Adaptive Mutual Exclusion with Local Spinning[*]

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

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

March 2001, Revised May 2003, January 2005, and July 2006

Abstract

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

Keywords: Adaptive mutual exclusion, local spinning, read/write atomicity, shared-memory systems, time complexity

1 Introduction

In this paper, we consider adaptive solutions to the mutual exclusion problem [14] under read/write atomicity. In work on mutual exclusion algorithms, *adaptive* and *local-spin* algorithms have received considerable interest. A mutual exclusion algorithm is adaptive if its time complexity is a function of the number of contending processes [13, 21, 24]. In local-spin algorithms, all busy-waiting loops are read-only loops in which only locally-accessible variables are read; a variable is locally accessible if it is stored in a local cache line (possible on a multiprocessor with coherent caches) or in a local memory partition (possible on a distributed shared-memory machine). By structuring busy-waiting loops in this way, contention for the processors-to-memory interconnection network can be greatly reduced. Performance studies presented in several papers [10, 15, 20, 25] have shown that local-spin algorithms typically scale well as contention increases, while non-local-spin algorithms do not.

In this paper, we present a local-spin algorithm that is adaptive under the *RMR* (*remote-memory-reference*) *time complexity measure* [25], which was motivated by work on local-spin algorithms. Under this measure, the time complexity of a mutual exclusion algorithm is defined as the number of remote memory references by one process in order to enter and then exit its critical section. Before describing our main contributions, we first give a brief overview of relevant related research on adaptive and local-spin algorithms.

Local-spin algorithms. In local-spin mutual exclusion 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 *locally accessible*, *i.e.*, they can be accessed without causing message traffic on the processors-to-memory interconnection network. Two architectural paradigms have been considered in the literature that allow shared

[*]Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289. A preliminary version of this paper was presented at the 14th International Symposium on Distributed Computing [6].

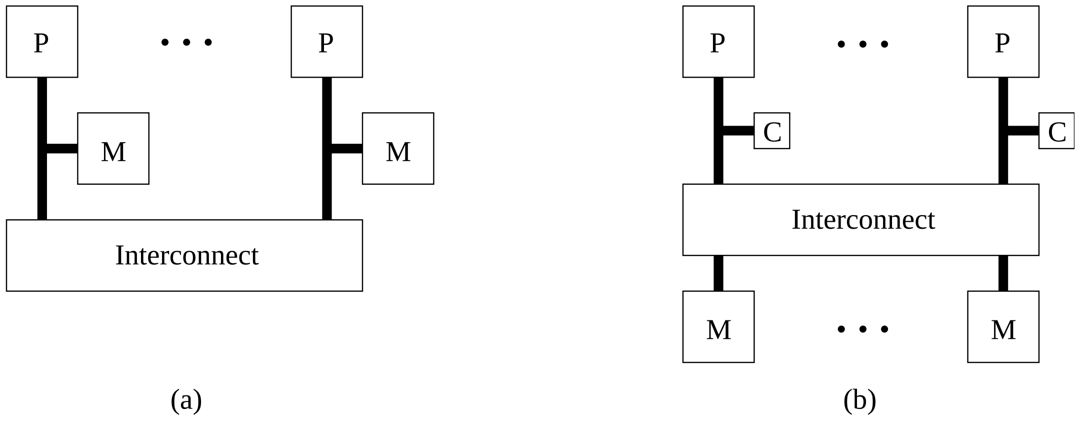


Figure 1: **(a)** DSM model. **(b)** CC model. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

variables to be locally accessed: distributed shared-memory (DSM) machines and cache-coherent (CC) machines. 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 on each processor; we further assume 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 dedicated spin variables are required on DSM machines, it is generally more difficult to design correct local-spin algorithms for DSM machines than for CC machines.

The first local-spin algorithms were algorithms in which read-modify-write operations are used to enqueue blocked processes onto the end of a "spin queue" [10, 15, 20]. Each of these algorithms has $O(1)$ RMR time complexity; thus, adaptivity is clearly a non-issue if appropriate read-modify-write instructions are available. Anderson presented the first local-spin mutual exclusion algorithm that uses only read and write operations. His algorithm has $\Theta(N)$ RMR time complexity, where N is the number of processes [5]. Later, Yang and Anderson presented a read/write algorithm with $\Theta(\log N)$ RMR time complexity, in which instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree [25]. They also presented a "fast-path" variant of this algorithm that allows the tree to be bypassed in the absence of contention. Although the contention-free RMR time complexity of this algorithm is $O(1)$, its RMR time complexity under contention is $\Theta(N)$ in the worst case, rather than $\Theta(\log N)$. This is because, to reopen the fast path after a period of contention ends, each process "polls" every other process to see if it is still contending. In recent work, Anderson and Kim presented a new fast-path mechanism that results in $O(1)$ RMR time complexity in the absence of contention and $\Theta(\log N)$ RMR time complexity under contention, when used with Yang and Anderson's algorithm [7].

Adaptive algorithms. Two notions of contention have been considered in the literature: "interval contention" and "point contention" [1]. These two notions are defined with respect to a history H.[1] The *interval contention* over H is the number of processes that are active in H, *i.e.*, that execute outside of their noncritical sections in H. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. In this paper, we mostly consider point contention. Throughout the paper, we let k' denote interval contention, and we let k

denote the point contention experienced by an arbitrary process over a history that starts when the process becomes active and ends when it once again becomes inactive.

Several different time complexity measures have been applied in work on adaptive algorithms. In defining a meaningful time complexity measure for concurrent algorithms, dealing with potentially unbounded busy-waiting loops is the main difficulty to be faced. Indeed, under the standard sequential-algorithms measure of counting all operations, such a busy-waiting loop has unbounded time complexity. This is inevitable under contention [4], and thus the standard sequential measure provides information that is not very interesting or useful. The *step time complexity* (also called the "remote step complexity") of an algorithm is the maximum number of shared-memory operations required by a process to enter and then exit its critical section, assuming that each "**await**" statement is counted as one operation [24]. This measure simply ignores repeated memory references generated by a process while it waits. The *system response time* is the length of time between the end of a critical section and the beginning of the next critical section, assuming every active process performs at least one step within some constant time bound [13]. By forcing active processes to take steps, unbounded waiting times are precluded, provided each waiting condition in an algorithm is eventually established by some process within a finite number of its own steps. The RMR time complexity measure is also of interest in work on adaptive algorithms.

