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

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 $\Theta(\max(1, \log, N))$ 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 [4].
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 \textit{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 \textit{remote-memory-references (RMR)} measure [5]. 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 \textit{fetch-and-φ} primitives. A \textit{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 \(\text{var}\) is the variable accessed.

\[
\text{fetch-and-φ} (\text{var}, \text{input}) \\
\text{old} := \text{var}; \\
\text{var} := φ(\text{old}, \text{input}); \\
\text{return}(\text{old})
\]

In this paper, we distinguish between \textit{fetch-and-φ} primitives that are comparison primitives and those that are not. A \textit{comparison primitive} conditionally updates a shared variable after first testing that its value meets some condition; examples include \textit{compare-and-swap} and \textit{test-and-set}.\footnote{\textit{compare-and-swap} and \textit{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 [9]. It is straightforward to modify any algorithm that uses the boolean versions of these primitives to instead use the versions considered in this paper.} Non-comparison primitives update variables unconditionally; examples include \textit{fetch-and-increment} and \textit{fetch-and-store}.

In recent work [3], we established a time-complexity lower bound of \(Ω(\log N/ \log \log N)\) 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 \textit{fetch-and-φ} primitives [6, 8, 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 [6], Graunke and Thakkar [8], Mellor-Crummey and Scott [12], Craig [7], and Landin and Hagersten [11]. In each of these algorithms, blocked processes wait within a “spin queue.” A process enqueues itself by using a \textit{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 \textit{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 \textit{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 [7] and Landin and Hagersten [11] independently proposed the same algorithm, which is based on \textit{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.
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 \(\text{fetch-and}-\phi\) 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 [9], 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 \(\text{fetch-and}-\phi\) 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 \(\text{fetch-and}-\phi\)-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 \(\text{fetch-and}-\phi\) 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.

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 [10], 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 \(\text{fetch-and}-\phi\) 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 [10], the \(\text{fetch-and}-\phi\) 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 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 feature is exploited, with a resulting asymptotic improvement in time complexity for primitives of rank \(o(\log N)\). As explained in [10], 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.

²Herlihy’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.
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-\(\phi\)** primitive, as defined in Section 1. We will use “Vartype” to denote the type of the accessed variable \(\text{var}\). (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-\(\phi\) 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-\(\phi\) primitive is the largest integer \(r\) satisfying the following.

For each process \(p\), there exists a constant array \(\alpha[p][0..k_p - 1]\) of input values (for some \(k_p\)), such that if \(p\) performs a sequence of fetch-and-\(\phi\) invocations as specified below on a variable \(v\) (of type Vartype) that is initially \(\perp\) (for some choice of \(\perp\)),

\[
\text{for } i := a_p \text{ to } \infty \text{ do fetch-and-\(\phi\)}(v, \alpha[p][i \bmod k_p]) \text{ od}
\]

where \(a_p\) is some integer value, 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 consecutive 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-\(\phi\) primitive has infinite rank if the condition above is satisfied for arbitrarily large values of \(r\). \(\Box\)

As our generic algorithm shows, a fetch-and-\(\phi\) 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.

**Examples.** An \(r\)-bounded fetch-and-increment primitive on a variable \(v\) with range \(0, \ldots, r - 1\) is defined by \(\phi(\text{old, input}) = \min(r - 1, \text{old} + 1)\). (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 \(r^{\text{th}}\)) returns \(r - 1\), which is the same as the return value of the \(r^{\text{th}}\) 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 \(\alpha\) is extraneous. However, this is not the case for other primitives. As a second example, consider a fetch-and-store primitive on a variable that is large enough to hold \(2N + 1\) distinct values (\(2N\) pairs \((p, 0)\) and \((p, 1)\), where \(p\) is a process, and an additional initial value \(\perp\)). It is easily shown that fetch-and-store has infinite rank. This follows by defining \(\alpha[p][j] = (p, j \bmod 2)\). (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.

\(^3\)Progress properties other than starvation-freedom may be of interest as well.
3 A Constant-Time Generic Algorithm

In this section, we present an $O(1)$ mutual exclusion algorithm that uses a generic fetch-and-$\phi$ 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. 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 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 4. 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.)

---

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.
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.

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,” \(\text{Tail}[j]\). 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 \(\text{CurrentQueue}\). 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 begins its entry section, it determines which queue is the current queue by reading the variable \(\text{CurrentQueue}\) (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 \((p, \text{prev} \neq \bot)\), then it waits until its predecessor in
Queue 0: the current queue
Queue 1: the old queue

(a)

New processes may enter the current queue.

Tail[0]

processes that have finished execution

head

waiting processes

Queue 0: the current queue
Queue 1: the old queue

(b)

(no active processes)

Queue 0: the current queue
Queue 1: the old queue

(c)

(empty: Tail[1] = ⊥)

Queue 0: the current queue
Queue 1: the old queue

(d)

Wait for process q to finish.

Process q

Position q

Process q

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.

the queue updates the spin variable \( Signal[p.idx][p.prev] \) (statement 6), which \( p \) then resets (statement 7).

Note that 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 \( \text{Entry}_2 \) and \( \text{Exit}_2 \), 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].

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 \( \text{Position}[j] \). 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 \( \text{Position}[0] \) equals 2. A process in queue \( j \) updates \( \text{Position}[j] \) while still effectively in its critical section (statements 10 and 11). Thus, \( \text{Position}[j] \) 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

\[5\] We use \( s.p \) to denote the statement with label \( s \) of process \( p \), and \( p.v \) to represent \( p \)'s private variable \( v \).
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 be yet to update 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 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 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 \( Signal[1]\ldots\). Finally, statement 20 exchanges the two queues.

We still must show that using a fetch-and-\( \phi \) 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).

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

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-$\phi$ 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. \hfill $\square$

### 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 4. 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 4), and each statement of the form "$B := true$" by the code fragment on the right (see statements 4–8, 40–44, and 45–49).

```plaintext
a: Entry2($J$, 0);
b: flag := B;
c: Waiter[$J$] := if flag then ⊥ else p;
d: Spin[$p$] := false;
e: Exit2($J$, 0);
f: if ¬flag then
  g: await Spin[$p$];
h: Waiter[$J$] := ⊥
  fi
i: Entry2($J$, 1);
j: $B := true$;
k: next := Waiter[$J$];
l: Exit2($J$, 1);
m: if next ≠ ⊥ then Spin[next] := true
```

