen}[i]-1] = (p, c+1)$, and hence falsifies the last conjunct of the antecedent. Statement $22.p$ establishes disjunct $\mathcal{B}$ if executed when disjunct $\mathcal{A}$ holds.

Disjunct $\mathcal{B}$ may be falsified only by statement $3.p$, which establishes either disjunct $\mathcal{A}$ or disjunct $\mathcal{D}$, depending on the value of $CurrentQueue$.

Disjunct $\mathcal{C}$ may be falsified only by statements $3.p$ and $20.q$, where q is any arbitrary process. Statement $3.p$ establishes disjunct $\mathcal{D}$ if executed when disjunct $\mathcal{C}$ holds. Statement $20.q$ may falsify disjunct $\mathcal{C}$ only if executed when $q.idx = 1 - i$. In this case, by (I39), $\mathsf{HistLen}[i] = 0$ holds before and after the execution of $20.q$, and hence the antecedent of (I20) is false before and after its execution.

Disjunct $\mathcal{D}$ may be falsified only by statements $22.p$ and $20.q$, where q is any arbitrary process. Statement $22.p$ establishes disjunct $\mathcal{C}$ if executed when disjunct $\mathcal{D}$ holds. As shown above, the antecedent is false after the execution of $20.q$. □

invariant $0 \le \mathsf{HistLen}[i] \le 2N$ (I21)

Proof: The only statement that may potentially falsify (I21) is $5.p$, where p is any arbitrary process. Since $5.p$ increments $\mathsf{HistLen}[p.idx]$ by one, it may falsify (I21) only if executed when $p.idx = i \ \wedge \ \mathsf{HistLen}[i] = 2N$ holds. However, this is precluded by (I38). □

invariant $0 \leq Position[i] \leq 2N$ (I22)

Proof: The only statement that may potentially falsify (I22) is $11.p$ (where p is any arbitrary process), which may do so only if executed when $p.idx = i$. In this case, by (I14) and (I26), statement $11.p$ establishes $Position[i] = p.pos + 1 = p.\mathsf{position} + 1$. Moreover, by (I3) and (I21), $p.\mathsf{position} < \mathsf{HistLen}[i] \leq 2N$ holds before $11.p$ is executed. Thus, statement $11.p$ preserves (I22). □

invariant $\mathsf{HistLen}[i] = 0 \ \Rightarrow \ (\forall x :: Signal[i][x] = false)$ (I23)

Proof: The only statement that may establish the antecedent is $18.p$, where p is any arbitrary process. Statement $18.p$ may establish the antecedent only if executed when $p@\{18\} \ \wedge \ p.idx = 1 - i$ holds. In this case, by (I24), the consequent holds before and after $18.p$ is executed.

The only statement that may falsify the consequent is $21.p$, where p is any arbitrary process. Statement $21.p$ may falsify the consequent only if executed when $p.idx = i$ holds. In this case, by (I3), $\mathsf{HistLen}[i] > p.\mathsf{position}$ holds before and after $21.p$ is executed. Thus, the antecedent is false before and after the execution of $21.p$. □

invariant $p@\{18\} \ \wedge \ p.idx = i \ \Rightarrow \ (\forall x :: Signal[1 - i][x] = false)$ (I24)

Proof: The only statement that may establish the antecedent is $17.p$, which may do so only if $p.idx = i$ holds. Assume that $Signal[1 - i][x] = true$ holds for some x before the execution of $17.p$. It suffices to show $x = p.tail$.

By (I6), we have

$$x = \mathsf{Hist}[1 - i][Position[1 - i]]. \tag{3}$$

Also, by (I13), we have

$$(\forall q :: q@\{0..3\} \ \vee \ q.idx = i). \tag{4}$$

Moreover, by (I2), $p@\{17\} \ \wedge \ p.idx = i$ implies

$$\neg(\exists r :: r@\{19\} \ \wedge \ r.idx = i). \tag{5}$$

Combining (4) and (5), and applying (I28) with 'i' $\leftarrow 1 - i$, we have $\mathsf{HistLen}[1 - i] = Position[1 - i]$. Hence, by (3) and (I17), we have

$$x = \mathsf{Hist}[1 - i][\mathsf{HistLen}[1 - i]] = Tail[1 - i].$$

Thus, by (I25), $x = p.tail$ holds.

The only statement that may falsify the consequent is $21.q$, where q is any arbitrary process. Statement $21.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds. However, when the antecedent holds, $q@\{21\} \ \wedge \ q.idx = 1 - i$ is false, by (I13). □

invariant $p@\{17\} \ \wedge \ p.idx = i \ \Rightarrow \ p.tail = Tail[1 - i]$ (I25)

Proof: The only statement that may establish the antecedent is $16.p$, which may do so only if $p.idx = i$ holds. In this case, $16.p$ establishes the consequent.

The only statements that may falsify the consequent are $5.q$ and $18.q$, where q is any arbitrary process. Statement $5.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds, which implies that the antecedent is false, by (I13). Similarly, statement $18.q$ may falsify the consequent only if executed when $q.idx = i$ holds, which implies that the antecedent is false, by (I2). □

invariant $p@\{7..11\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[i] = p.\mathsf{position}$ (I26)

Proof: The only statements that may establish the antecedent are $5.p$ and $6.p$. Statement $5.p$ may establish the antecedent only if executed when

$$p@\{5\} \ \wedge \ \mathit{Tail}[i] = \bot \ \wedge \ p.\mathit{idx} = i \tag{6}$$

holds. In this case, by (I4) and (I38), we have $\mathsf{HistLen}[i] = 0$. Thus, statement $5.p$ establishes $p.\mathsf{position} = 0$. By (I28), $\mathsf{HistLen}[i] = 0$ also implies that either $\mathit{Position}[i] = 0$ or $(\exists q :: q@\{19\} \ \wedge \ q.\mathit{idx} = 1 - i)$ holds. In the former case, the consequent is established. In the latter case, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, we have $p@\{0..3\} \ \vee \ p.\mathit{idx} = 1 - i$, which contradicts (6). It follows that the latter case in fact cannot arise.

Statement $6.p$ may establish the antecedent only if executed when

$$p@\{6\} \ \wedge \ p.\mathit{idx} = i \ \wedge \ \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true} \tag{7}$$

holds. By (I6), this implies $p.\mathit{prev} = \mathsf{Hist}[i][\mathit{Position}[i]]$. Let $k = p.\mathsf{position}$. By (I3) (with 'h' $\leftarrow k$) and (I47), we have $p.\mathit{prev} = \mathsf{Hist}[i][k]$ and $\mathit{Position}[i] \leq k < \mathsf{HistLen}[i]$. If $k > \mathit{Position}[i]$, then by applying (I48) with 'j' $\leftarrow \mathit{Position}[i]$, we have $\mathsf{Hist}[i][\mathit{Position}[i]] \neq \mathsf{Hist}[i][k]$, a contradiction. It follows that $\mathit{Position}[i] = k = p.\mathsf{position}$ holds before the execution of $6.p$. Thus, it also holds after its execution.