Before continuing, let us examine the relationship between RMR time complexity and step time complexity. Under the DSM model, if all **await** statements within an algorithm access only locally-accessible variables, then each statement has $O(1)$ step time complexity and $O(1)$ RMR time complexity.[2] On the other hand, if an **await** statement accesses a non-local variable, then the algorithm's RMR time complexity is obviously unbounded. Under the CC model, any variable accessed in an **await** statement will be brought into a local cache line. Therefore, if write-update cache is used, then an algorithm's RMR time complexity is at most its step time complexity. However, in systems with write-invalidate caches, a single **await** statement might generate a large number of cache misses if the variables it accesses keep changing value without satisfying the **await** condition. The step time complexity measure ignores the remote references caused by these misses. Therefore, in general, an algorithm's step time complexity and RMR time complexity need not be the same.

Several read/write mutual exclusion algorithms have been presented that are adaptive under some of these time complexity measures. One of the first such algorithms was an algorithm of Styer that has $O(\min(N, k' \log N))$ step time complexity and $O(\min(N, k' \log N))$ system response time [24]. Choy and Singh presented an algorithm with $O(N)$ step time complexity and $O(k')$ system response time [13]. More recently, Attiya and Bortnikov presented an algorithm with $O(k)$ step time complexity and $O(\log k)$ system response time [11]. This algorithm was obtained by improving some of the mechanisms used in Choy and Singh's algorithm. In other work, Afek, Stupp, and Touitou [2] constructed an adaptive bakery algorithm, by combining Lamport's bakery algorithm [18] with wait-free objects. Their algorithm has $O(k^4)$ step time complexity and $O(k^4)$ system response time. Merritt and Taubenfeld [22] investigated mutual exclusion when the number of participating processes is possibly unbounded. In particular, they presented a number of mutual exclusion algorithms under various system models, some of which are adaptive under the system-response-time measure. Hence, we have the following: for systems with a possibly unbounded number of processes, there exists a starvation-free mutual exclusion algorithm with infinitely many variables and system response time that is a function of interval contention.

Contributions. None of the previously-cited adaptive algorithms is a local-spin algorithm, and thus each has unbounded RMR time complexity. Surprisingly, while adaptivity and local spinning have been the predominate themes in recent work on mutual exclusion, the problem of designing an adaptive, local-spin algorithm under read/write atomicity has remained open until recently. This problem is closed by us in this

[1]We define a *state* of a program to be an assignment of values to the variables of the program, including process program counters. An *event* is a particular execution of a statement. A *history* is a sequence t_1, e_1, t_2, e_2, $\ldots$, where each t_j is a state, each e_j is an event, and state t_{j+1} can be reached from state t_j via the execution of e_j.

[2]This conclusion is based upon the assumption that, under the step time complexity measure, each **await** statement has constant cost regardless of the number of variables it accesses. It is not clear if this is reasonable if the number of variables accessed is a function of N. In all papers where step time complexity is used, only **await** statements that access a constant number of variables are considered.

Algorithm	**System response time**	**Step time complexity**	**RMR/DSM time complexity**	**Space complexity**
Styer [24]	$O(\min(N, k' \log N))$	$O(\min(N, k' \log N))$	∞	$\Theta(N)$
Choy & Singh [13]	$O(k')$	$O(N)$	∞	$\Theta(N)$
Attiya & Bortnikov [11]	$O(\log k)$	$O(k)$	∞	$\Theta(N \log M)$
Attiya & Bortnikov [11]	$O(\log k')$	$O(k')$	∞	$\Theta(M \log M)$
Afek, *et al.* [2]	$O(k^4)$	$O(k^4)$	∞	$\Theta(N^3 + M^3 N)$
Afek, *et al.* [3]	$O((k')^2)$	$O(\min((k')^2, k' \log N))$	$O(\min((k')^2, k' \log N))$	$\Theta(N^2)$
This paper				
Algorithm U	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	∞
Algorithm B	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N^2)$
Algorithm L	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$O(\min(k, \log N))$	$\Theta(N)$

Table 1: Comparison of known adaptive algorithms. In this table, k denotes point contention, k' denotes interval contention, and M denotes an upper bound on the maximum number of processes concurrently active in the system (possibly less than N). (Although [2] uses a bounded number of variables, some of these variables are unbounded.) Each algorithm has bounded RMR time complexity on CC machines with write-update caches. Since these algorithms are quite complicated, it is unclear whether they are adaptive on CC machines in general.

paper. In addition, Afek, Stupp, and Touitou [3] have independently devised another local-spin adaptive algorithm, based on a long-lived implementation of a splitter element (see Section 2.1), with a structure that is similar to our algorithm. The adaptive algorithms mentioned so far are summarized in Table 1.

Our algorithm can be seen as an extension of the fast-path algorithm of Anderson and Kim [7]. That algorithm was devised by thinking about connections between fast-path mechanisms and long-lived renaming [23]. Long-lived renaming algorithms are used to "shrink" the size of the name space from which process identifiers are taken. The problem is to design operations that processes may invoke in order to acquire new names from the reduced name space when they are needed, and to release any previously-acquired name when it is no longer needed. In Anderson and Kim's algorithm, a particular name is associated with the fast path; to take the fast path, a process must first acquire the fast-path name. Our adaptive algorithm can be seen as a generalization of Anderson and Kim's fast-path mechanism in which *every* name is associated with some "path" to the critical section. The length of the path taken by a process is determined by the point contention that it experiences.

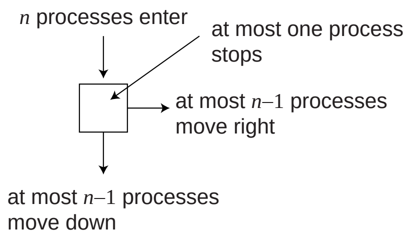
An informal description of our adaptive algorithm is given in the following section. A formal correctness proof for the algorithm is given in an appendix.

2 Adaptive Algorithm

In our adaptive algorithm, code sequences from several other algorithms are used. In Section 2.1, we present a review of these other algorithms and discuss some of the basic ideas underlying our algorithm. Then, in Sections 2.2–2.4, we present a detailed description of our algorithm.