The variable $\text{Waiter}[J]$ is assumed to be initially $\bot$, and $\text{Spin}[p]$ is a spin variable used exclusively by process $p$ (and, hence, it can be stored in memory local to $p$). $\text{Entry}_2$ and $\text{Exit}_2$ represent an instance of a two-process mutual exclusion algorithm, indexed by $J$. 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 $B = false$ holds before the execution of b–d, then $p$ assigns $\text{Waiter}[J] := p$ at line c, and initializes its spin variable at line d. Process $q$ subsequently reads $\text{Waiter}[J] = p$ at line k, and establishes $\text{Spin}[p] = true$ 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 $\text{Waiter}[J] = \bot$ (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 $\text{Waiter}[J] = \bot$, 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 4.)

```plaintext
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 $i = p$.idx and $x = p$.self. By Property 3, $p$ cannot possibly have a successor. Moreover, a process may read $\text{Signal}[i][x]$ (at statement 6) only if its predecessor has written (or will eventually write) $\text{Signal}[i][x] := true$ by executing statement 21. Hence, in Algorithm G-CC, the value of $\text{Signal}[i][x]$ written
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

Entry2(p, 1);

if q̸= ⊥ then Spin[q] := true fi;

q := Waiter1[p];

Exit2(p, 1);

if q̸= ⊥ then Spin[q] := true fi;

wait Spin[p];

Waiter2[1][q] := ⊥ fi;

Signal[idx][prev] := false fi;

Entry2(idx)

Critical Section;

pos := Position[idx];

Position[idx] := pos + 1;

Exit2(idx);

if (pos < N) ∧ (pos ̸= p) then

q := pos;

Entry2(q, 0);

flag := ¬Active[q] ∨ (QueueIdx[q] = idx);

Waiter1[q] :=

if flag then ⊥ else p;

Spin[p] := false;

Exit2(q, 0);

if ¬flag then

await Spin[p];

Waiter1[q] := ⊥ fi

else pos = N then

temp := Tail[1 − idx];

Signal[1 − idx][temp] := false;

Tail[1 − idx] := ⊥;

Position[1 − idx] := 0;

CurrentQueue := 1 − idx fi;

if pos < 2N − 1 then

Entry2((idx, self), 1);

Signal[idx][self] := true;

next := Waiter2[idx][self];

Exit2((idx, self), 1);

if next ̸= ⊥ then Spin[next] := true fi fi;

Entry2(p, 1);

Active[p] := false;

next := Waiter1[p];

Exit2(p, 1);

if next ̸= ⊥ then Spin[next] := true fi

od

Figure 4: Algorithm G-DSM: Generic fetch-and-∅-based mutual exclusion algorithm for DSM machines. Statements different from Figure 2 are shown with boldface line numbers.
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 \( \text{Signal}[i][x] \) by executing statement 17.

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

We now argue that the various two-process mutual exclusion algorithms added to Figure 4 are safe by considering each pair of \text{Entry/Exit} calls in this figure in turn. (The arguments below are also presented more formally in Appendix B.)

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

\textit{Statements 10–14:} Because the \text{fetch-and-}\phi \text{ primitive used in } \text{Algorithm G-CC}' \text{ has rank at least } 2N, \text{ by Property 3, each process that executes statements } 5–7 \text{ of } \text{Algorithm G-CC}' \text{ within a particular queue must have a distinct value of } \text{prev}. \text{ This implies that the } \text{Entry/Exit} \text{ pair at statements } 10–15 \text{ in Figure 4 is safe.}

\textit{Statements 25–30:} Since the queues are switched after a process at position \( N \) executes its critical section, statements 14 and 15 of \text{Algorithm G-CC}' \text{ may be executed only by processes in the current queue. Moreover, since every process that executes within the current queue has a distinct position (between 0 and } 2N–1), every such process that executes statements 14 and 15 does so with a distinct value of \( \text{pos} \). \text{ This implies that the } \text{Entry/Exit} \text{ pair at statements } 25–30 \text{ in Figure 4 is safe.}

\textit{Statements 40–43:} Because the \text{fetch-and-}\phi \text{ primitive has rank at least } 2N, \text{ each process at queue } i \text{ (for some } i) \text{ whose position is between 0 and } 2N–2 \text{ (inclusive) writes a distinct value to } \text{Tail}[i] \text{ at statement } 6 \text{ of } \text{Algorithm G-CC}', \text{ and hence, has a distinct value of } \text{self}. \text{ This implies that the } \text{Entry/Exit} \text{ pair at statements } 40–43 \text{ in Figure 4 is safe.}

Note that, if process \( p \) is at position \( 2N–1 \) of queue \( i \), then \( p \) is allowed to write any value to \( \text{Tail}[i] \), by the definition of rank. (By Property 3, \( p \) cannot have a successor.) Hence, in \text{Algorithm G-CC}, \( p \)’s execution of statement 21 may write the same element of \text{Signal} as some other process residing at an earlier position. In particular, in \text{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 \text{Signal}.

This does not pose a problem for \text{Algorithm G-CC}, because 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 \textit{would} be a problem if we were to implement statement 21 with statements \( i–m \) as in \text{Algorithm G-DSM}. This is why we introduce an additional "if" clause in \text{Algorithm G-CC}'.

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 [8] to a variant that locally spins on DSM machines.\(^7\) Given the correctness of the above transformation, we have the following.

\textbf{Lemma 2} \textit{If the underlying fetch-and-}\phi \textit{ primitive has rank at least } 2N, \textit{ then Algorithm G-DSM is a correct, starvation-free mutual exclusion algorithm with } O(1) \textit{ RMR time complexity in DSM machines.} \hfill \Box

\(^6\)In the conference version of this paper [4], we presented a DSM algorithm as a direct transformation of \text{Algorithm G-CC}. We later discovered that an additional "if" clause is necessary, as explained here.

\(^7\)In [4], we stated that the above transformation could also be applied to the algorithm of T. Anderson [6]. 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 \text{fetch-and-increment} primitive is exploited to avoid the \text{Entry} and \text{Exit} calls.
If we have a fetch-and-\(\phi\) 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 \([r/2]\)-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-\(\phi\) primitive [13].)

**Theorem 1** Using any fetch-and-\(\phi\) 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 [2]. The adaptive algorithm utilizes two trees, called renaming and overflow trees, respectively, as illustrated in Figure 5. 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. 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 [2], if a process experiences point contention\(^8\) \(k\), 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 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-\(\phi\)-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-\(\phi\) 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.

**4 Concluding Remarks**

We have presented a ranking of fetch-and-\(\phi\) 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-\(\phi\) primitive of rank \(r\)

---

\(^8\)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].
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 [10] that $\Theta(\max(1, \log_r N))$ is not optimal for certain “self-resettable” 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-$\phi$ 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 [9], 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 [3]). 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


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. 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 a statement of the form "await B" is enabled for execution only if the boolean condition `B` is true, and when executed, causes a single transition to the next executable statement. Our proof also makes use of a number of auxiliary variables. In Figure 6, 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.position` keeps track of `p`’s position in the queue; as shown by invariant (149) below, when `p` is in its exit section, `p.position` equals the non-auxiliary variable `p.pos`.

For `i = 0` or `1`, `HistLen[i]`, `Hist[i][0..∞]`, and `Param[i][0..∞]` 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 `Hist[i][...]` and `Param[i][...]` are actually used.)