The only statements that may falsify the consequent are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify the consequent (when the antecedent holds) only if executed when $q.\mathit{idx} = i$ holds. Taken together with the antecedent, this implies that $q = p$ holds, by (I2). Thus, statement $11.q$ falsifies the antecedent.

Statement $19.q$ may falsify the consequent only if executed when $q.\mathit{idx} = 1 - i$ holds. By applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, this implies $p@\{0..3\} \ \vee \ p.\mathit{idx} = 1 - i$. Hence, the antecedent is false before and after the execution of $19.q$. □

invariant $\quad p@\{12..21\} \ \wedge \ p.\mathit{idx} = i \ \Rightarrow \ \mathit{Position}[i] = p.\mathsf{position} + 1$ (I27)

Proof: The only statement that may establish the antecedent is $11.p$, which may do so only if $p.\mathit{idx} = i$ holds. In this case, by (I14) and (I26), $11.p$ establishes the consequent.

The only statements that may falsify the consequent (while the antecedent holds) are $11.q$ and $19.q$, where q is any arbitrary process. Statement $11.q$ may falsify the consequent only if executed when $q@\{11\} \ \wedge \ q.\mathit{idx} = i$ holds. In this case, if $q = p$, then statement $11.p$ preserves (I27) as shown above. If $q \neq p$, then by (I2), the antecedent is false before and after the execution of $11.q$.

The proof that $19.q$ preserves (I27) is similar to that given in the proof of (I26). □

invariant $\quad \big[\ \mathsf{HistLen}[i] - \mathit{Position}[i] = \big|\{p :: p@\{6..11\} \ \wedge \ p.\mathit{idx} = i\}\big| \ \big] \ \vee$
$(\exists p :: p@\{19\} \ \wedge \ p.\mathit{idx} = 1 - i)$ (I28)

Proof: Define

$$X = \big|\{p :: p@\{6..11\} \ \wedge \ p.\mathit{idx} = i\}\big| \, .$$

The only statements that may potentially falsify (I28) are $5.q$ (which may modify $\mathsf{HistLen}[i]$ and X), $11.q$ (which may modify $\mathit{Position}[i]$ and X), $18.q$ (which may modify $\mathsf{HistLen}[i]$), and $19.q$ (which may modify $\mathit{Position}[i]$ and also falsify the second disjunct), where q is any arbitrary process.

Statements $5.q$ and $11.q$ may modify $\mathsf{HistLen}[i]$, $\mathit{Position}[i]$, or X only if executed when $q.\mathit{idx} = i$ holds. In this case, $5.q$ increments both $\mathsf{HistLen}[i]$ and X by one, and hence preserves (I28). Similarly, by (I14), $11.q$ increments $\mathit{Position}[i]$ and decrements X by one, and hence preserves (I28).

Statement $18.q$ may update $\mathsf{HistLen}[i]$ only if executed when $p.\mathit{idx} = 1 - i$, in which case it establishes the second disjunct. Statement $19.q$ may falsify the second disjunct only if executed when $q.\mathit{idx} = 1 - i$. In this case, by (I13) and (I39) (with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$), we have $X = 0$ and $\mathsf{HistLen}[i] = 0$, respectively. Since $19.q$ establishes $\mathit{Position}[i] = 0$, it establishes the first disjunct of (I28). □

invariant $\quad \big|\{p :: p@\{6..22\} \ \wedge \ p.\mathit{idx} = i \ \wedge \ p.\mathsf{position} = h\}\big| \leq 1$ (I29)

Proof: Assume that there exists a process p satisfying $p@\{6..22\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h$. By (I3), this implies $\mathsf{HistLen}[i] > h$.

The only statement that may potentially falsify (I29) is $5.q$, where q is any arbitrary process different from p. Statement $5.q$ may falsify (I29) only if executed when $q.idx = i$ holds. However, since $\mathsf{HistLen}[i] > h$, statement $5.q$ establishes $q.\mathsf{position} > h$, and hence cannot increase the left-hand side of (I29). □

invariant $CurrentQueue = i \ \Rightarrow$
$$\left|\{p :: p@\{4, 5\} \ \wedge \ p.idx = 1 - i\}\right| \leq 2N - \mathsf{HistLen}[1 - i] \tag{I30}$$

Proof: The only statement that may establish the antecedent is $20.q$, where q is any arbitrary process. Let X denote the value of
$$\left|\{p :: p@\{6..11\} \ \wedge \ p.idx = 1 - i\}\right|$$
prior to the execution of $20.q$. Statement $20.q$ may establish the antecedent only if executed when
$$q@\{20\} \ \wedge \ q.idx = 1 - i \tag{8}$$
holds, which also implies the following.
$$\left|\{p :: p@\{6..11, 20\} \ \wedge \ p.idx = 1 - i\}\right| \geq X + 1$$

By (8), and by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, $\neg(\exists r :: r@\{19\} \ \wedge \ r.idx = i)$ holds, and hence, by (I28), we have
$$\mathsf{HistLen}[1 - i] - Position[1 - i] = X.$$

Also, since $20.q$ may be executed only if $q.pos = N$ holds, by applying (I15) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, we have
$$Position[1 - i] = N + 1.$$

Combining these assertions, we have the following.
$$\begin{aligned}
\left|\{p :: p@\{4, 5\} \ \wedge \ p.idx = 1 - i\}\right| &\leq N - X - 1 \\
&= N - (\mathsf{HistLen}[1 - i] - Position[1 - i]) - 1 \\
&= N - (\mathsf{HistLen}[1 - i] - N - 1) - 1 \\
&= 2N - \mathsf{HistLen}[1 - i].
\end{aligned}$$

Thus, the consequent holds before the execution of $20.q$, and hence it also holds after its execution.

The only statements that may falsify the consequent are $3.q$ (which may increment X) and $5.q$ (which may increment $\mathsf{HistLen}[1 - i]$), where q is any arbitrary process. If statement $3.q$ is executed while the antecedent holds, then it establishes $q.idx = i$, and hence cannot increment X. Statement $5.q$ may increment $\mathsf{HistLen}[1 - i]$ (by one) only if executed when $q.idx = 1 - i$ holds, in which case it also decrements X by one, and hence preserves the consequent. □

invariant $Position[i] = N + 1 \ \Rightarrow$
$$CurrentQueue = 1 - i \ \vee$$
$$(\exists p :: p@\{12..20\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N) \tag{I31}$$

Proof: The only statement that may establish the antecedent is $11.q$, where q is any arbitrary process. However, if $11.q$ establishes the antecedent, then by (I14), it also establishes the second disjunct of the consequent.

The only statements that may falsify the consequent are $20.q$ (which may update *CurrentQueue*) and $13.p$, $14.p$, $15.p$, and $20.p$ (which may falsify $p@\{12..20\} \ \wedge \ p.idx = i \ \wedge \ p.pos = N$), where q is any arbitrary process. Statement $20.q$ may falsify the consequent only if executed when $q.idx = 1 - i$ holds. In this case, by (I40), $Position[i] = 0$ holds before and after the execution of $20.q$. Thus, the antecedent is false before and after $20.q$ is executed.