2.1 Related Algorithms and Key Ideas

At the heart of our algorithm is the splitter element from the grid-based long-lived renaming algorithm of Moir and Anderson [23]. The splitter element, which was actually first used in Lamport's fast mutual exclusion algorithm [19], is defined by the code fragment shown in Figure 2. In this and subsequent figures, we assume that each labeled sequence of statements is atomic; in each figure, each labeled sequence reads or writes at most one shared variable. (References to unspecified code fragments, such as statement 4 in Figure 5, should be interpreted as *branches* to these code fragments.) Each process that invokes the splitter



n processes enter
at most one process stops
at most n–1 processes move right
at most n–1 processes move down

```
    shared variable  $X$: $\{\bot\} \cup \{0..N-1\}$ initially $\bot$;
                     $Y$: boolean initially $true$

    private variable  $dir$: $\{stop, right, down\}$

1:  $X := p$;
2:  if $\neg Y$ then $dir := right$
    else
3:      $Y := false$;
4:      if $X = p$ then $dir := stop$
                   else $dir := down$
        fi
    fi
```

Figure 2: The splitter element and the code fragment that implements it.

code either stops, moves down, or moves right (the move is defined by the value assigned to the variable *dir*). One of the key properties of the splitter that makes it so useful is the following: if several processes invoke a splitter, then at most one of them can stop at that splitter. To see why this property holds, suppose to the contrary that two processes p and q stop. Let p be the process that executed statement 4 (of Figure 2) last. Because p found that $X = p$ held at statement 4, X is not written by any process between p's execution of statement 1 and p's execution of statement 4. Thus, q executed statement 4 before p executed statement 1. This implies that q executed statement 3 before p executed statement 2. Thus, p must have read $Y = false$ at statement 2 and then assigned "$dir := right$," which is a contradiction. Similar arguments can be applied to show that if n processes invoke a splitter, then at most $n-1$ can move right, and at most $n-1$ can move down.

Because of these properties, it is possible to solve the renaming problem by interconnecting a collection of splitters in a grid as shown in Figure 3(a). (The diagonal numbering scheme depicted here is due to [12].) A name is associated with each splitter. If the grid has N rows and N columns, then by induction, every process eventually stops at some splitter.[3] When a process stops at a splitter, it acquires the name associated with that splitter. In the *long-lived* renaming problem [23], processes must have the ability to release the names they acquire. In the grid algorithm, a process can release its name by resetting (*i.e.*, reopening) each splitter on the path traversed by it in acquiring its name. (A splitter is *open* if it can be acquired once again by some process, and *closed* otherwise.) A splitter can be reset by resetting its Y variable to *true*. For the renaming mechanism to work correctly, it is important that a splitter be reset *only* if there are no processes "downstream" from it (*i.e.*, in the sub-grid "rooted" at that splitter). In Moir and Anderson's algorithm, it takes $O(N)$ time to determine if there are "downstream" processes. This is because each process checks every other process individually to determine if it is downstream from a splitter. As we shall see, a more efficient reset mechanism is needed for our adaptive algorithm.

The main idea behind our algorithm is to let an arbitration tree form dynamically within a structure similar to the renaming grid. This tree may not remain balanced, but its height is proportional to contention. The job of integrating the renaming aspects of the algorithm with the arbitration tree is greatly simplified if we replace the grid by a binary tree of splitters as shown in Figure 3(b). (Since we are now working with a tree, we will henceforth refer to the directions associated with a splitter as *stop*, *left*, and *right*.) Note that this results in many more names than before. However, this is not a major concern, because we are really not interested in minimizing the name space. The arbitration tree is defined by associating a three-process mutual exclusion algorithm[4] with each node in the renaming tree. This three-process algorithm can be implemented in constant time using the local-spin mutual exclusion algorithm of Yang and Anderson [25]. This algorithm may be invoked at a node by the process that stopped at that node, and one process from

[3]To see why, note that at most $N-1$ processes may move to the second row and below, at most $N-2$ to the third row and below, and so on. A similar remark applies to horizontal moves. Hence, by induction, the number of all processes that move down at least n times *and* move right at least m times is at most $N-n-m$. It follows that at most one process may enter each "leaf node" (nodes 11–15 in Figure 3(a)), where it must stop.

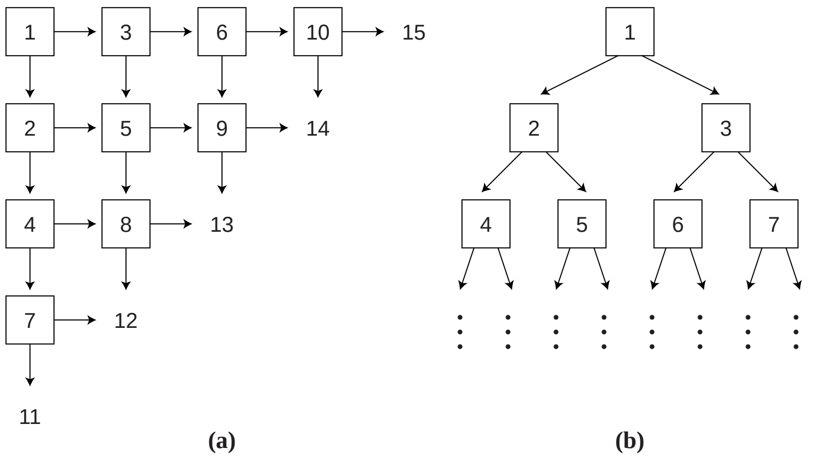


Figure 3: **(a)** Renaming grid (depicted for $N = 5$). **(b)** Renaming tree.

each of the left and right subtrees beneath that node. This is why a three-process algorithm is needed.

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

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

Step 2 p then competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root.

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

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

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

If we were to use a binary tree of height N, just as we previously had a grid with N row and N columns, then the total number of nodes in the tree would be $\Theta(2^N)$. We circumvent this problem by defining the tree's height to be $\lfloor \log N \rfloor$, which results in a tree with $\Theta(N)$ nodes. With this change, a process could "fall off" the end of the tree without acquiring a name. However, this can happen only if contention is $\Omega(\log N)$. To handle processes that "fall off the end," we introduce a second arbitration tree, which is implemented using Yang and Anderson's local-spin arbitration-tree algorithm [25]. We refer to the two trees used in our algorithm as the *renaming tree* and *overflow tree*, respectively. These two trees are connected by placing a two-process version of Yang and Anderson's algorithm on top of each tree, as illustrated in Figure 4(a). Figure 4(b) illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Figure 4(c) illustrates the steps that might be taken by a process q if contention is $\Omega(\log N)$.