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

\[
\begin{align*}
\text{Tail}[i] &= \bot; \\
\text{HistLen}[i] &= 0; \\
\text{Hist}[i] &= (\bot, \bot, \bot, \ldots); \\
\text{Param}[i] &= (\bot, \bot, \bot, \ldots).
\end{align*}
\]

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

\[
\begin{align*}
\text{Tail}[i] &= (p, 1); \\
\text{HistLen}[i] &= 4; \\
\text{Hist}[i] &= (\bot, (p, 0), (q, 1), (r, 1), (p, 1), \bot, \bot, \ldots); \\
\text{Param}[i] &= ((p, 0), (q, 3), (r, 5), (p, 1), \bot, \bot, \ldots).
\end{align*}
\]

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

\[
\begin{align*}
\text{Tail}[i] &= 4; \\
\text{Hist}[i] &= (\bot (=0), 1, 2, 3, 4, \bot, \bot, \ldots).
\end{align*}
\]

**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..∞`; `x` and `y` range over `Vartype`; `p`, `q`, and `r` range over `0..N - 1`.

\[
\begin{align*}
\text{invariant (Exclusion)} &\quad \left|\{ p :: p@\{9..12\}\}\right| \leq 1 \quad (I1) \\
\text{invariant} &\quad \left|\{ p :: p@\{7..21\} \land p.idx = i\}\right| \leq 1 \quad (I2) \\
\text{invariant} &\quad p@\{6..22\} \land p.idx = i \land p.position = h \Rightarrow
\end{align*}
\]

14
shared variables

\begin{align*}
& \text{CurrentQueue: } 0..1; \\
& \text{Tail: array}[0..1] \text{ of Vartype initially } \bot; \\
& \text{Position: array}[0..1] \text{ of } 0.2N - 1 \text{ initially } 0; \\
& \text{Signal: array}[0..1][\text{Vartype}] \text{ of boolean initially } false; \\
& \text{Active: array}[0..N - 1] \text{ of boolean initially } false; \\
& \text{QueueIdx: array}[0..N - 1] \text{ of } (\bot, 0..1)
\end{align*}

private variables

\begin{align*}
& \text{idx: } 0..1; \\
& \text{counter: integer}; \\
& \text{prev, self, tail: Vartype}; \\
& \text{pos: } 0.2N - 1
\end{align*}

private auxiliary variable

position: 0..∞

type

\begin{verbatim}
ParamType = record
  proc: 0..N - 1; \\
  counter: integer
end
\end{verbatim}

shared auxiliary variables

\begin{align*}
& \text{HistLen: array}[0..1] \text{ of } 0..\infty \text{ initially } 0; \\
& \text{Hist: array}[0..1][0..\infty] \text{ of Vartype initially } \bot; \\
& \text{Param: array}[0..1][0..\infty] \text{ of ParamType initially } \bot
\end{align*}

process \( p :: \) /* 0 \leq p < N */

while true do

0: Noncritical Section;
1: \text{QueueIdx}[p] := \bot; \\
2: \text{Active}[p] := true; \\
3: \text{idx} := \text{CurrentQueue}; \\
4: \text{QueueIdx}[p] := \text{idx}; \\
5: /* atomically execute lines 5a–5h */
5a: \text{prev} := \text{Tail}[\text{idx}]; \\
5b: \text{self} := \phi(\text{prev}, \alpha_p[\text{counter}]); \\
5c: \text{Tail}[\text{idx}] := \text{self}; \\
5d: \text{position} := \text{HistLen}[\text{idx}]; \\
5e: \text{Param}[\text{idx}][\text{position}] := (\text{p}, \text{counter}); \\
5f: \text{Hist}[\text{idx}][\text{position} + 1] := \text{self}; \\
5g: \text{HistLen}[\text{idx}] := \text{position} + 1; \\
5h: \text{counter} := \text{counter} + 1;

if \text{prev} \neq \bot then
6: \text{await Signal}[\text{idx}][\text{prev}]; \\
7: \text{Signal}[\text{idx}][\text{prev}] := false
fi
8: \text{Entry}_2(\text{idx});
9: Critical Section;
10: \text{pos} := \text{Position}[\text{idx}]; \\
11: \text{Position}[\text{idx}] := \text{pos} + 1; \\
12: \text{Exit}_2(\text{idx}); \\
13: \text{if } (\text{pos} < N) \land (\text{pos} \neq p) \text{ then}
14: \text{await } \neg \text{Active}[\text{pos}] \lor \\
15: (\text{QueueIdx}[\text{pos}] = \text{idx})
\text{elseif } \text{pos} = N \text{ then}
16: \text{tail} := \text{Tail}[1 - \text{idx}]; \\
17: \text{Signal}[1 - \text{idx}][\text{tail}] := false;
18: /* atomically execute lines 18a–18d */
18a: \text{Tail}[1 - \text{idx}] := \bot; \\
18b: \text{HistLen}[1 - \text{idx}] := 0; \\
18c: \text{forall } j \text{ do Param}[1 - \text{idx}][j] := \bot \od; \\
18d: \text{forall } j \text{ do Hist}[1 - \text{idx}][j] := \bot \od;
19: \text{Position}[1 - \text{idx}] := 0; \\
20: \text{CurrentQueue} := 1 - \text{idx}
fi
21: \text{Signal}[\text{idx}][\text{self}] := true; \\
22: \text{Active}[p] := false
\end{verbatim}

Figure 6: Algorithm G-CC with auxiliary variables added.
\[\text{HistLen}[i] > h \land p.\text{prev} = \text{Hist}[i][h] \land p.\text{self} = \text{Hist}[i][h+1] \land \text{Param}[i][h].\text{proc} = p\] (I3)

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

invariant \(p@\{8.21\} \land p.\text{idx} = i \Rightarrow (\forall x :: \text{Signal}[i][x] = \text{false})\) (I5)

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

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

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

invariant \(\{p :: p@\{6.7\} \land p.\text{idx} = i \land p.\text{prev} = x\} \leq 1\) (I9)

invariant \(\text{Position}[i] = q + 1 \Rightarrow q@\{0..3\} \lor q.\text{idx} = i \lor (\exists p :: p@\{12.20\} \land p.\text{idx} = i)\) (I10)

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

invariant \(p@\{11.13\} \land p.\text{idx} = i \land p.\text{pos} = N \Rightarrow \text{CurrentQueue} = i \land (\forall q :: q@\{0..3\} \lor q.\text{idx} = i)\) (I12)

invariant \(p@\{16.20\} \land p.\text{idx} = i \Rightarrow \text{CurrentQueue} = i \land (\forall q :: q@\{0..3\} \lor q.\text{idx} = i)\) (I13)

invariant \(p@\{11\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] = p.\text{pos}\) (I14)

invariant \(p@\{12.21\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] = p.\text{pos} + 1\) (I15)

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

invariant \(\text{Tail}[i] = \text{Hist}[i][\text{HistLen}[i]]\) (I17)

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

invariant \(0 \leq j < k < \text{HistLen}[i] \land \text{Param}[i][j] = (p, c_1) \land \text{Param}[i][k] = (p, c_2) \land (\forall l :: 0 \leq l < k :: \text{Param}[i][l].\text{proc} = p) \Rightarrow c_2 = c_1 + 1\) (I19)

invariant \(0 \leq j < \text{HistLen}[i] \land \text{Param}[i][j] = (p, c) \land (\forall k :: j < k < \text{HistLen}[i] :: \text{Param}[i][k].\text{proc} = p) \Rightarrow (p.\text{counter} = c + 1 \land p@\{4..22\} \land p.\text{idx} = i) \lor (p.\text{counter} = c + 1 \land p@\{0..3\}) \lor (p@\{0..3\} \land \text{CurrentQueue} = 1 - i) \lor (p@\{4..22\} \land p.\text{idx} = \text{CurrentQueue} = 1 - i)\) (I20)

invariant \(0 \leq \text{HistLen}[i] \leq 2N\) (I21)

invariant \(0 \leq \text{Position}[i] \leq 2N\) (I22)

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

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

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

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

invariant \(p@\{12.21\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] = p.\text{position} + 1\) (I27)

invariant \(\left[\text{HistLen}[i] - \text{Position}[i] = \left\{p :: p@\{6..11\} \land p.\text{idx} = i\right\}\right] \lor (\exists p :: p@\{19\} \land p.\text{idx} = 1 - i)\) (I28)

invariant \(\left\{p :: p@\{6..22\} \land p.\text{idx} = i \land p.\text{position} = h\right\} \leq 1\) (I29)

invariant \(\text{CurrentQueue} = i \Rightarrow \left\{p :: p@\{4..5\} \land p.\text{idx} = 1 - i\right\} \leq 2N - \text{HistLen}[1 - i]\) (I30)

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

invariant \(\text{Position}[i] > N + 1 \Rightarrow \text{CurrentQueue} = 1 - i\) (I32)
invariant $Active[p] = p@[3..22]$ (I33)

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

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

invariant $Position[i] \leq N \Rightarrow$

$\text{CurrentQueue} = i \lor$

$(\exists p :: p@[20] \land p.idx = 1 - i) \lor$

$(\text{Position}[i] = 0 \land (\forall q :: q@[0..3] \land q.idx = 1 - i))$ (I36)

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

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

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

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

invariant $\text{Position}[i] \leq h < \text{HistLen}[i] \Rightarrow$

$(\exists p :: p@[6..11] \land p.idx = i \land p.position = h)$ (I41)

invariant $\text{Position}[i] = h > 0 \Rightarrow$

$Signal[i][\text{Hist}[i][h]] = \text{true} \lor$

$(\exists p :: p@[12..21] \land p.idx = i \land p.position = h - 1) \lor$