Since $p.pos = N$, statement $13.p$ establishes $p@\{16\}$, and statements $14.p$ and $15.p$ cannot be executed. If statement $20.p$ is executed when $p.idx = i$ holds, then it establishes $CurrentQueue = 1 - i$, and hence preserves the consequent. □

invariant $Position[i] > N + 1 \;\Rightarrow\; CurrentQueue = 1 - i$ (I32)

Proof: The only statement that may establish the antecedent is $11.p$, where p is any arbitrary process. By (I14), $11.p$ may establish the antecedent only if executed when $p@\{11\} \wedge p.idx = i \wedge Position[i] = N+1$ holds. In this case, by (I2) and (I31), we have the consequent.

The only statement that may falsify the consequent is $20.q$, where q is any arbitrary process. As shown in the proof of (I31), if $20.q$ falsifies the consequent, then the antecedent is false after its execution. □

invariant $Active[p] = p@\{3..22\}$ (I33)
invariant $p@\{5..22\} \wedge p.idx = i \;\Rightarrow\; QueueIdx[p] = i$ (I34)
invariant $p@\{2..4\} \;\Rightarrow\; QueueIdx[p] = \perp$ (I35)

Proof: These invariants follow trivially from inspecting Algorithm G-CC. □

invariant $Position[i] \leq N \;\Rightarrow$
$CurrentQueue = i \;\vee$ $\mathcal{A}$
$(\exists p :: p@\{20\} \wedge p.idx = 1 - i) \;\vee$ $\mathcal{B}$
$\big(Position[i] = 0 \;\wedge\; (\forall q :: q@\{0..3\} \;\vee\; q.idx = 1 - i)\big)$ $\mathcal{C}$
(I36)

Proof: The only statements that may establish the antecedent are $11.r$ and $19.r$, where r is any arbitrary process. However, if statement $11.r$ updates $Position[i]$, then by (I14), it increments $Position[i]$ by one. It follows that, although statement $11.r$ may preserve the antecedent, it cannot establish it. Statement $19.r$ may establish the antecedent only if executed when $r.idx = 1 - i$ holds, in which case it establishes disjunct $\mathcal{B}$.

The only statement that may falsify disjunct $\mathcal{A}$ is $20.r$, where r is any arbitrary process. Statement $20.r$ may falsify disjunct $\mathcal{A}$ only if executed when $r.idx = i \;\wedge\; r.pos = N$ holds, which implies that $Position[i] = N + 1$ holds, by (I15). Thus, in this case, the antecedent is false before and after the execution of $20.r$.

The only statement that may falsify disjunct $\mathcal{B}$ is $20.p$, which establishes disjunct $\mathcal{A}$.

The only statements that may falsify disjunct $\mathcal{C}$ are $3.q$ and $11.q$, where q is any arbitrary process. Statement $3.q$ may falsify disjunct $\mathcal{C}$ only if executed when $CurrentQueue = i$ holds, in which case disjunct $\mathcal{A}$ holds before and after its execution. Statement $11.q$ may falsify disjunct $\mathcal{C}$ only if executed when $q@\{11\} \wedge q.idx = i$ holds, which is precluded when disjunct $\mathcal{C}$ holds. □

invariant $1 \leq Position[i] \leq N \;\Rightarrow\; CurrentQueue = i$ (I37)

Proof: By (I36), the antecedent implies one of the following.

$$\begin{array}{ll} \mathcal{A}: & CurrentQueue = i, \\ \mathcal{B}: & (\exists p :: p@\{20\} \;\wedge\; p.idx = 1 - i), \quad \text{or} \\ \mathcal{C}: & Position[i] = 0 \;\wedge\; (\forall q :: q@\{0..3\} \;\vee\; q.idx = 1 - i). \end{array}$$

By (I40), $\mathcal{B}$ implies $Position[i] = 0$. Also, $\mathcal{C}$ clearly implies $Position[i] = 0$. Thus, both are precluded by the antecedent. It follows that $\mathcal{A}$ is true. □

invariant $p@\{4, 5\} \;\wedge\; p.idx = i \;\Rightarrow\; \mathsf{HistLen}[i] < 2N$ (I38)

Proof: For the sake of contradiction, assume

$$p@\{4, 5\} \;\wedge\; p.idx = i \;\wedge\; \mathsf{HistLen}[i] \geq 2N. \tag{9}$$

By applying (I30) with 'i' $\leftarrow 1 - i$, we have $CurrentQueue \neq 1 - i$, *i.e.*,

$$CurrentQueue = i. \tag{10}$$

Thus, by (I32), we have

$$Position[i] \leq N + 1. \tag{11}$$

Also, (9) implies

$$\big|\{q :: q@\{6..11\} \ \wedge \ q.idx = i\}\big| \leq N - 1.$$

Hence, by (I28), we have

$$\mathsf{HistLen}[i] - Position[i] \leq N - 1 \quad \vee \quad (\exists r :: r@\{19\} \ \wedge \ r.idx = 1 - i).$$

However, if there exists a process r satisfying $r@\{19\} \ \wedge \ r.idx = 1 - i$, then by (I13) (with '$p$' $\leftarrow r$ and 'i' $\leftarrow 1-i$), we have $p@\{0..3\} \vee p.idx = 1-i$, which contradicts (9). Therefore, we have $\mathsf{HistLen}[i] - Position[i] \leq N-1$.

Note that the only common solution to $\mathsf{HistLen}[i] \geq 2N$ (given in (9)), (11), and $\mathsf{HistLen}[i] - Position[i] \leq N - 1$ is $\mathsf{HistLen}[i] = 2N$ and

$$Position[i] = N + 1.$$

By (10) and (I31), this implies that $(\exists r :: r@\{12..20\} \ \wedge \ r.idx = i)$ holds. From this and (9), we have

$$\big|\{q :: q@\{6..11\} \ \wedge \ q.idx = i\}\big| \leq N - 2.$$

Hence, by (I28), we have

$$\mathsf{HistLen}[i] - Position[i] \leq N - 2 \quad \vee \quad (\exists r :: r@\{19\} \ \wedge \ r.idx = 1 - i).$$

The second disjunct is precluded by (I13) as shown above, and hence we have $\mathsf{HistLen}[i] - Position[i] \leq N-2$. However, this cannot hold simultaneously with (9) and (11). Thus, we have reached a contradiction. □

invariant $p@\{19, 20\} \ \wedge \ p.idx = i \ \Rightarrow \ \mathsf{HistLen}[1 - i] = 0$ (I39)

Proof: The antecedent may be established only by statement 18.p, which may do so only if $p.idx = i$ holds. In this case, 18.p also establishes the consequent.

The only statement that may falsify the consequent is 5.q, where q is any arbitrary process. Statement 5.q may potentially falsify (I39) only if executed when

$$p@\{19, 20\} \ \wedge \ p.idx = i \ \wedge \ q@\{5\} \ \wedge \ q.idx = 1 - i$$

holds. However, this contradicts (I13). □

invariant $p@\{20\} \ \wedge \ p.idx = i \ \Rightarrow \ Position[1 - i] = 0$ (I40)

Proof: The antecedent may be established only by statement 19.p, which may do so only if $p.idx = i$ holds. In this case, 19.p also establishes the consequent.