A major difficulty that we have ignored until this point is that of efficiently reopening a splitter, as described in Step 4 above. In Moir and Anderson's renaming algorithm, it takes $O(N)$ time to reopen a splitter. To see why reopening a splitter is difficult, consider again Figure 2. If a process does succeed in stopping at a splitter, then that process can reopen the splitter itself by simply assigning $Y := true$. On the other hand, if no process succeeds in stopping at a splitter, then some process that moved left or right from that splitter must reset it. Unfortunately, because processes are asynchronous and communicate only

[4]We assume that all mutual exclusion algorithms used as subroutines in this paper are starvation-free.

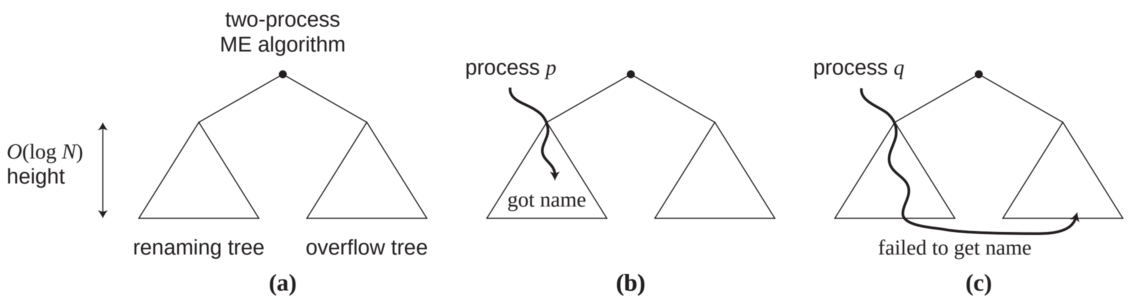


Figure 4: **(a)** Renaming tree and overflow tree. **(b)** Process p gets a name in the renaming tree. **(c)** Process q fails to get a name and must compete within the overflow tree.

by means of atomic read and write operations, it can be difficult for a left- or right-moving process to know whether some process has stopped at a splitter.

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

The algorithm of Afek *et al.* As mentioned before, Afek, Stupp, and Touitou [3] independently devised a local-spin adaptive algorithm (hereafter denoted as ALGORITHM AST) with the same tree structure as shown in Figures 3(b) and 4. Here we give a brief comparison between our algorithm and ALGORITHM AST.

The most challenging problem in using the renaming tree is the releasing of splitters once a process finishes its critical section (Step 4 of our algorithm). In our algorithm, this problem is solved by exploiting the fact that releasing can be done inside a critical section. In Afek's algorithm, the problem is solved by adopting a *long-lived adaptive splitter*, which supports two functions *Acquire* and *Release* [1]. Function *Acquire* returns *stop*, *right* or *down*, as in the splitter of Figure 2. If a process acquires a splitter (*i.e.*, invokes *Acquire* and gets the return value *stop*), then it must later release the splitter by calling *Release*. Processes that fail to acquire a splitter do not have to release it. In addition, a long-lived splitter satisfies the following properties.

- If a process invokes *Acquire* in the absence of contention, then it acquires (*i.e.*, stops at) the splitter.
- At most one process may acquire the splitter at any time.
- If n processes invoke *Acquire* on an unacquired splitter, then at most $n-1$ may move right and at most $n-1$ may move down.

Because of these properties, Steps 4 and 5 of our algorithm can be replaced by the following steps, as done in ALGORITHM AST.

Step 4 p executes each of the three-process exit sections on the path from the root to its acquired node.

Step 5 If p has stopped inside the renaming tree at step 1, then p releases the single splitter it has acquired.

As shown in [1], by using $3N$ copies of the splitter element, a long-lived adaptive splitter can be constructed that has $O(k')$ step time complexity, $O(k')$ RMR time complexity (under the DSM model), and $\Theta(N)$ space complexity, where k' is interval contention. Since each process accesses at most $\min(k', \log N)$ splitters in the renaming tree, ALGORITHM AST has $O(\min((k')^2, k' \log N))$ step and RMR time complexity, and $\Theta(N^2)$ space complexity. In can also be shown that ALGORITHM AST achieves $O((k')^2)$ system response time.

Having introduced the major ideas that underlie our algorithm, we now present a detailed description of the algorithm and its properties. We do this in three steps. In Section 2.2, we consider a version of the algorithm in which unbounded memory is used to reset splitters in constant time. Then, in Section 2.3, we consider a variant of the algorithm with $\Theta(N^2)$ space complexity in which all variables are bounded. Finally, in Section 2.4, we present another variant that has $\Theta(N)$ space complexity. A formal correctness proof for the final algorithm is presented in an appendix.

2.2 Algorithm U: Unbounded Algorithm

The first algorithm, which we call Algorithm U (for unbounded), is shown in Figure 5. Before describing how this algorithm works, we first examine its basic structure. At the top of Figure 5, definitions of two constants are given: L, which is the maximum level in the renaming tree (the root is at level 0), and T, which gives the total number of nodes in the renaming tree. As mentioned earlier, the renaming tree is comprised of a collection of splitters. These splitters are indexed from 1 to T. If splitter i is not a leaf, then its left and right children are splitters $2i$ and $2i+1$, respectively.

Each splitter i is accessed by three subroutines *AcquireNode*(i), *ReleaseNode*(i, dir), and *ClearNode*(i). Function *AcquireNode*(i) determines if the process can acquire splitter i by executing a modified version of the splitter code in Figure 2, and returns *stop*, *left*, or *right* depending on its outcome. A process invokes *ReleaseNode*(i, dir) in its exit section (statement 9) if and only if it has invoked *AcquireNode*(i) in its entry section and obtained a return value of *dir*. Similarly, a process additionally invokes *ClearNode*(i) in its exit section (statement 12) if and only if it has stopped at splitter i in its entry section (*i.e.*, its call to *AcquireNode*(i) returned *stop*). These two procedures "reopen" splitter i for future use, as explained in detail later.