$(\exists q :: q@[8..11] \land q.idx = i \land q.position = h)$ (I42)

invariant $0 < j < k \leq \text{HistLen}[i] \land k < 2N \land$

$\text{Param}[i][j - 1].proc \neq \text{Param}[i][k - 1].proc \Rightarrow$

$\text{Hist}[i][j] \neq \text{Hist}[i][k]$ (I43)

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

invariant $0 < j < k \leq \text{HistLen}[i] \land k < 2N \land$

$\text{Param}[i][j - 1] = (p, c) \land \text{Param}[i][k - 1] = (p, c + 1) \Rightarrow$

$\text{Hist}[i][j] \neq \text{Hist}[i][k]$ (I45)

invariant $\text{Position}[i] \leq h < \text{HistLen}[i] \land \text{Param}[i][h].proc = p \Rightarrow$

$p@[6..11] \land p.idx = i \land p.position = h$ (I46)

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

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

invariant $p@[11..22] \Rightarrow p.pos = p.position$ (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) $|\{p :: p@[9..12]\}| \leq 1$ (I1)

Proof: Since the Entry and Exit routines (statements 8 and 12) are assumed to be correct, (I1) follows easily from (I2).

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

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

$p.idx = i \land Tail[i] = \perp \land q@[7..21] \land q.idx = i$
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), \( p@\{5\} \land p.idx = i \land \text{Tail}[i] = ⊥ \) implies \( \text{HistLen}[i] = 0 \). However, by applying (I3) with ‘\( p' \leftarrow q \), we have \( \text{HistLen}[i] > \text{position} \geq 0 \), a contradiction. Thus, statement 5., \( p \) cannot falsify (I2).

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

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

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

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

The only statements that may falsify the consequent are 5., and 18., where \( q \) is any arbitrary process. As shown above, statement 5., \( p \) cannot falsify (I3). For \( q \neq p \), if statement 5., \( p \) is executed when \( \text{HistLen}[i] > h \) holds, then it preserves \( \text{HistLen}[i] > h \), and does not update \( \text{Hist}[i][h] \), \( \text{Hist}[i][h + 1] \), or \( \text{Param}[i][h] \). Thus, statement 5., \( p \) 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 \( p@\{0..3\} \land p.idx = 1 - i \). Thus, in this case, the antecedent is false before and after the execution of 18., \( q \).

\[
\text{Proof:} \quad \text{Invariant (14) follows easily by applying (I44) with ‘\( j' \leftarrow \text{HistLen}[i], \text{ and using (I17).}
\]

\[
\text{Proof:} \quad \text{The only statements that may establish the antecedent are } 5., \text{ and } 7., \text{. Statement } 5., \text{ may establish the antecedent only if executed when } \text{HistLen}[i] = 0. \text{ Hence, by (I23), the consequent is true before and after the execution of } 5., \text{.}
\]

Statement 7., \( p \) may establish the antecedent only if executed when \( p@\{7\} \land p.idx = i \) holds. In this case, by (I7) and (I8), \( \text{Signal}[i][p.prev] \) is the only entry among \( \text{Signal}[i][\ldots] \) that is \text{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 \( p@\{8..21\} \land p.idx = i \land q@\{21\} \land q.idx = i \) holds, which implies \( p = q \) by (I2). Thus, 21., \( q \) falsifies the antecedent in this case.

\[
\text{Proof:} \quad \text{The only statement that may establish the antecedent is } 21., \text{ which may do so only if } \text{HistLen}[i] = 0. \text{ In this case, before } 21., \text{ is executed, } \text{Position}[i] = \text{position} + 1 \text{ and } p.self = \text{Hist}[i][\text{position} + 1] \text{ hold, by (I27) and (I3), respectively. Thus, the consequent is true before and after the execution of } 21., \text{.}
\]

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

\[
p@\{5\} \land p.idx = i
\]

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

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

\[ \text{invariant } \forall i : \text{Signal}[i][x] = \text{Signal}[i][y] = \text{true} \Rightarrow x = y \tag{17} \]

\[ \text{Proof: } \text{This invariant follows trivially from (I6).} \tag{\mathbin{\Box}} \]

\[ \text{invariant } p@\{7\} \land p.idx = i \Rightarrow \text{Signal}[i][p.prev] = \text{true} \tag{18} \]

\[ \text{Proof: } \text{The only statement that may establish the antecedent is 6.p, which may do so only if the consequent holds.} \]

The only statements that may falsify the consequent are 7.q and 17.q, where \( q \) is any arbitrary process. Statement 7.q may potentially falsify (I8) only if executed when \( p@\{7\} \land p.idx = i \land q@\{7\} \land q.idx = i \) 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 \( q.idx = 1 - i \) holds, which implies that \( p@\{0..3\} \lor p.idx = 1 - i \) holds, by (I13). Thus, the antecedent is false before and after the execution of 17.q.

\[ \text{invariant } \left| \{ p :: p@\{6,7\} \land p.idx = i \land p.prev = x \} \right| \leq 1 \tag{19} \]

\[ \text{Proof: } \text{Assume the following.} \]

\[ p@\{6,7\} \land q@\{6,7\} \land p.idx = q.idx = i \land p.prev = q.prev = x. \]

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

Combining these assertions, by (I48), we have \( j = k \). By (I29), this implies \( p = q \). \tag{\mathbin{\Box}}

\[ \text{invariant } \text{Position}[i] = q + 1 \Rightarrow \]

\[ q@\{0..3\} \lor q.idx = i \lor (\exists p :: p@\{12..20\} \land p.idx = i) \tag{110} \]

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

The only statement that may falsify \( q@\{0..3\} \lor q.idx = i \) is 3.q, which may do so only if executed when \( \text{CurrentQueue} = 1 - i \) holds, which in turn implies that \( \text{Position}[i] = 0 \lor \text{Position}[i] > N \) 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 \( p@\{12..20\} \land p.idx = i \) 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 \( \text{Active}[q] = false \) 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 \( p@\{12..20\} \land p.idx = i \) only if executed when \( \text{QueueIdx}[q] = i \) holds. By (I34) and (I35), this implies \( q@\{0,1\} \lor q.idx = i \), and hence the consequent of (I10) holds after its execution.

\[ \text{invariant } q + 1 < \text{Position}[i] \leq N \Rightarrow q@\{0..3\} \lor q.idx = i \tag{111} \]
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\} ∨ p.idx = i, before the execution of 11.r. However, since we have r@\{11\} ∨ 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 ≠ i holds. In this case, by (I37), the antecedent is false before and after the execution of 3.q.

\[
\text{invariant } p@\{11..13\} ∨ p.idx = i ∨ p.pos = N \Rightarrow \text{CurrentQueue = i } \land (\forall q :: q@\{0..3\} ∨ q.idx = i)
\] (I12)

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

\[
p@\{10\} ∨ p.idx = i \land \text{Position[i] = N}
\] (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\} ∨ q.idx = i) holds after the execution of 10.p, we consider two cases, depending on the value of q. If 0 ≤ q < N − 1, then by (I11), q@\{0..3\} ∨ 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\} ∨ q.idx = i holds, or there exists a process r (different from p) satisfying r@\{12..20\} ∨ 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\} ∨ 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.

\[
\text{invariant } p@\{16..20\} ∨ p.idx = i \Rightarrow \text{CurrentQueue = i } \land (\forall q :: q@\{0..3\} ∨ q.idx = i)
\] (I13)

Proof: The only statement that may establish the antecedent is 13.p, which may do so only if executed when p.idx = i ∧ 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).

\[
\text{invariant } p@\{11\} ∨ p.idx = i \Rightarrow \text{Position[i] = p.pos}
\] (I14)

\[
\text{invariant } p@\{12..21\} ∨ p.idx = i \Rightarrow \text{Position[i] = p.pos + 1}
\] (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 ∧ q.pos = N holds. By (I13), this implies that p@\{0..3\} ∨ p.idx = 1 − i holds. Hence, the antecedents of (I14) and (I15) are false before and after the execution of 19.q.

\[
\text{invariant } \text{HistLen}[i] > h ∧ \text{Param[i][h]} = (p, c) \Rightarrow \text{Hist[i][h + 1] = } \phi(\text{Hist[i][h]}, a_p[c])