The only statement that may falsify the consequent is 11.q, where q is any arbitrary process. Statement 11.q may potentially falsify (I40) only if executed when $p@\{20\} \ \wedge \ p.idx = i \ \wedge \ q@\{11\} \ \wedge \ q.idx = 1 - i$ holds. However, this contradicts (I13). □

invariant $Position[i] \leq h < \mathsf{HistLen}[i] \ \Rightarrow$
$(\exists p :: p@\{6..11\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h)$ (I41)

Proof: The only statements that may establish the antecedent are 5.q and 18.q (which may modify $\mathsf{HistLen}[i]$) and 11.q and 19.q (which may modify $Position[i]$), where q is any arbitrary process. Statement 5.q may establish $\mathsf{HistLen}[i] > h$ only if executed when $q.idx = i \wedge \mathsf{HistLen}[i] = h$ holds, in which case it establishes the consequent.

If statement 18.q modifies $\mathsf{HistLen}[i]$, then it establishes $\mathsf{HistLen}[i] = 0$, and hence falsifies the antecedent.

Statement 11.q may modify $Position[i]$ only if executed when $q.idx = i$ holds. In this case, by (I14), it increments $Position[i]$ by one. Hence, although 11.q may preserve the antecedent, it cannot establish it.

Statement 19.q may establish $Position[i] \leq h$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is 11.p, which may do so only if executed when $p.idx = i \ \wedge \ p.\mathsf{position} = h$ holds (which comes from the consequent itself, which is assumed to hold before being falsified). In this case, by (I14) and (I26), statement 11.p establishes $Position[i] = h + 1$, and hence falsifies the antecedent. □

invariant $Position[i] = h > 0 \ \Rightarrow$
$$\begin{array}{ll} Signal[i][\mathsf{Hist}[i][h]] = true \ \vee & \mathcal{A} \\ \big(\exists p :: p@\{12..21\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h - 1\big) \ \vee & \mathcal{B} \\ \big(\exists q :: q@\{8..11\} \ \wedge \ q.idx = i \ \wedge \ q.\mathsf{position} = h\big) & \mathcal{C} \end{array} \tag{I42}$$

Proof: The only statement that may establish the antecedent is 11.p, where p is any arbitrary process. By (I14) and (I26), statement 11.p may establish the antecedent only if executed when $p.idx = i \ \wedge \ p.\mathsf{position} = h - 1$ holds, in which case it establishes disjunct $\mathcal{B}$.

The only statements that may falsify disjunct $\mathcal{A}$ are 5.p, 7.p, and 18.p, where p is any arbitrary process. Statement 5.p may falsify disjunct $\mathcal{A}$ only if executed when

$$p@\{5\} \ \wedge \ p.idx = i \ \wedge \ \mathsf{HistLen}[i] = h - 1 \tag{12}$$

holds. Combining this with the antecedent, and using (I28), this implies $(\exists q :: q@\{19\} \ \wedge \ q.idx = 1 - i)$. Hence, by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, we have $p@\{0..3\} \ \vee \ p.idx = 1 - i$, which contradicts (12). It follows that statement 5.p cannot falsify disjunct $\mathcal{A}$ while the antecedent holds.

Statement 7.p may falsify disjunct $\mathcal{A}$ only if executed when $p.idx = i$ holds, in which case, by (I26), we have $p.\mathsf{position} = Position[i] = h$. Hence, statement 7.$p$ establishes disjunct $\mathcal{C}$ in this case.

Statement 18.p may falsify disjunct $\mathcal{A}$ only if executed when $p.idx = 1 - i$ holds. In this case, by (I24), disjunct $\mathcal{A}$ is already false before 18.p is executed.

The only statement that may falsify disjunct $\mathcal{B}$ is 21.p (where p is any arbitrary process), which may do so only if executed when $p.idx = i \ \wedge \ p.\mathsf{position} = h - 1$ holds. In this case, by (I3), $p.self = \mathsf{Hist}[i][h]$ holds before its execution. Thus, statement 21.p establishes disjunct $\mathcal{A}$.

The only statement that may falsify disjunct $\mathcal{C}$ is 11.p (where p is any arbitrary process), which may do so only if executed when $p.idx = i \ \wedge \ p.\mathsf{position} = h$ holds. In this case, by (I14) and (I26), 11.p establishes $Position[i] = h + 1$, and hence falsifies the antecedent. □

invariant $0 < j < k \leq \mathsf{HistLen}[i] \ \wedge \ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1].proc \neq \mathsf{Param}[i][k-1].proc \ \Rightarrow$
$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I43)

invariant $0 < j \leq \mathsf{HistLen}[i] \ \wedge \ j < 2N \ \Rightarrow \ \mathsf{Hist}[i][j] \neq \perp$ (I44)

invariant $0 < j < k \leq \mathsf{HistLen}[i] \ \wedge \ k < 2N \ \wedge$
$\mathsf{Param}[i][j-1] = (p, c) \ \wedge \ \mathsf{Param}[i][k-1] = (p, c+1) \ \Rightarrow$
$\mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I45)

Proof: Invariants (I43)–(I45) follow easily from invariants (I16), (I17), (I18), and (I19), together with the assumption that the underlying *fetch-and-ϕ* primitive has rank at least $2N$. In particular, (I43) states that any two invocations (among the first $2N - 1$) by different processes write different values to $Tail[i]$; (I44) states that each of the first $2N - 1$ invocations writes to $Tail[i]$ a value different from $\perp$; (I45) states that any two successive invocations (among the first $2N - 1$) by the same process write different values to $Tail[i]$. □

invariant $Position[i] \leq h < \mathsf{HistLen}[i] \ \wedge \ \mathsf{Param}[i][h].proc = p \ \Rightarrow$
$p@\{6..11\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h$ (I46)

Proof: The only statements that may establish the antecedent are 5.q and 18.q (which may update $\mathsf{HistLen}[i]$ and $\mathsf{Param}[i][h].proc$) and 11.$q$ and 19.q (which may update $Position[i]$), where q is any arbitrary process.

Statement 5.q may establish $h < \mathsf{HistLen}[i] \wedge \mathsf{Param}[i][h].proc = p$ only if executed when $\mathsf{HistLen}[i] = h \wedge p = q \wedge q.idx = i$ holds, in which case it establishes the antecedent.

Statement 18.q may update $\mathsf{HistLen}[i]$ or $\mathsf{Param}[i][h]$ only if $q.idx = 1 - i$, in which case it establishes $\mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

Statement 11.q may update $Position[i]$ only if executed when $q.idx = i$ holds. In this case, by (I14), it increments $Position[i]$ by one. Hence, although 11.q may preserve the antecedent, it cannot establish it.