Each splitter i is defined by four shared variables and an infinite shared array: $X[i]$, $Y[i]$, *Reset*$[i]$, *Round*$[i]$ (the array), and *Acquired*$[i]$. Variables $X[i]$ and $Y[i]$ are as in Figure 2, with the exception that $Y[i]$ now has an additional integer *rnd* field. As explained below, Algorithm U works by associating "round numbers" with the various rounds of competition for the name corresponding to each splitter. In Algorithm U, these round numbers grow without bound. The *rnd* field of $Y[i]$ gives the current round number for splitter i. *Reset*$[i]$ is used to reinitialize the *rnd* field of $Y[i]$ when name i is released. *Round*$[i][r]$ is used to determine if there exists a potential "winning" process that has succeeded in acquiring name i in round r. *Acquired*$[i]$ is set when some process acquires name i.

Each process descends the renaming tree, starting at the root, until it either acquires a name or "falls off the end" of the tree, as discussed earlier. A process determines if it can acquire name i by invoking *AcquireNode*(i) at statement 2. Statement 3 simply prepares for the next iteration of the **repeat** loop (if there is one). In *AcquireNode*(i), statements 15–18 correspond to the splitter code in Figure 2, and statements 19–23 are executed as part of a handshaking mechanism that prevents a process that is releasing a name from adversely interfering with processes attempting to acquire that name; this mechanism is discussed in detail below.

If a process p succeeds in acquiring a name while descending within the renaming tree, then it competes within the renaming tree by moving up from its acquired name to the root, executing the three-process entry sections on this path (statement 4). Each of these three-process entry sections is denoted "$\texttt{ENTRY}_3(n, d)$," where n is the corresponding tree node, and d is the "identity" of the invoking process. The "identity" that is used is simply the invoking process's direction out of node n (*stop*, *left*, or *right*) when it descended the renaming tree. After ascending the renaming tree, p invokes the two-process entry section "on top" of the renaming and overflow trees (as illustrated in Figure 4(a)) using "0" as a process identifier (statement 5). This entry section is denoted "$\texttt{ENTRY}_2(0)$."

If a process p does *not* succeed in acquiring a name while descending within the renaming tree, then it competes within the overflow tree (statement 6), which is implemented using Yang and Anderson's N-process arbitration-tree algorithm. The entry section of this algorithm is denoted $\texttt{ENTRY}_N(p)$. Note that p uses its own process identifier in this algorithm. After competing within the overflow tree, p executes the two-process algorithm "on top" of both trees using "1" as a process identifier (statement 7). This entry section is denoted "$\texttt{ENTRY}_2(1)$."

```
const
    $L = \lfloor \log N \rfloor$;                    /* depth of renaming tree = $O(\log N)$ */
    $T = 2^{L+1} - 1$                                /* size of renaming tree = $O(N)$ */
type
    $Ytype$ = record  $free$: boolean; $rnd$: $0..\infty$  end;   /* stored in one word */
    $Dtype$ = ($left$, $right$, $stop$);                           /* splitter moves */
    $Ptype$ = record  $node$: $1..2T+1$; $dir$: $Dtype$  end      /* path information */
shared variables
    $X$: array[$1..T$] of $0..N-1$;
    $Y$, $Reset$: array[$1..T$] of $Ytype$ initially ($true$, 0);
    $Round$: array[$1..T$][$0..\infty$] of boolean initially $false$;
    $Acquired$: array[$1..T$] of boolean initially $false$
private variables
    $node$: $1..2T+1$;      $n$: $1..T$;
    $level$, $j$: $0..L+1$;   $y$: $Ytype$;
    $dir$: $Dtype$;          $path$: array[$0..L$] of $Ptype$
```

```
process $p$ ::   /* $0 \le p < N$ */
while $true$ do
0:    Noncritical Section;
1:    $node$, $level$ := 1, 0;
      /* descend renaming tree */
      repeat
2:        $dir$ := $AcquireNode(node)$;
3:        $path[level]$ := ($node$, $dir$);
          if $dir = left$ then
              $level$, $node$ := $level + 1$, $2 \cdot node$
          elseif $dir = right$ then
              $level$, $node$ := $level + 1$, $2 \cdot node + 1$
          fi
      until ($level > L$) $\vee$ ($dir = stop$);
      if $level \le L$ then      /* got a name */
          /* compete in renaming tree, then 2-proc. alg. */
          for $j$ := $level$ downto 0 do
4:            $\mathtt{ENTRY}_3$($path[j].node$, $path[j].dir$)
          od;
5:        $\mathtt{ENTRY}_2(0)$
      else      /* did not get a name */
          /* compete in overflow tree, then 2-proc. alg. */
6:        $\mathtt{ENTRY}_N(p)$;
7:        $\mathtt{ENTRY}_2(1)$
      fi;
8:    Critical Section;
      /* reset splitters */
      for $j$ := $min(level, L)$ downto 0 do
          $n$, $dir$ := $path[j].node$, $path[j].dir$;
9:        $ReleaseNode(n, dir)$
      od;
      /* execute appropriate exit sections */
      if $level \le L$ then
10:       $\mathtt{EXIT}_2(0)$;
          for $j$ := 0 to $level$ do
11:           $\mathtt{EXIT}_3$($path[j].node$, $path[j].dir$) od;
12:       $ClearNode(node)$
      else
13:       $\mathtt{EXIT}_2(1)$;
14:       $\mathtt{EXIT}_N(p)$
      fi
od
```

```
function $AcquireNode$($n$: $1..T$): $Dtype$
15:  $X[n]$ := $p$;
16:  $y$ := $Y[n]$;
     if $\neg y.free$ then return $right$ fi;
17:  $Y[n]$ := ($false$, 0);
18:  if $X[n] \ne p$ $\vee$
19:      $Acquired[n]$ then
         return $left$
     fi;
20:  $Round[n][y.rnd]$ := $true$;
21:  if $Reset[n] \ne y$ then
22:      $Round[n][y.rnd]$ := $false$;
         return $left$
     fi;
23:  $Acquired[n]$ := $true$;
     return $stop$

procedure $ReleaseNode$($n$: $1..T$, $dir$: $Dtype$)
24:  if $dir = right$ then return fi;
25:  $y$ := $Reset[n]$;
26:  $Reset[n]$ := ($false$, $y.rnd$);
27:  if $dir = stop$ $\vee$ $\neg Round[n][y.rnd]$ then
28:      $Reset[n]$ := ($true$, $y.rnd + 1$);
29:      $Y[n]$ := ($true$, $y.rnd + 1$)
     fi

procedure $ClearNode$($n$: $1..T$)
30:  $Acquired[n]$ := $false$
```

Figure 5: ALGORITHM U: Adaptive algorithm with unbounded memory.