\] (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 HistLen[1−q.idx] := 0, and hence cannot establish the antecedent.)

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

\[ Hist[i][h + 1] = q.self = φ(Hist[i][h], α_q[c']) \quad \text{and} \quad \text{Param}[i][h] = (q, c'), \]

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 \( q.idx = i \land (\text{HistLen}[i] = h - 1 \lor \text{HistLen}[i] = h) \) holds. If \( \text{HistLen}[i] = h - 1 \) holds before its execution, then \( \text{HistLen}[i] = h \) holds after its execution, and hence the antecedent is false. On the other hand, if \( \text{HistLen}[i] = h \) 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 \( \text{HistLen}[i] = 0 \), and hence the antecedent is false after its execution.

\[ \text{invariant} \quad \text{Tail}[i] = \text{Hist}[i][\text{HistLen}[i]] \quad \text{(I17)} \]
\[ \text{invariant} \quad \text{Hist}[i][0] = ⊥ \quad \quad \text{(I18)} \]

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 \( \text{HistLen}[i] \) is always nonnegative (by (I21)), line 5f cannot update \( \text{Hist}[i][0] \), and hence (I18) is maintained.

\[ \text{invariant} \quad 0 \leq j < k < \text{HistLen}[i] \land \]
\[ \text{Param}[i][j] = (p, c_1) \land \text{Param}[i][k] = (p, c_2) \land \]
\[ (\forall l : j < l < k : \text{Param}[i][l].proc \neq p) \Rightarrow \]
\[ c_2 = c_1 + 1 \quad \quad \text{(I19)} \]

Proof: The only statement that may establish the antecedent is 5.q, where \( q \) is any arbitrary process. (Note that statement 18.q assigns HistLen[1−1.idx] := 0 and does not update any entry of Param.) Since 5.q increments HistLen[q.idx] by one, it may establish the antecedent only if executed when \( q.idx = i \land \text{HistLen}[i] = k \) holds. Note that 5.q establishes \( \text{Param}[i][k] = (q, c) \) 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.

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

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

\[ \text{invariant} \quad 0 \leq j < \text{HistLen}[i] \land \]
\[ \text{Param}[i][j] = (p, c) \land \]
\[ (\forall k : j < k < \text{HistLen}[i] : \text{Param}[i][k].proc \neq p) \Rightarrow \]
\[ (p.counter = c + 1 \land p@\{4..22\} \land p.idx = i) \lor \]
\[ (p.counter = c + 1 \land p@\{0..3\}) \lor \]
\[ (p@\{0..3\} \land \text{CurrentQueue} = 1 - i) \lor \]
\[ (p@\{4..22\} \land p.idx = \text{CurrentQueue} = 1 - i) \]
\[ A \quad B \quad C \quad D \quad \text{(I20)} \]

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 HistLen[q.idx] by one, it may establish the
antecedent only if executed when \( q\.idx = i \land \text{HistLen}[i] = j \) holds. Note that \( 5.q \) establishes \( \text{Param}[i][k] = (q, c') \), where \( c' \) is the value of \( q\.counter \) before the execution of \( 5.q \). Thus, the antecedent may be established only if \( q = p \land c' = c \) holds. It follows that statement \( 5.q \) establishes disjunct \( A \) in this case.

Disjunct \( A \) 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 \( \text{Param}[i][\text{HistLen}[i] - 1] = (p, c+1) \), and hence falsifies the last conjunct of the antecedent. Statement \( 22.p \) establishes disjunct \( B \) if executed when disjunct \( A \) holds.

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

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

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

\[ \text{invariant } 0 \leq \text{HistLen}[i] \leq 2N \] (I21)

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

\[ \text{invariant } 0 \leq \text{Position}[i] \leq 2N \] (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 \( \text{Position}[i] = p\.pos + 1 = p\.position + 1 \). Moreover, by (I3) and (I21), \( p\.position < \text{HistLen}[i] \leq 2N \) holds before \( 11.p \) is executed. Thus, statement \( 11.p \) preserves (I22). □

\[ \text{invariant } \text{HistLen}[i] = 0 \Rightarrow (\forall x :: \text{Signal}[i][x] = false) \] (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 \( p@[18] \land p\.idx = 1 - i \) 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), \( \text{HistLen}[i] > p\.position \) holds before and after \( 21.p \) is executed. Thus, the antecedent is false before and after the execution of \( 21.p \). □

\[ \text{invariant } p@[18] \land p\.idx = i \Rightarrow (\forall x :: \text{Signal}[1 - i][x] = false) \] (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 \( \text{Signal}[1 - i][x] = true \) holds for some \( x \) before the execution of \( 17.p \). It suffices to show \( x = p\.tail \).

By (I6), we have
\[ x = \text{Hist}[1 - i][\text{Position}[1 - i]]. \] (3)

Also, by (I13), we have
\[ (\forall q :: q@[0..3] \lor q\.idx = i). \] (4)

Moreover, by (I2), \( p@[17] \land p\.idx = i \) implies
\[ \neg(\exists r :: r@[19] \land r\.idx = i). \] (5)
Combining (4) and (5), and applying (I28) with \( i' \leftarrow 1 - i \), we have \( \text{HistLen}[1 - i] = \text{Position}[1 - i] \). Hence, by (3) and (117), we have \( x = \text{Hist}[1 - i] \text{[HistLen}[1 - i]] = \text{Tail}[1 - i] \).

Thus, by (I25), \( x = p.\text{tail} \) 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 \( q.\text{idx} = 1 - i \) holds. However, when the antecedent holds, \( q@\{21\} \land q.\text{idx} = 1 - i \) is false, by (I13).

\[
\text{invariant } p@\{17\} \land p.\text{idx} = i \Rightarrow p.\text{tail} = \text{Tail}[1 - i] \tag{I25}
\]

**Proof:** The only statement that may establish the antecedent is \( 16.p \), which may do so only if \( p.\text{idx} = i \) 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 \( q.\text{idx} = 1 - i \) holds, which implies that the antecedent is false, by (I13). Similarly, statement \( 18.q \) may falsify the consequent only if executed when \( q.\text{idx} = i \) holds, which implies that the antecedent is false, by (I2).

\[
\text{invariant } p@\{7..11\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] = p.\text{position} \tag{I26}
\]

**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

\[
p@\{5\} \land \text{Tail}[i] = \bot \land p.\text{idx} = i \tag{6}
\]

holds. In this case, by (I4) and (I38), we have \( \text{HistLen}[i] = 0 \). Thus, statement \( 5.p \) establishes \( p.\text{position} = 0 \). By (I28), \( \text{HistLen}[i] = 0 \) also implies that either \( \text{Position}[i] = 0 \) or \((\exists q : q@\{19\} \land q.\text{idx} = 1 - i)\) 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 \( p@\{0..3\} \lor p.\text{idx} = 1 - i \), which contradicts (6). It follows that the latter case in fact cannot arise.

Statement \( 6.p \) may establish the antecedent only if executed when

\[
p@\{6\} \land p.\text{idx} = i \land \text{Signal}[i][p.\text{prev}] = \text{true} \tag{7}
\]

holds. By (I6), this implies \( p.\text{prev} = \text{Hist}[i][\text{Position}[i]] \). Let \( k = p.\text{position} \). By (I3) (with \( 'h' \leftarrow k \)) and (I47), we have \( p.\text{prev} = \text{Hist}[i][k] \) and \( \text{Position}[i] \leq k < \text{HistLen}[i] \). If \( k > \text{Position}[i] \), then by applying (I48) with \( 'j' \leftarrow \text{Position}[i] \), we have \( \text{Hist}[i][\text{Position}[i]] \neq \text{Hist}[i][k] \), a contradiction. It follows that \( \text{Position}[i] = k = p.\text{position} \) 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 \( q.\text{idx} = i \) 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 \( q.\text{idx} = 1 - i \) holds. By applying (I13) with \( 'p' \leftarrow q \) and \( 'i' \leftarrow 1 - i \), this implies \( p@\{0..3\} \lor p.\text{idx} = 1 - i \). Hence, the antecedent is false before and after the execution of \( 19.q \).

\[
\text{invariant } p@\{12..21\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] = p.\text{position} + 1 \tag{I27}
\]

**Proof:** The only statement that may establish the antecedent is \( 11.p \), which may do so only if \( p.\text{idx} = i \) 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 \( q@\{11\} \land q.\text{idx} = i \) 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).
\[ \text{invariant } \left[ \text{HistLen}[i] - \text{Position}[i] = \left| \left\{ p : p@\{6..11\} \land p.idx = i \right\} \right| \right] \lor
\left( \exists p : p@\{19\} \land p.idx = 1 - i \right) \]

\hspace{1cm} (I28)