Statement 19.q may establish $Position[i] \leq h$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is 11.p, which may do so only if executed when $p.idx = i \wedge p.\mathsf{position} = h$ holds. In this case, by (I14) and (I26), statement 11.p establishes $Position[i] = h + 1$, and hence the antecedent is false after its execution. □

invariant $p@\{6\} \wedge p.idx = i \Rightarrow Position[i] \leq p.\mathsf{position}$ (I47)

Proof: The only statement that may establish the antecedent is 5.p, which may do so only if $p.idx = i$. Let h be the value of $\mathsf{HistLen}[i]$ before the execution of 5.p. By (I28), we have either $Position[i] \leq h$ or $(\exists q :: q@\{19\} \wedge q.idx = 1-i)$. However, by applying (I13) with 'i' $\leftarrow 1-i$, the latter implies $p@\{0..3\} \vee p.idx = 1-i$, which implies that the antecedent is false. On the other hand, if $Position[i] \leq h$ holds before the execution of 5.p, then $Position[i] \leq h = p.\mathsf{position}$ is established.

The only statement that may falsify the consequent (while the antecedent holds) is 11.q (where q is any arbitrary process), which may do so only if $q.idx = i$. In this case, by (I14), 11.q increments $Position[i]$ by one, and hence it may falsify the consequent only if executed when $Position[i] = p.\mathsf{position}$ holds. By applying (I26) with 'p' $\leftarrow q$, we also have $Position[i] = q.\mathsf{position}$. Combining these assertions with the antecedent, and using (I29), we have $p = q$. However, in this case, the antecedent is false after the execution of 11.q. □

invariant $Position[i] \leq j < k < \mathsf{HistLen}[i] \Rightarrow \mathsf{Hist}[i][j] \neq \mathsf{Hist}[i][k]$ (I48)

Proof: The only statements that may establish the antecedent are 5.p and 18.p (which may update $\mathsf{HistLen}[i]$, $\mathsf{Hist}[i][j]$, or $\mathsf{Hist}[i][k]$) and 11.p and 19.p (which may update $Position[i]$), where p is any arbitrary process.

Statement 5.p may establish the antecedent only if executed when $p.idx = i \wedge Position[i] \leq j < k = \mathsf{HistLen}[i]$ holds. In this case, by (I38),

$$Position[i] \leq j < k = \mathsf{HistLen}[i] < 2N \tag{13}$$

holds before its execution. We consider two cases.

- If $j = 0$, then by (I18), we have $\mathsf{Hist}[i][j] = \perp$. Also, by applying (I44) with 'j' $\leftarrow k$, and using (13), we have $\mathsf{Hist}[i][k] \neq \perp$. Hence, the consequent of (I48) holds before and after the execution of 5.p.
- If $j > 0$, then let (q, c_1) denote the value of $\mathsf{Param}[i][j-1]$ and (r, c_2) denote the value of $\mathsf{Param}[i][k-1]$ prior to the execution of 5.p. If $q \neq r$, then by (13) and (I43), the consequent holds before and after the execution of 5.p.

  Therefore, assume that $q = r$. Let q_l denote the value of $\mathsf{Param}[i][l-1].proc$ prior to the execution of 5.p, for each $0 < l \leq k$. Then, we have $q = q_j = q_k$.

  For each l satisfying $Position[i] < l \leq k$, by applying (I46) with 'p' $\leftarrow q_l$ and 'h' $\leftarrow l-1$, and using (13), we have $q_l.\mathsf{position} = l-1$ prior to the execution of 5.p. In particular, we have $q.\mathsf{position} = q_k.\mathsf{position} = k-1$, and

  $$(\forall l : Position[i] < l < k :: q_l \neq q). \tag{14}$$

  Since $q_j = q$, this implies that $Position[i] < j < k$ is false. Thus, by (13), we have $j = Position[i]$. Combining this with (14), we also have

  $$(\forall l : j < l < k :: \mathsf{Param}[i][l-1].proc \neq q)$$

prior to the execution of $5.p$.

Therefore, by applying (I19) with 'p' $\leftarrow q$, 'j' $\leftarrow j-1$, and 'k' $\leftarrow k-1$, and using (13) and the assertions above, we have $c_2 = c_1 + 1$. Combining this with (13), and using (I45), it follows that the consequent holds both before and after the execution of $5.p$.

If statement $18.p$ updates $\mathsf{HistLen}[i]$, then it establishes $\mathsf{HistLen}[i] = 0$, and hence the antecedent is false after its execution.

If statement $11.p$ updates $Position[i]$, then by (I14), it increments $Position[i]$ by one. Hence, although $11.p$ may preserve the antecedent, it cannot establish it.

Statement $19.p$ may establish $Position[i] \leq j$ only if executed when $q.idx = 1 - i$ holds, in which case, by (I39), it establishes $Position[i] = \mathsf{HistLen}[i] = 0$. Thus, the antecedent is false after its execution.

The only statement that may falsify the consequent is $5.p$ (which may update either $\mathsf{Hist}[i][j]$ or $\mathsf{Hist}[i][k]$), where p is any arbitrary process. Statement $5.p$ may update $\mathsf{Hist}[i][j]$ only if executed when $p.idx = i \ \wedge \ \mathsf{HistLen}[i] = j - 1$ holds, in which case it establishes $\mathsf{HistLen}[i] = j$. Thus, in this case, the antecedent is false after $5.p$ is executed. Similar reasoning applies to $\mathsf{Hist}[i][k]$. □

invariant $p@\{11..22\} \ \Rightarrow \ p.pos = p.\mathsf{position}$ (I49)

Proof: The only statement that may establish the antecedent is $10.p$, which also establishes the consequent by (I26). The consequent cannot be falsified while the antecedent holds. □

Proof of Starvation-Freedom

To establish the Starvation-Freedom property, we begin with proving the following *unless* and *leads-to* properties. Informally, (U1) states that if a process p is waiting at statement 6, and if the busy-waiting condition is established, then it holds continuously until p exits the busy-waiting loop. (L1) (respectively, (L2)) is used to prove that, if p has entered the current queue (respectively, the old queue), and waits at statement 6, then the busy-waiting condition is eventually established. Throughout this section, we assume that the $\texttt{Entry}_2$ and $\texttt{Exit}_2$ routines are starvation-free.

$$p@\{6\} \ \wedge \ p.idx = i \ \wedge \ Signal[i][p.prev] = true \quad unless \quad p@\{7\} \tag{U1}$$

Proof: The only statement that may falsify $p@\{6\} \ \wedge \ p.idx = i$ is $6.p$, which establishes $p@\{7\}$.

$Signal[i][p.prev] = true$ may be falsified only if some process q ($\neq p$) executes statement $7.q$ when $q.idx = i \ \wedge \ q.prev = p.prev$ holds. However, if the left-hand side of (U1) is true, then this is precluded by (I9). □

$$\begin{array}{l} CurrentQueue = i \ \wedge \ p@\{6\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \wedge \ Position[i] = k \\ \quad leads\text{-}to \ \ (Signal[i][p.prev] = true \ \vee \ Position[i] = k + 1) \end{array} \tag{L1}$$

$$\begin{array}{l} CurrentQueue = 1 - i \ \wedge \ p@\{6\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h \ \wedge \ Position[i] = k \\ \quad leads\text{-}to \ \ (Signal[i][p.prev] = true \ \vee \ Position[i] = k + 1) \end{array} \tag{L2}$$

Proof: We prove (L1) and (L2) by induction on h. Since their proofs are nearly identical, we simply say the "left-hand side" when the argument applies to both (L1) and (L2).