After completing the appropriate two-process entry section, process p executes its critical section (statement 8). It then resets each of the splitters that it visited while ascending the renaming tree by invoking *ReleaseNode* at statement 9. This reset mechanism is discussed in detail below. (As explained later in Section 2.4, resetting the splitters in ascending order is essential to achieving point-contention sensitivity.) Process p then executes the exit sections corresponding to the entry sections it executed previously (statements 10–14). The exit sections are specified in a manner that is similar to the entry sections. (Statement 12 is discussed below.)

We now describe in detail the three subroutines that are executed to acquire or reset some splitter i. A process p tries to acquire node i by executing $AcquireNode(i)$. Statements 15–18 correspond to the original splitter code. If p reaches statement 19, then it first checks if the last process that acquired node i is still active (statement 19). If that is the case, then p is deflected left.

Otherwise, p assigns "$Round[i][r] := true$" (statement 20), where r is the value of $Y.rnd$ when p executed statement 16. As stated before, $Round[i][r]$ is used to indicate that there exists a process (that is, p) that may potentially acquire node i. Process p then determines if the value of $Reset[i]$ has changed (statement 21). As explained later, when node i is open, the value of $Reset[i]$ is always equal to $Y[i]$. (In particular, $Reset[i] = Y[i]$ was true when p executed statement 16.) Therefore, if $Reset[i]$ has changed, then either node i is closed, or node i has been reopened (since p executed statement 17) by some other process and a new round of competition has started. In either case, p cannot acquire node i. Hence, p resets $Round[i][r]$ (statement 22), and then returns *left*. Finally, if p passes each of these tests, then it declares that node i is now acquired (statement 23), and returns *stop*.

Now consider $ReleaseNode(i)$. A process p tries to reopen node i if it has visited node i during its entry section. Note that a process in its entry section may move right at node i only if the node is already closed (statement 16). Hence, if $p.dir = right$, then p does not have to reset node i (statement 24).

Otherwise, p reads the value of $Reset[i]$ (statement 25), which contains the most recently used round number. It then changes $Reset[i].free$ from *true* to *false* (statement 26). This is necessary because some other process q may be concurrently executing in $AcquireNode(i)$. (For example, p and q may have started executing *AcquireNode* concurrently, but q may have been delayed until p finished its critical section.)

The handshaking of statements 20, 21, 26, and 27 prevents p from reopening node i while q acquires node i. In particular, p writes to $Reset[i]$ and then checks the value of $Round[i][r]$ (statements 26 and 27); q writes to $Round[i][r]$ and then checks the value of $Reset[i]$ (statements 20 and 21). Hence, either p discovers q and skips resetting node i, or q discovers p and is deflected left.

On the other hand, if p discovers no such process q, or if p has acquired node i, then p executes statements 28 and 29, thereby resetting node i.[5]

Procedure $ClearNode(i)$ is executed as the last step of p's exit section, only if p has acquired node i — it prevents the reopening of splitter i from actually taking effect until after p has finished executing its exit section. (Otherwise, another process may acquire node i and execute statement 4 while p is still executing within statement 11 with "process identifier" $(i, stop)$. Clearly, this must be avoided.) This is the reason we need two separate procedures *ReleaseNode* and *ClearNode*, so that the former is executed effectively within critical sections and the latter, outside of critical sections.

We now prove the correctness of ALGORITHM U. To facilitate this discussion, we will index all statements by i. For example, when we refer to the execution of statement 17[i] by process p, we mean the execution of statement 17 by p when its argument n equals i. We begin by stating several notational conventions that will be used throughout this discussion.

Notational Conventions: We use $s.p$ to denote the statement with label s of process p, and $p.v$ to represent p's private variable v. Let S be a subset of the statement labels in process p. Then, $p@S$ holds if and only if the program counter for process p equals some value in S. (Note that if s is a statement label, then $p@\{s\}$

[5]In some cases, statements 28 and 29 are unnecessary. For example, if the round number that p reads at statement 25 is greater than the value p has read at statement 16, then a new round has already started, so p can safely skip statements 25 and 26 without the danger of node i being closed forever. However, such an additional check would complicate the algorithm. Moreover, it would require p to remember the round numbers of every node it has visited, inducing additional $L \cdot N = \Theta(N \log N)$ private variables (L for each process). Since our final algorithm has $\Theta(N)$ space complexity, this is rather undesirable.

means that statement s of process p is *enabled*, *i.e.*, p has not yet executed s.)

As stated earlier, we assume that each labeled sequence of statements is atomic. For example, consider statement $18.p$. If $18.p$ is executed while $X[p.n] = p$ holds, then it establishes $p@\{19\}$. On the other hand, if $18.p$ is executed while $X[p.n] \neq p$ holds, then it returns *left* from *AcquireNode* and establishes $p@\{3\}$. Statements 2, 9, and 12 are considered as *branches* to each subroutine, and do nothing but establishing $p@\{15, 24, 30\}$, respectively. We number statements in this way to reduce the number of cases that must be considered in the proof. Note that each numbered sequence of statements reads or writes at most one shared variable, except for calls to `ENTRY` and `EXIT` routines.