**Proof:** Define

\[ X = \left| \left\{ p : p@\{6..11\} \land p.idx = i \right\} \right| . \]

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

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

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

\[ \text{invariant } \left| \left\{ p : p@\{6..22\} \land p.idx = i \land p.position = h \right\} \right| \leq 1 \]

(I29)

**Proof:** Assume that there exists a process \(p\) satisfying \(p@\{6..22\} \land p.idx = i \land p.position = h\). By (I3), this implies \text{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 \text{HistLen}[i] > h, statement \(5.q\) establishes \(q.position > h\), and hence cannot increase the left-hand side of (I29).

\[ \text{invariant } \text{CurrentQueue} = i \Rightarrow
\left| \left\{ p : p@\{4, 5\} \land p.idx = 1 - i \right\} \right| \leq 2N - \text{HistLen}[1 - i] \]

(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

\[ \left| \left\{ p : p@\{6..11\} \land p.idx = 1 - i \right\} \right| \]

prior to the execution of \(20.q\). Statement \(20.q\) may establish the antecedent only if executed when

\[ q@\{20\} \land q.idx = 1 - i \]

holds, which also implies the following.

\[ \left| \left\{ p : p@\{6..11, 20\} \land p.idx = 1 - i \right\} \right| \geq X + 1 \]

By (I13) with ‘\(p’ ← q and ‘\(i’ ← 1 - i\), \(\neg(\exists r : r@\{19\} \land r.idx = i)\) holds, and hence, by (I28), we have

\[ \text{HistLen}[1 - i] - \text{Position}[1 - i] = X. \]

Also, since \(20.q\) may be executed only if \(q.pos = N\) holds, by applying (I15) with ‘\(p’ ← q and ‘\(i’ ← 1 - i\), we have

\[ \text{Position}[1 - i] = N + 1. \]

Combining these assertions, we have the following.

\[ \left| \left\{ p : p@\{4, 5\} \land p.idx = 1 - i \right\} \right| \leq N - X - 1 \]
\[ = N - (\text{HistLen}[1 - i] - \text{Position}[1 - i]) - 1 \]
\[ = N - (\text{HistLen}[1 - i] - N - 1) - 1 \]
\[ = 2N - \text{HistLen}[1 - i]. \]

24
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 $HistLen[1 - i]$), 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 $HistLen[1 - i]$ (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.

\textbf{Invariant} \quad Position[i] = N + 1 \Rightarrow \\
\quad CurrentQueue = 1 - i \vee \\
\quad (\exists p : p@\{12..20\} \land p.idx = i \land p.pos = N) \quad (I31)

\textbf{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 $p@\{12..20\} \land p.idx = i \land p.pos = N$), 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.

\textbf{Invariant} \quad Position[i] > N + 1 \Rightarrow \quad CurrentQueue = 1 - i \quad (I32)

\textbf{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 $p@\{11\} \land p.idx = i \land Position[i] = N + 1$ 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.

\textbf{Invariant} \quad Active[p] = p@\{3..22\} \quad (I33)

\textbf{Invariant} \quad p@\{5..22\} \land p.idx = i \Rightarrow QueueIdx[p] = i \quad (I34)

\textbf{Invariant} \quad p@\{2.4\} \Rightarrow QueueIdx[p] = \bot \quad (I35)

\textbf{Proof:} These invariants follow trivially from inspecting Algorithm G-CC.

\textbf{Invariant} \quad Position[i] \leq N \Rightarrow \\
\quad CurrentQueue = i \vee \\
\quad (\exists p : p@\{20\} \land p.idx = 1 - i) \vee \\
\quad (Position[i] = 0 \land (\forall q : q@\{0..3\} \land q.idx = 1 - i)) \quad A \quad B \quad C \quad (I36)

\textbf{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 $B$.

The only statement that may falsify disjunct $A$ is 20.r, where $r$ is any arbitrary process. Statement 20.r may falsify disjunct $A$ only if executed when $r.idx = i \land r.pos = N$ 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 $B$ is 20.p, which establishes disjunct $A$.

The only statements that may falsify disjunct $C$ are 3.q and 11.q, where $q$ is any arbitrary process. Statement 3.q may falsify disjunct $C$ only if executed when $CurrentQueue = i$ holds, in which case disjunct $A$ holds before and after its execution. Statement 11.q may falsify disjunct $C$ only if executed when $q@\{11\} \land q.idx = i$ holds, which is precluded when disjunct $C$ holds.
invariant $1 \leq \text{Position}[i] \leq N \Rightarrow \text{CurrentQueue} = i$ \hfill (I37)

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

- $\mathcal{A}: \text{CurrentQueue} = i$, \hfill (9)
- $\mathcal{B}: (\exists p :: p@\{20\} \land p.idx = 1 - i)$, \hfill or
- $\mathcal{C}: \text{Position}[i] = 0 \land (\forall q :: q@\{0..3\} \lor q.idx = 1 - i)$.

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

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

**Proof:** For the sake of contradiction, assume

$p@\{4, 5\} \land p.idx = i \land \text{HistLen}[i] \geq 2N$. \hfill (9)

By applying (I30) with ‘$i$’ $\leftarrow$ $1 - i$, we have $\text{CurrentQueue} \neq 1 - i$, i.e.,

$\text{CurrentQueue} = i$. \hfill (10)

Thus, by (I32), we have

$\text{Position}[i] \leq N + 1$. \hfill (11)

Also, (9) implies

$\left| \{q :: q@\{6..11\} \land q.idx = i\} \right| \leq N - 1$.

Hence, by (I28), we have

$\text{HistLen}[i] - \text{Position}[i] \leq N - 1 \lor (\exists r :: r@\{19\} \land r.idx = 1 - i)$.

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

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

$\text{Position}[i] = N + 1$.

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

$\left| \{q :: q@\{6..11\} \land q.idx = i\} \right| \leq N - 2$.

Hence, by (I28), we have

$\text{HistLen}[i] - \text{Position}[i] \leq N - 2 \lor (\exists r :: r@\{19\} \land r.idx = 1 - i)$.

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

invariant $p@\{19, 20\} \land p.idx = i \Rightarrow \text{HistLen}[1 - i] = 0$ \hfill (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

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

holds. However, this contradicts (I13). \hfill □
invariant $p \oplus \{20\} \land p.\text{idx} = i \Rightarrow \text{Position}[1 - i] = 0$ \hfill (I40)

**Proof:** The antecedent may be established only by statement 19.$p$, which may do so only if $p.\text{idx} = i$ 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 $p \oplus \{20\} \land p.\text{idx} = i \land q \oplus \{11\} \land q.\text{idx} = 1 - i$ holds. However, this contradicts (I13).

\[\square\]

invariant $\text{Position}[i] \leq h < \text{HistLen}[i]$ \Rightarrow
\begin{align*}
(\exists p :: p \oplus \{6..11\} \land p.\text{idx} = i \land p.\text{position} = h) & \quad A \\
(\exists q :: q \oplus \{12..21\} \land p.\text{idx} = i \land p.\text{position} = h) & \quad B \\
(\exists q :: q \oplus \{8..11\} \land q.\text{idx} = i \land q.\text{position} = h) & \quad C
\end{align*}

\hfill (I41)

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

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

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

Statement 19.$q$ may establish $\text{Position}[i] \leq h$ only if executed when $q.\text{idx} = 1 - i$ holds, in which case, by (I39), it establishes $\text{Position}[i] = \text{HistLen}[i] = 0$. 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 $p.\text{idx} = i \land p.\text{position} = h$ holds. In this case, by (I14) and (I26), statement 11.$p$ establishes $\text{Position}[i] = h + 1$, and hence falsifies the antecedent.

\[\square\]

invariant $\text{Position}[i] = h > 0 \Rightarrow$
\begin{align*}
\text{Signal}[i][\text{Hist}[i][h]] = \text{true} & \quad A \\
(\exists p :: p \oplus \{12..21\} \land p.\text{idx} = i \land p.\text{position} = h - 1) & \quad B \\
(\exists q :: q \oplus \{8..11\} \land q.\text{idx} = i \land q.\text{position} = h) & \quad C
\end{align*}

\hfill (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 $p.\text{idx} = i \land p.\text{position} = h - 1$ holds, in which case it establishes disjunct $B$.

The only statements that may falsify disjunct $A$ are 5.$p$, 7.$p$, and 18.$p$, where $p$ is any arbitrary process. Statement 5.$p$ may falsify disjunct $A$ only if executed when $p \oplus \{5\} \land p.\text{idx} = i \land \text{HistLen}[i] = h - 1$ holds. Combining this with the antecedent, and using (I28), this implies $(\exists q :: q \oplus \{19\} \land q.\text{idx} = 1 - i)$. Hence, by applying (I13) with $p' \leftarrow q$ and $i' \leftarrow 1 - i$, we have $p \oplus \{0..3\} \lor p.\text{idx} = 1 - i$, which contradicts (12). It follows that statement 5.$p$ cannot falsify disjunct $A$ while the antecedent holds.