First, assume $h = 0$. The assertion $p@\{6\} \ \wedge \ p.idx = i \ \wedge \ p.\mathsf{position} = h$ may be established only if statement $5.p$ is executed when $\mathsf{HistLen}[i] = 0$ holds. However, by (I17) and (I18), this implies that $Tail[i] = \mathsf{Hist}[i][0] = \perp$ holds. Thus, statement $5.p$ establishes $p@\{8\}$, and cannot establish the left-hand side. It follows that (L1) and (L2) hold vacuously for $h = 0$.

Now assume that $h > 0$, and that (L1) and (L2) hold for smaller values of h. Consider a state t that satisfies the left-hand side.

By (I13), the left-hand side implies $\neg(\exists q :: q@\{19\} \ \wedge \ q.idx = 1 - i)$. Thus, by (I28), and using $p@\{6\} \ \wedge \ p.idx = i$, we have $k < \mathsf{HistLen}[i]$. By applying (I41) with 'h' $\leftarrow k$, this in turn implies that a process q exists such that

$$q@\{6..11\} \ \wedge \ q.idx = i \ \wedge \ q.\mathsf{position} = Position[i] = k. \tag{15}$$

We consider two cases, depending on the value of k.

Case 1: $k = 0$. If $k = 0$, then by (I18), and by applying (I3) with 'p' $\leftarrow q$ and 'h' $\leftarrow k$, we have $q.prev = \mathsf{Hist}[i][0] = \bot$. Because $q@\{6, 7\} \Rightarrow q.prev \neq \bot$ is (trivially) an invariant, this implies that $q@\{8..11\}$ holds. Thus, q eventually executes statement 11 while $q.idx = i \wedge q.\mathsf{position} = k$ holds. In this case, by (I14) and (I26), $11.q$ establishes $Position[i] = k + 1$, and hence the right-hand side of (L1)/(L2) is established.

Case 2: $k > 0$. Let x be the value of $\mathsf{Hist}[i][k]$ at state t. By (I42), (15) implies that one of the following holds at state t, where r is some process.

$$\begin{array}{ll} \mathcal{A}: & Signal[i][x] = true, \\ \mathcal{B}: & r@\{12..21\} \wedge r.idx = i \wedge r.\mathsf{position} = k - 1, \quad \text{or} \\ \mathcal{C}: & r@\{8..11\} \wedge r.idx = i \wedge r.\mathsf{position} = k. \end{array}$$

Moreover, by (I3), $q.prev = x$ also holds at state t. We now prove that, in each of the three cases given by $\mathcal{A}$–$\mathcal{C}$, the right-hand side of (L1)/(L2) is eventually established. Toward this goal, we prove the following three claims.

Claim 1: If $\mathcal{A}$ is true at state u, where u is either t or some later state, then the right-hand side of (L1)/(L2) is true at either t or some later state.

Claim 2: If $\mathcal{B}$ is true at state t, then $\mathcal{A}$ is true at either t or some later state u.

Claim 3: If $\mathcal{C}$ is true at state t, then the right-hand side of (L1)/(L2) is true at either t or some later state.

Proof of Claim 1: First, if (15) is false at state u, then since it holds at state t, the execution of statement $11.q$ occurs between state t and state u. Note that $q.idx$ and $q.\mathsf{position}$ do not change while $q@\{6..11\}$ holds. Hence, by (I14) and (I26), $q.\mathsf{position} = q.pos = Position[i] = k$ holds before the execution of $11.q$. Therefore, $11.q$ establishes the right-hand side of (L1)/(L2).

On the other hand, assume that (15) is true at state u. Then, by (I3), $q.prev = x$ holds at state u. Moreover, by (U1), if $q@\{6\} \wedge q.prev = x$ holds at state u, then $\mathcal{A}$ continues to hold until $q@\{7\}$ is established. It follows that q eventually executes statement $11.q$, which establishes the right-hand side of (L1)/(L2) as shown above. □

Proof of Claim 2: Assume that $\mathcal{B}$ holds at state t. In this case, by (I3), $r.self = \mathsf{Hist}[i][k] = x$ holds at state t. Hence, if r eventually executes statement 21, then $21.r$ establishes $\mathcal{A}$.

Thus, it suffices to show that statement $21.r$ is eventually executed. Since $r@\{12..21\}$ holds at state t, it suffices to show the following.

> *If $r@\{14, 15\}$ holds at state u, where u is either t or some later state, then $r@\{21\}$ is eventually established.*

Clearly, $r@\{14, 15\}$ implies $0 \leq r.pos < N$, which is true at state t (where $\mathcal{B}$ holds) as well as state u. Thus, by (I15), $1 \leq Position[i] = r.pos + 1 \leq N$ holds at state t. Hence, by (I37), $CurrentQueue = i$ holds at state t. Thus, if the left-hand side of (L2) is true at state t, then $\mathcal{B}$ is false at state t.

On the other hand, if the left-hand side of (L1) is true at state t, then by (L7), given later, $r@\{21\}$ is eventually established. (Note that the proof of (L2) does not depend on (L7). As explained shortly, this is necessary in order to avoid circular reasoning.) □

Proof of Claim 3: Clearly, r eventually executes statement $11.r$ if $\mathcal{C}$ holds. Thus, by (I14) and (I26), $11.r$ establishes $Position[i] = k + 1$, and hence the right-hand side of (L1)/(L2) is established. □

Finally, from these three claims, (L1) and (L2) follow. □

The reader may wonder why we have two separate properties (L1) and (L2), when they can be proved in essentially the same way. The reason is that the proof of (L7), given later, indirectly depends on (L2). Since the proof of (L1) depends on (L7), (L1) and (L2) must be kept separate to avoid circular reasoning.

The following properties are consequences of (U1), (L1), and (L2).

$$\mathit{CurrentQueue} = i \ \wedge \ p@\{6\} \ \wedge \ p.\mathit{idx} = i \quad \mathit{leads\text{-}to} \quad p@\{7\} \tag{L3}$$
$$\mathit{CurrentQueue} = 1 - i \ \wedge \ p@\{6\} \ \wedge \ p.\mathit{idx} = i \quad \mathit{leads\text{-}to} \quad p@\{7\} \tag{L4}$$

Proof: Since $\mathit{Position}[i]$ is bounded by (I22), by inductively applying (L1) and (L2), respectively, we have the following.

$$\begin{array}{l} \mathit{CurrentQueue} = i \ \wedge \ p@\{6\} \ \wedge \ p.\mathit{idx} = i \ \ \mathit{leads\text{-}to} \\ \quad \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}; \end{array} \tag{16}$$

$$\begin{array}{l} \mathit{CurrentQueue} = 1 - i \ \wedge \ p@\{6\} \ \wedge \ p.\mathit{idx} = i \ \ \mathit{leads\text{-}to} \\ \quad \mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}. \end{array} \tag{17}$$

Assume that the left-hand side of (L3) holds at some state t. By (16), $\mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}$ is eventually established at some later state u. If $p@\{7\}$ is established before state u, then (L3) holds. Otherwise, $p@\{6\}$ continues to hold from state t to u. Thus, $p.\mathit{idx} = i$ also continues to hold from state t to u. Therefore, the left-hand side of (U1) holds at state u. By (U1), $\mathit{Signal}[i][p.\mathit{prev}] = \mathit{true}$ is not falsified until $p@\{7\}$ is established, and hence p eventually establishes $p@\{7\}$ by executing statement 6.