We now show that each invocation of these routines can be also considered atomic. For example, consider some fixed node i. Algorithm U ensures that, for each direction *dir*, calls to $\texttt{ENTRY}_3(i, dir)$ and $\texttt{EXIT}_3(i, dir)$ by possibly different processes do not overlap. (In other words, if a process p invokes $\texttt{EXIT}_3(i, dir)$, then no other process may start $\texttt{ENTRY}_3(i, dir)$ until p's execution of $\texttt{EXIT}_3$ is finished. This is because a process may reach statement $4[i]$ only after the previous owner of node i has executed *ClearNode*(i).) Similar arguments apply to other `ENTRY`/`EXIT` calls. As a result, we can assume that these subroutines execute atomically and do not access any of the shared variables of Algorithm U. $\square$

As explained above, one of the problems with the splitter code is that it is difficult for a left- or right-moving process at splitter i to know which (if any) process has acquired name i. In Algorithm U, this problem is solved by viewing the computation involving each splitter as occurring in a sequence of rounds. Each round ends when the splitter is reset. During a round, at most one process succeeds in acquiring the name of the splitter. Note that it is possible that *no* process acquires the name during a round. So that processes can know the current round number at splitter i, an additional *rnd* field has been added to $Y[i]$. In essence, the round number at splitter i is used as a temporary identifier to communicate with the winning process (if any) at splitter i. This identifier will increase without bound over time, so the potential winner of each round can be uniquely identified. Formally, we have the following definitions.

Definition We define a *state* to be an assignment of values to the variables of the program, including process program counters. An *event* is a particular execution of a numbered statement, denoted $s.p$. (Although a process p may execute the same statement s multiple times, it is usually clear from the context which event we are concerned with. For the sake of brevity, we do not extend our notation to distinguish among these events.) $\square$

Definition An *execution interval* is denoted (e, f), where e is either 0 (indicating that the interval starts with the initial state) or some event, and f is either ∞ (indicating that the interval never terminates) or another event that is executed after e. Interval (e, f) contains all states after the execution of e and before the execution of f. We say that (e, f) is *infinite* if $f = \infty$, and *finite* otherwise. $\square$

Definition A *round* of a splitter i is an execution interval $H = (e, f)$ satisfying the following.

- e is either 0 or an event that writes $Y[i] := (true, r)$, for some r.
- f is either ∞ or another event that writes $Y[i] := (true, s)$, for some s.
- For any event g inside H (excluding e and f), if g writes $Y[i]$, then g writes $Y[i] := (false, 0)$.

We say that r is the *round number* of H. (If $e = 0$, then the round number of H is 0, the initial value of $Y[i].rnd$.) We also use $\mathcal{R}(i, r)$ to denote round r of splitter i. $\square$

In order to show the correctness of Algorithm U, we need several properties, given below. The following property follows immediately from the correctness of the original splitter code.

Property 1: Let $\mathcal{S}$ be the set of processes that execute statement $16[i]$ during round $\mathcal{R}(i, r)$. Then, at most one process p in $\mathcal{S}$ reaches statement $20[i]$ (during either round $\mathcal{R}(i, r)$ or some later round). $\square$

We say that p is the *winner* of round $\mathcal{R}(i, r)$ if p executes statement $16[i]$ during $\mathcal{R}(i, r)$ and then stops at splitter i (*i.e.*, returns *stop* at statement $23[i]$). By Property 1, if a winner exists, then it is uniquely defined.

The following property asserts that $\mathit{Reset}[i]$ correctly indicates the current round number.

Property 2: During round $\mathcal{R}(i,r)$, either $\mathit{Reset}[i].\mathit{rnd} = r$ holds, or there exists a process p satisfying $p@\{29\} \wedge p.n = i$ (*i.e.*, p is about to execute statement $29[i]$). Moreover, in the latter case, $\mathit{Reset}[i].\mathit{rnd} = r+1$ holds.

Proof: Note that $\mathit{Reset}[i]$ is only updated inside $\mathit{ReleaseNode}(i)$. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, invocations of *ReleaseNode* (by different processes) cannot overlap. From this, Property 2 easily follows. $\Box$

With the added *rnd* field, a left- or right-moving process at splitter i has a way of identifying a process that has acquired the name at splitter i. To see how this works, consider what happens during round $\mathcal{R}(i,r)$. By Property 1, of the processes that execute statement $16[i]$ during $\mathcal{R}(i,r)$, at most one will reach statement $20[i]$. A process that reaches statement $20[i]$ will either stop at node i by executing statement $23[i]$ (*i.e.*, become the winner of $\mathcal{R}(i,r)$) or be deflected left. This gives us two cases to analyze, depending on whether $\mathcal{R}(i,r)$ has a winner or not. These two cases are considered in the following two properties.

Property 3: Assume that $\mathcal{R}(i,r)$ has a winner p. Then, $\mathcal{R}(i,r)$ does not end until p executes statement $29[i]$.

Proof: By definition, $\mathcal{R}(i,r)$ may end only if some process q executes statement $29[i]$. For the sake of contradiction, assume that some other process $q \neq p$ executes statement $29[i]$ during $\mathcal{R}(i,r)$ (which would then terminate $\mathcal{R}(i,r)$). Without loss of generality, assume that $\mathcal{R}(i,r)$ is the first round of splitter i in which this happens.

Consider the state at which q executes statement $29[i]$. Because p has executed statement $16[i]$ but has not yet executed statement $29[i]$, $p@\{17[i]..23[i], 3..5, 8, 9, 24[i]..29[i]\}$ holds at that state. Since we are assuming that the `ENTRY` and `EXIT` calls are correct, q cannot execute statement $29[i]$ while $p@\{8, 9, 24[i]..29[i]\}$ holds. Also note that statement $29[i].q$ starts a new round, say, $\mathcal{R}(i,s')$, where $s' > r$. Thus, if p executes statement $21[i]$ *after* q executes statement $29[i]$, then $21[i].p$ is executed during some round $\mathcal{R}(i,s)$, where $s \geq s' > r$. Therefore, by Property 2, p will find either $\mathit{Reset}[i].\mathit{rnd} = s$ or $\mathit{Reset}[i].\mathit{rnd} = s+1$ when it executes statement $21[i]$. In either case, we have $\mathit{Reset}[i].\mathit{rnd} > r$, and hence p cannot stop at splitter i.

The only remaining possibility is that $p@\{23[i], 3..5\}$ holds when q executes statement $29[i]$. (Note that, in this case, if q *were* to reopen splitter i then we could end up with two processes concurrently invoking $\mathtt{ENTRY}_3(i, \mathit{stop})$ and $\mathtt{EXIT}_3(i, \mathit{stop})$ at statements 4 and 11, *i.e.*, both processes use i as a "process identifier." The `ENTRY` and `EXIT` calls obviously cannot be assumed to work correctly if such a scenario could happen.)