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

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

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

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

\[\square\]

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

**Invariant** \(0 < j \leq \text{HistLen}[i] \land j < 2N \Rightarrow \text{Hist}[i][j] \neq \bot\) (I44)

\[
\text{HistLen}[i][j-1] = (p, c) \land \text{Hist}[i][k-1] = (p, c + 1) \Rightarrow \\
\text{Hist}[i][j] \neq \text{Hist}[i][k]
\] (I45)

**Proof:** Invariants (I43)–(I45) follow easily from invariants (I16), (I17), (I18), and (I19), together with the assumption that the underlying fetch-and-\(\phi\) 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 \(\text{Tail}[i]\); (I44) states that each of the first \(2N - 1\) invocations writes to \(\text{Tail}[i]\) a value different from \(\bot\); (I45) states that any two successive invocations (among the first \(2N - 1\)) by the same process write different values to \(\text{Tail}[i]\).

\[
\text{Invariant} \quad \text{Position}[i] \leq h < \text{HistLen}[i] \land \text{Param}[i][h].\text{proc} = p \Rightarrow \\
p@\{6..11\} \land p.\text{idx} = i \land p.\text{position} = h
\] (I46)

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

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

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

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

Statement 19.\(q\) may establish \(\text{Position}[i] \leq h\) only if executed when \(q.\text{idx} = 1 - i\) holds, in which case, by (I39), it establishes \(\text{Position}[i] = \text{HistLen}[i] = 0\). 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 \(p.\text{idx} = i \land p.\text{position} = h\) holds. In this case, by (I14) and (I26), statement 11.\(p\) establishes \(\text{Position}[i] = h + 1\), and hence the antecedent is false after its execution.

\[
\text{Invariant} \quad p@\{6\} \land p.\text{idx} = i \Rightarrow \text{Position}[i] \leq p.\text{position}
\] (I47)

**Proof:** The only statement that may establish the antecedent is 5.\(p\), which may do so only if \(p.\text{idx} = i\). Let \(h\) be the value of \(\text{HistLen}[i]\) before the execution of 5.\(p\). By (I28), we have either \(\text{Position}[i] \leq h\) or \((3q :: q@\{19\} \land q.\text{idx} = 1 - i)\). However, by applying (I13) with ‘\(i' \leftarrow 1 - i\), the latter implies \(p@\{0..3\} \lor p.\text{idx} = 1 - i\), which implies that the antecedent is false. On the other hand, if \(\text{Position}[i] \leq h\) holds before the execution of 5.\(p\), then \(\text{Position}[i] \leq h = p.\text{position}\) 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 \(q.\text{idx} = i\). In this case, by (I14), 11.\(q\) increments \(\text{Position}[i]\) by one, and hence it may falsify the consequent only if executed when \(\text{Position}[i] = p.\text{position}\) holds. By applying (I26) with ‘\(p' \leftarrow q\), we also have \(\text{Position}[i] = q.\text{position}\). 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.\(q\).

\[
\text{Invariant} \quad \text{Position}[i] \leq j < k < \text{HistLen}[i] \Rightarrow \text{Hist}[i][j] \neq \text{Hist}[i][k]
\] (I48)

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

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

\[
\text{Position}[i] \leq j < k = \text{HistLen}[i] < 2N
\] (13)

holds before its execution. We consider two cases.
• If \( j = 0 \), then by (I18), we have \( \text{Hist}[i][j] = \bot \). Also, by applying (I44) with \( 'j' \leftarrow k \), and using (13), we have \( \text{Hist}[i][k] \neq \bot \). 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 \( \text{Param}[i][j-1] \) and \((r, c_2)\) denote the value of \( \text{Param}[i][k-1] \) 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 \( \text{Param}[i][l-1].\text{proc} \) 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 \( \text{Position}[i] < l \leq k \), by applying (I46) with \( 'p' \leftarrow q_l \) and \( 'h' \leftarrow l-1 \), and using (13), we have \( q_l.\text{position} = l-1 \) prior to the execution of 5.p. In particular, we have \( q.\text{position} = q_k.\text{position} = k-1 \), and

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

Since \( q_j = q \), this implies that \( \text{Position}[i] < j < k \) is false. Thus, by (13), we have \( j = \text{Position}[i] \).

Combining this with (14), we also have

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

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 \( \text{HistLen}[i] \), then it establishes \( \text{HistLen}[i] = 0 \), and hence the antecedent is false after its execution.

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

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

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

\textbf{invariant} \hspace{1em} p@\{11..22\} \Rightarrow p.\text{pos} = p.\text{position} \hspace{1em} (I49)

\textbf{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. □

\section*{Proof of Starvation-Freedom}

We begin with proving the following \textit{unless} and \textit{leads-to} properties. (A \textit{leads-to} B is true if and only if the following holds: if A holds at some state, then eventually B holds at either the same state or some later state. \textit{Leads-to} properties must hold only in fair histories. Recall that A \textit{unless} B is true if and only if the following holds: if A \land \neg B holds before some statement execution, then A \lor B holds after that execution. Informally, A is not falsified until B is established.) 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.

\[
p@\{6\} \land p.\text{idx} = i \land \text{Signal}[i][p.\text{prev}] = \text{true} \ \text{unless} \ p@\{7\} \hspace{1em} (U1)
\]
Proof: The only statement that may falsify \( p \oplus \{6\} \land p.\text{idx} = i \) is 6.\( p \), which establishes \( p \oplus \{7\} \).

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

\[
\text{CurrentQueue} = i \land p \oplus \{6\} \land p.\text{idx} = i \land p.\text{position} = h \land \text{Position}[i] = k
\]
leads-to \((\text{Signal}[i][p.\text{prev}] = \text{true} \lor \text{Position}[i] = k + 1)\)

\[
\text{CurrentQueue} = 1 - i \land p \oplus \{6\} \land p.\text{idx} = i \land p.\text{position} = h \land \text{Position}[i] = k
\]
leads-to \((\text{Signal}[i][p.\text{prev}] = \text{true} \lor \text{Position}[i] = k + 1)\)

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 \( p \oplus \{6\} \land p.\text{idx} = i \land p.\text{position} = h \) may be established only if statement 5.\( p \) is executed when \( \text{HistLen}[i] = 0 \) holds. However, by (I17) and (I18), this implies that \( \text{Tail}[i] = \text{Hist}[i][0] = \bot \) holds. Thus, statement 5.\( p \) establishes \( p \oplus \{8\} \), 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 \( \neg(\exists q :: q \oplus \{19\} \land q.\text{idx} = 1 - i) \). Thus, by (I28), and using \( p \oplus \{6\} \land p.\text{idx} = i \), we have \( k < \text{HistLen}[i] \). By applying (I41) with ‘\( h' \leftarrow k \), this in turn implies that a process \( q \) exists such that

\[
q \oplus \{6..11\} \land q.\text{idx} = i \land q.\text{position} = \text{Position}[i] = k. \quad (15)
\]

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 \( q.\text{prev} = \text{Hist}[i][0] = \bot \). Because \( q \oplus \{6, 7\} \Rightarrow q.\text{prev} \neq \bot \) is (trivially) an invariant, this implies that \( q \oplus \{8..11\} \) holds. Thus, \( q \) eventually executes statement 11 while \( q.\text{idx} = i \land q.\text{position} = k \) holds. In this case, by (I14) and (I26), 11.\( q \) establishes \( \text{Position}[i] = k + 1 \), and hence the right-hand side of (L1)/(L2) is established.

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

\[
\begin{align*}
\mathcal{A} : & \quad \text{Signal}[i][x] = \text{true}, \\
\mathcal{B} : & \quad r \oplus \{12..21\} \land r.\text{idx} = i \land r.\text{position} = k - 1, \quad \text{or} \\
\mathcal{C} : & \quad r \oplus \{8..11\} \land r.\text{idx} = i \land r.\text{position} = k.