The reasoning for (L4) is similar, except that (17) is used instead of (16). □

Note that the proof of (L3) indirectly depends on (L7), while the proof of (L4) does not.

The following properties state that, if a process p is waiting for process q at statements 14 and 15, then the busy-waiting condition is eventually established. (Note that $q@\{0\}$ implies $\mathit{Active}[q] = \mathit{false}$ by (I33), and $q@\{6\} \ \wedge \ q.\mathit{idx} = i$ implies $\mathit{QueueIdx}[q] = i$ by (I34).)

$$\begin{array}{l} p@\{14, 15\} \ \wedge \ p.\mathit{idx} = i \ \wedge \ p.\mathit{pos} = q \\ \quad \mathit{leads\text{-}to} \ \ q@\{0\} \ \vee \ (q@\{6\} \ \wedge \ q.\mathit{idx} = i) \ \vee \ p@\{21\} \end{array} \tag{L5}$$

$$\begin{array}{l} p@\{14, 15\} \ \wedge \ p.\mathit{idx} = i \ \wedge \ p.\mathit{pos} = q \ \wedge \ q@\{1\} \\ \quad \mathit{leads\text{-}to} \ \ (q@\{6\} \ \wedge \ q.\mathit{idx} = i) \ \vee \ p@\{21\} \end{array} \tag{L6}$$

Proof: Assume that the left-hand side of either (L5) or (L6) holds at state t. By (I2), one of the following holds at t.

$$\begin{array}{ll} \mathcal{A}: & q@\{3..22\} \ \wedge \ q.\mathit{idx} = 1 - i; \\ \mathcal{B}: & q@\{22\} \ \wedge \ q.\mathit{idx} = i; \\ \mathcal{C}: & q@\{0\}; \\ \mathcal{D}: & q@\{1..3\}; \\ \mathcal{E}: & q@\{4, 5\} \ \wedge \ q.\mathit{idx} = i; \\ \mathcal{F}: & q@\{6\} \ \wedge \ q.\mathit{idx} = i. \end{array}$$

Also, by (I15), $\mathit{Position}[i] = q + 1$ holds at state t, and hence, by (I37), $\mathit{CurrentQueue} = i$ also holds at state t.

If $p@\{21\}$ is established at some future state, then (L5) and (L6) both hold. Thus, in the rest of the proof, we assume that $p@\{14, 15\} \ \wedge \ p.\mathit{idx} = i$ holds continuously at and after state t. We claim that $\mathit{CurrentQueue} = i$ also holds at all future states. Note that $\mathit{CurrentQueue} = i$ may be falsified only by statement $20.r$ (where r is any arbitrary process), which may do so only if executed when $r.\mathit{idx} = i$ holds. However, by (I2), this is precluded when $p@\{14, 15\} \ \wedge \ p.\mathit{idx} = i$ holds. Thus, we have the following.

- $p@\{14, 15\} \ \wedge \ p.\mathit{idx} = i \ \wedge \ \mathit{CurrentQueue} = i$ holds at t and all later states. (18)

Note that the left-hand side of (L6) implies $\mathcal{D}$, and $\mathcal{F}$ implies the right-hand side of (L6). Hence, in order to prove (L6), it suffices to prove the following.

- If $\mathcal{D} \vee \mathcal{E}$ holds at state u, where u is either t or some later state, then $\mathcal{F}$ is established at some state after u. (19)

Also, since $\mathcal{C} \vee \mathcal{F}$ implies the right-hand side of (L5), in order to prove (L5), it suffices to prove the following claim in addition to (19).

- If $\mathcal{A} \vee \mathcal{B}$ holds at state u, where u is either t or some later state, then $\mathcal{C}$ is established at some state after u. (20)

We prove (19) and (20) by considering each of $\mathcal{A}$, $\mathcal{B}$, $\mathcal{D}$, and $\mathcal{E}$.

- Assume that $\mathcal{A}$ holds at state u. We claim that, in this case, $\mathcal{B}$ is eventually established.

  It suffices to show that the busy-waiting loops at statement 6 and statements 14 and 15 eventually terminate for q. By applying (L4) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1 - i$, and using $\mathit{CurrentQueue} = i$ (given in (18)), it follows that the former loop eventually terminates. If $q@\{14, 15\}$, then by (I15), we have $\mathit{Position}[1 - i] = q.\mathit{pos} + 1$. Also, by (18) and (I37), we have $\mathit{Position}[1 - i] = 0 \vee \mathit{Position}[1 - i] > N$. Combining these two assertions, we have $q.\mathit{pos} = \mathit{Position}[1 - i] - 1 \geq N$, and hence $q@\{14, 15\}$ is false. It follows that q in fact does not execute the busy-waiting loop at statements 14 and 15 while $\mathcal{A}$ holds.

- Assume that $\mathcal{B}$ holds at state u. Clearly, $\mathcal{C}$ is eventually established.

- Assume that $\mathcal{D}$ holds at state u. Then, by (18), $\mathcal{E}$ is eventually established.

- Assume that $\mathcal{E}$ holds at state u. In this case, q eventually executes statement 5. By (I38), $\mathsf{HistLen}[i] < 2N$ holds before the execution of $5.q$. Moreover, by (18) and (I3), $\mathsf{HistLen}[i] > p.\mathsf{position} \geq 0$ also holds. (Note that $\mathsf{HistLen}[i]$ is always nonnegative by (I21). Since $p.\mathsf{position}$ is updated only by line 5d, it follows that $p.\mathsf{position}$ is always nonnegative.) Combining these two assertions with (I4), we have $\mathit{Tail}[i] \neq \perp$. It follows that statement $5.q$ establishes $\mathcal{F}$.

From the reasoning above, assertions (20) and (19) follow. Therefore, we have (L5) and (L6). □

Note that the proofs of (L5) and (L6) do not depend on (L7).

The following property states that the busy-waiting loop at statements 14 and 15 eventually terminates.

$p@\{14, 15\}$ *leads-to* $p@\{21\}$ (L7)

Proof: For the sake of contradiction, assume that $p@\{21\}$ is never established. Let $i = p.\mathit{idx}$ and $q = p.\mathit{pos}$. By (L5), $q@\{0\} \vee (q@\{6\} \wedge q.\mathit{idx} = i)$ is eventually established.

First, assume that $q@\{0\}$ is established. If q remains in its noncritical section forever, then by (I33), $\mathit{Active}[q] = \mathit{false}$ holds forever. Thus, p eventually establishes $p@\{21\}$ by executing statement 14, a contradiction.

On the other hand, if q enters its entry section again, then it establishes $q@\{1\}$. In this case, by (L6), q eventually establishes $q@\{6\} \wedge q.\mathit{idx} = i$.