\end{align*}
\]

Moreover, by (I3), \( q.\text{prev} = x \) 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 \( q.\text{idx} \) and \( q.\text{position} \) do not change while \( q \oplus \{6..11\} \) holds. Hence, by (I14) and (I26), \( q.\text{position} = q.\text{pos} = \text{Position}[i] = k \) 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), \( q.\text{prev} = x \) holds at state \( u \). Moreover, by (U1), if \( q \oplus \{6\} \land q.\text{prev} = x \) holds at state \( u \), then \( \mathcal{A} \) continues to hold until \( q \oplus \{7\} \) 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 \( B \) holds at state \( t \). In this case, by (I3), \( r.self = \text{Hist}[i][k] = x \) holds at state \( t \). Hence, if \( r \) eventually executes statement 21, then 21.\( r \) establishes \( A \).

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 \( B \) holds) as well as state \( u \). Thus, by (I15), \( 1 \leq \text{Position}[i] = r.pos + 1 \leq N \) holds at state \( t \). Hence, by (I37), \( \text{CurrentQueue} = i \) holds at state \( t \). Thus, if the left-hand side of (L2) is true at state \( t \), then \( B \) 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 \( C \) holds. Thus, by (I14) and (I26), 11.\( r \) establishes \( \text{Position}[i] = k + 1 \), 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).

\[
\begin{align*}
\text{CurrentQueue} &= i \land p@\{6\} \land p.idx = i \quad \text{leads-to} \quad p@\{7\} \quad \text{(L3)} \\
\text{CurrentQueue} &= 1 - i \land p@\{6\} \land p.idx = i \quad \text{leads-to} \quad p@\{7\} \quad \text{(L4)}
\end{align*}
\]

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

\[
\begin{align*}
\text{CurrentQueue} &= i \land p@\{6\} \land p.idx = i \quad \text{leads-to} \\
\text{Signal}[i][p.prev] &= true; \quad \text{(16)} \\
\text{CurrentQueue} &= 1 - i \land p@\{6\} \land p.idx = i \quad \text{leads-to} \\
\text{Signal}[i][p.prev] &= true. \quad \text{(17)}
\end{align*}
\]

Assume that the left-hand side of (L3) holds at some state \( t \). By (16), \( \text{Signal}[i][p.prev] = true \) 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, \( p.idx = i \) also continues to hold from state \( t \) to \( u \). Therefore, the left-hand side of (U1) holds at state \( u \). By (U1), \( \text{Signal}[i][p.prev] = true \) 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 state is eventually established. (Note that \( q@\{0\} \) implies \( \text{Active}[q] = false \) by (I33), and \( q@\{6\} \land q.idx = i \) implies \( \text{QueueIdx}[q] = i \) by (I34).)

\[
\begin{align*}
p@\{14, 15\} \land p.idx = i \land p.pos &= q \\
n\quad \text{leads-to} \quad q@\{0\} \lor (q@\{6\} \land q.idx = i) \lor p@\{21\} \quad \text{(L5)} \\
p@\{14, 15\} \land p.idx = i \land p.pos &= q \land q@\{1\} \\
n\quad \text{leads-to} \quad (q@\{6\} \land q.idx = i) \lor p@\{21\} \quad \text{(L6)}
\end{align*}
\]
Proof: Assume that the left-hand side of either (L5) or (L6) holds at state $t$. By (I2), one of the following holds at $t$.

$$\begin{align*}
A & : \ q \in \{3..22\} \land q.idx = 1 - i; \\
B & : \ q \in \{22\} \land q.idx = i; \\
C & : \ q \in \{0\}; \\
D & : \ q \in \{1..3\}; \\
E & : \ q \in \{4, 5\} \land q.idx = i; \\
F & : \ q \in \{6\} \land q.idx = i. 
\end{align*}$$

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

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

- If $p \in \{14, 15\} \land p.idx = i \land \text{CurrentQueue} = i$ 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 $\mathcal{D} \lor \mathcal{E}$ 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 $\mathcal{C} \lor \mathcal{F}$ implies the right-hand side of (L5), in order to prove (L5), it suffices to prove the following claim in addition to (19).

- If $\mathcal{A} \lor \mathcal{B}$ 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 $\text{CurrentQueue} = i$ (given in (18)), it follows that the former loop eventually terminates. If $q \in \{14, 15\}$, then by (I15), we have $\text{Position}[1 - i] = q.pos + 1$. Also, by (18) and (I37), we have $\text{Position}[1 - i] = 0 \lor \text{Position}[1 - i] > N$. Combining these two assertions, we have $q.pos = \text{Position}[1 - i] - 1 \geq N$, and hence $q \in \{14, 15\}$ 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), $\text{HistLen}[i] < 2N$ holds before the execution of 5.$q$. Moreover, by (18) and (I3), $\text{HistLen}[i] > p.position \geq 0$ also holds. (Note that $\text{HistLen}[i]$ is always nonnegative by (I21). Since $p.position$ is updated only by line 5d, it follows that $p.position$ is always nonnegative.) Combining these two assertions with (14), we have $\text{Tail}[i] \neq \bot$. 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.

\[ p \in \{14, 15\} \text{ leads-to } p \in \{21\} \] (L7)
Proof: For the sake of contradiction, assume that $p_{\{21\}}$ is never established. Let $i = p.idx$ and $q = p.pos$. By (L5), $q_{\{0\}} \lor (q_{\{6\}} \land q.idx = i)$ is eventually established.

First, assume that $q_{\{0\}}$ is established. If $q$ remains in its noncritical section forever, then by (I33), $Active[q] = false$ 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 $q_{\{6\}} \land q.idx = i$.

It follows that $q_{\{6\}} \land q.idx = i$ 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 \( p.pos \geq 2N - 1 \land p.idx = i \land p.self = x \) 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.

1. **Position** \([i] = 2N \) holds at \( s \).

2. If any process \( q \) satisfies \( q@\{16..20\} \land q.idx = 1 - i \) at state \( s \), then by applying (I13) with \( 'p' \leftarrow q \) and \( 'i' \leftarrow t \), every process \( r \) must satisfy \( r@\{0..3\} \lor r.idx = 1 - i \). However, this contradicts \( p@\{21\} \land p.idx = i \). Therefore, we have the following.

3. \( ¬(\exists r :: q@\{16..20\} \land q.idx = 1 - i) \) holds at \( s \).

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

\[
HistLen[i] = 2N \land ¬(\exists r :: r@\{6..11\} \land r.idx = i).
\]

By (I38), this also implies \( ¬(\exists r :: r@\{4,5\} \land r.idx = i) \). In particular, process \( p' \) satisfies \( ¬(p'@\{4,6\} \land p'.idx = i) \) at \( s \). Since \( p'@\{6\} \land p'.idx = i \) 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 \( s \) but 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 \( q@\{19,20\} \land q.idx = 1 - i \) 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, \( p' \)).

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 4) 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),

\[
|\{p :: p@\{6,7\} \land p.idx = i \land p.prev = x\}| \leq 1
\]

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 4 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 4 is safe, it suffices to show that
\[ | \{ p :: p \in \{14, 15, 21\} \land p.\text{pos} = h \} | \leq 1 \] (23)
is an invariant of Algorithm G-CC', for any h.

Consider a process \( p \) that satisfies \( p \in \{14, 15, 21\} \land p.\text{pos} = h \). By (I15) and (I27), Position\([p.\text{idx}] = h + 1 \land p.\text{position} = h \) holds as well. Since \( h < N \) holds by assumption (see statement 13 in Figure 6), by (I36), we also have \( p.\text{idx} = \text{CurrentQueue} \). Taken together with (I29), it follows that \( p \) is uniquely determined (i.e., (23) is true).

\[ \square \]

Statements 40–43: In order to show that the Entry/Exit pair at statements 40–43 in Figure 4 is safe, it suffices to show that
\[ | \{ p :: p \in \{21, 22\} \land p.\text{idx} = i \land p.\text{self} = x \land p.\text{pos} < 2N - 1 \} | \leq 1 \] (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
\[ p \in \{21, 22\} \land q \in \{21, 22\} \land p.\text{idx} = q.\text{idx} = i \land p.\text{self} = q.\text{self} = x \land p.\text{pos} < 2N - 1 \land q.\text{pos} < 2N - 1. \]

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

By applying (I3) to \( p \) and \( q \) separately, we have
\[ j < k < \text{HistLen}[i] \land \]
\[ x = p.\text{self} = \text{Hist}[i][j + 1] \land x = q.\text{self} = \text{Hist}[i][k + 1] \land \]
\[ \text{Param}[i][j].\text{proc} = p \land \text{Param}[i][k].\text{proc} = q. \]

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

\[ \square \]