It follows that $q@\{6\} \wedge q.\mathit{idx} = i$ is eventually established. By (I5), and using $p@\{14, 15\}$, it follows that $q@\{6\}$ remains true forever. But then, by (I34), p eventually establishes $p@\{21\}$ by executing statement 15, a contradiction. □

Finally, by (L3), (L4), and (L7), it follows that each **await** statement in Algorithm G-CC eventually terminates. Thus, Algorithm G-CC is starvation-free.

Appendix B: Correctness Proof for Algorithm G-CC′ and Algorithm G-DSM

In this appendix, we first prove that Algorithm G-CC′ is equivalent to Algorithm G-CC, and also prove that the CC-to-DSM transformation introduced in Section 3.2 can be safely applied to Algorithm G-CC′. The following lemma proves the former claim.

Lemma 3 *In* Algorithm G-CC*, if some process p executes statement 21 while $p.pos \geq 2N-1 \ \wedge \ p.idx = i \ \wedge \ p.self = x$ holds, and if some process p' later reads $Signal[i][x]$ by executing statement 6, then between these two events, some process p'' writes $Signal[i][x] := false$.*

Proof: Let s and t be the states right before the executions of statements $21.p$ and $6.p'$, respectively. First, consider state s. By (I15) and (I22), we have the following.

- $Position[i] = 2N$ holds at s. (21)

If any process q satisfies $q@\{16..20\} \ \wedge \ q.idx = 1-i$ at state s, then by applying (I13) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, every process r must satisfy $r@\{0..3\} \ \vee \ r.idx = 1-i$. However, this contradicts $p@\{21\} \ \wedge \ p.idx = i$. Therefore, we have the following.

- $\neg\big(\exists q :: q@\{16..20\} \ \wedge \ q.idx = 1-i\big)$ holds at s. (22)

Hence, by (I28), we have $\mathsf{HistLen}[i] - Position[i] = \big|\big\{r :: r@\{6..11\} \ \wedge \ r.idx = i\big\}\big|$. Taken together with (I21) and (21), we have

$$\mathsf{HistLen}[i] = 2N \quad \wedge \quad \neg\big(\exists r :: r@\{6..11\} \ \wedge \ r.idx = i\big).$$

By (I38), this also implies $\neg\big(\exists r :: r@\{4,5\} \ \wedge \ r.idx = i\big)$. In particular, process p' satisfies $\neg\big(p'@\{4..6\} \ \wedge \ p'.idx = i\big)$ at s. Since $p'@\{6\} \ \wedge \ p'.idx = i$ holds at state t, p' must execute statement 3 while $CurrentQueue = i$ holds *between* states s and t.

However, by (21) and (I32), $CurrentQueue = 1-i$ holds at state s. Hence, some process q must write $CurrentQueue := i$, by executing statement 20 while $q.idx = 1-i$ holds, *after* state s buf *before* the execution of $3.p'$.

Clearly, q also executes $18.q$ (while $q.idx = 1-i$ holds) at some earlier time. If $18.q$ is executed before state s, then $q@\{19,20\} \ \wedge \ q.idx = 1-i$ holds at s, which contradicts (22). Thus, we have the following sequence of events: $21.p$, $18.q$, $20.q$, $3.p'$, and $6.p'$. By applying (I24) with 'p' $\leftarrow q$ and 'i' $\leftarrow 1-i$, it follows that $Signal[i][x]$ equals *false* before the execution of $18.q$. Since $Signal[i][x] = true$ holds right after the execution of $21.p$, some process p'' must write $Signal[i][x] := false$, after the execution of $21.p$ but before the execution of $18.q$ (and hence, before the execution of $6.p'$).[9] □

By Lemma 3, whenever Algorithm G-CC′ causes a process to skip a write at statement 21, this behavior is guaranteed not to affect any process's behavior in the future. Moreover, as a result, all the invariants stated in Appendix A also remain valid for Algorithm G-CC′, except for those that explicitly refer to the *Signal* array.

We now formally prove that the various two-process mutual exclusion algorithms added to Algorithm G-DSM (shown in Figure 5) are safe, by considering each pair of `Entry`/`Exit` calls. Statements 4–7 and 45–48 are clearly safe, as explained in Section 3.2.

Statements 10–14: By (I9),

$$\big|\big\{p :: p@\{6,7\} \ \wedge \ p.idx = i \ \wedge \ p.prev = x\big\}\big| \leq 1$$

is an invariant of Algorithm G-CC′, for any i and x. This implies that the `Entry`/`Exit` pair at statements 10–14 in Figure 5 is safe. □

[9] In fact, it can be easily shown that p'' must be q.

Statements 25–30: In order to show that the `Entry`/`Exit` pair at statements 25–30 in Figure 5 is safe, it suffices to show that

$$\left|\left\{p :: p@\{14, 15, 21\} \ \wedge \ p.pos = h\right\}\right| \leq 1 \tag{23}$$

is an invariant of Algorithm G-CC′, for any h.

Consider a process p that satisfies $p@\{14, 15, 21\} \ \wedge \ p.pos = h$. By (I15) and (I27), $Position[p.idx] = h+1 \ \wedge \ p.\mathsf{position} = h$ holds as well. Since $h < N$ holds by assumption (see statement 13 in Figure 7), by (I36), we also have $p.idx = CurrentQueue$. Taken together with (I29), it follows that p is uniquely determined (*i.e.*, (23) is true). □

Statements 40–43: In order to show that the `Entry`/`Exit` pair at statements 40–43 in Figure 5 is safe, it suffices to show that

$$\left|\left\{p :: p@\{21, 22\} \ \wedge \ p.idx = i \ \wedge \ p.self = x \ \wedge \ p.pos < 2N - 1\right\}\right| \leq 1 \tag{24}$$

is an invariant of Algorithm G-CC′, for any i and x.

For the sake of contradiction, assume that there exists two distinct processes p and q, satisfying

$$\begin{array}{l} p@\{21, 22\} \ \wedge \ q@\{21, 22\} \ \wedge \\ p.idx = q.idx = i \ \wedge \ p.self = q.self = x \ \wedge \\ p.pos < 2N - 1 \ \wedge \ q.pos < 2N - 1. \end{array}$$

Define $j = p.pos$ and $k = q.pos$. By (I49), we also have $j = p.\mathsf{position}$ and $k = q.\mathsf{position}$. Hence, by (I29), $j \neq k$ holds. Without loss of generality, assume $j < k$.

By applying (I3) to p and q separately, we have

$$\begin{array}{l} j < k < \mathsf{HistLen}[i] \ \wedge \\ x = p.self = \mathsf{Hist}[i][j+1] \ \wedge \ x = q.self = \mathsf{Hist}[i][k+1] \ \wedge \\ \mathsf{Param}[i][j].proc = p \ \wedge \ \mathsf{Param}[i][k].proc = q. \end{array}$$

Therefore, by using $k+1 < 2N$, and applying (I43) with 'j' $\leftarrow j+1$ and 'k' $\leftarrow k+1$, we have $\mathsf{Hist}[i][j+1] \neq \mathsf{Hist}[i][k+1]$, a contradiction. □