So, assume that q executes statement $29[i]$ while $p@\{23[i], 3..5\}$ holds. Note that the round of splitter i can change only by some process executing statement $29[i]$ inside $\mathit{ReleaseNode}(i)$. Since invocations of *ReleaseNode* (by different processes) cannot overlap, it follows that q executes statements $25[i]$–$29[i]$ during $\mathcal{R}(i,r)$. Thus, by Property 2, q reads $\mathit{Reset}[i].\mathit{rnd} = r$ at statement $25[i]$.

Thus, for q to execute statement $29[i]$, it must find $q.\mathit{dir} = \mathit{stop} \vee \mathit{Round}[i][r] = \mathit{false}$ at statement $27[i]$. First, if $q.\mathit{dir} = \mathit{stop}$ holds, then q has stopped at splitter i, *i.e.*, q is the winner of round $\mathcal{R}(i,s)$, for some s. Combined with our assumption that q executes statement $29[i]$ during round $\mathcal{R}(i,r)$, it follows that round $\mathcal{R}(i,s)$ has ended (and round $\mathcal{R}(i,r)$ has started) before q executes statement $29[i]$, which contradicts our assumption that $\mathcal{R}(i,r)$ is the first such round.

It follows that q reads $\mathit{Round}[i][r] = \mathit{false}$ at statement $27[i]$. For this to happen, q must execute statement $27[i]$ before p assigns $\mathit{Round}[i][r] := \mathit{true}$ at statement $20[i]$. Hence, statement $26[i]$ is executed by q before statement $21[i]$ is executed by p. Thus, p must find $\mathit{Reset}[i] \neq p.y$ at statement $21[i]$, *i.e.*, it is deflected left at splitter i, a contradiction. $\Box$

The following property pertains to the case when $\mathcal{R}(i,r)$ has no winner — in this case, some process that is deflected left eventually reopens splitter i.

Property 4: Assume that round $\mathcal{R}(i,r)$ has no winner. If some process executes statement $17[i]$ during $\mathcal{R}(i,r)$, then some process eventually executes statement $29[i]$.

Proof: Let $\mathcal{S}$ be the (nonempty) set of processes that execute statement 17[i] during round $\mathcal{R}(i,r)$. We claim that at least one process p in $\mathcal{S}$ finds $Round[i][r]$ to be false at statement 27[i]. We consider two cases.

First, assume that there exists some process p that executes statement 16[i] during $\mathcal{R}(i,r)$, and subsequently executes statement 20[i]. By Property 1, p is unique. Since $\mathcal{R}(i,r)$ has no winner, p must be deflected left by executing statements 21[i] and 22[i], re-establishing $Round[i][r] = \mathit{false}$. Thus, p eventually reads $Round[i][r] = \mathit{false}$ at statement 27[i] (unless some other process has already done so and executed statement 29[i]). Note that, because the `ENTRY` and `EXIT` routines are assumed to be starvation-free, and because Algorithm U does not contain any busy-waiting loops elsewhere, p cannot be stalled indefinitely without executing statement 27[i].

Second, assume that there exists no process that executes statement 16[i] during $\mathcal{R}(i,r)$, and then executes statement 20[i]. In this case, since $Round[i][r]$ is initially *false*, it remains *false* throughout the execution of the algorithm. Thus, some process eventually reads $Round[i][r] = \mathit{false}$ at statement 27[i]. □

By Property 1 and Property 3, if a process p stops at splitter i, then no other process stops at splitter i, and the splitter remains closed until p finishes execution of $ReleaseNode(i)$. Recall that the assignments to $Acquired[i]$ at statements 23[i] and 30[i] prevent the reopening of splitter i from actually taking effect until after p has finished executing its exit section.

Note that splitter i is closed if and only if some process establishes $Y[i] = (\mathit{false}, 0)$ by executing statement 17[i]. Thus, by Property 4, if no process stops at splitter i, and if splitter i becomes closed, then it is eventually reopened.

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

Having dispensed with basic correctness, we now informally argue that Algorithm U is contention-sensitive. (For the sake of brevity, we give a rigorous proof of contention sensitivity only for Algorithm L.) For a process p to descend to a splitter at level l in the renaming tree, it must have been deflected left or right at each prior splitter it accessed. Just as with the original grid-based long-lived renaming algorithm [23], this can only happen if the point contention experienced by p is $\Omega(l)$. Note that the time complexity per level of the renaming tree is constant. Moreover, with the `ENTRY` and `EXIT` calls implemented using Yang and Anderson's algorithm, the $\mathtt{ENTRY}_2$, $\mathtt{EXIT}_2$, $\mathtt{ENTRY}_3$, and $\mathtt{EXIT}_3$ calls take constant time, and the $\mathtt{ENTRY}_N$ and $\mathtt{EXIT}_N$ calls take $\Theta(\log N)$ time. Note that the $\mathtt{ENTRY}_N$ and $\mathtt{EXIT}_N$ routines are called by a process only if its point contention is $\Omega(\log N)$. Thus, we have the following.

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

Of course, the problem with Algorithm U is that we need infinite-sized arrays. We now consider a variant of Algorithm U in which space is bounded.

2.3 Algorithm B: Bounded Algorithm

In Algorithm B (for bounded space), $Y[i].rnd$ is incremented modulo-N, and hence does not grow without bound. Algorithm B is shown in Figure 6. The only new variable is *Obstacle*, which acts as an "obstacle" to prevent $Y[i].rnd$ from cycling modulo-N as long as there is a possibility of "interference," as explained below.

With $Y[i].rnd$ being incremented modulo-N, the following potential problem arises. A process p may reach statement 21[i] in Figure 6 with $y.rnd = r$ and then be delayed. While delayed, other processes may repeatedly increment $Y[i].rnd$ (statement 34[i]) until it "cycles back" to r. Another process q could then reach statement 21[i] with $y.rnd = r$. This is a problem because both p and q may acquire node i simultaneously.

Algorithm B prevents such a scenario from happening by preventing $Y[i].rnd$ from cycling while a process p that stops (or may stop) at splitter i executes within $AcquireNode(i)$. Informally, cycling is

```
/* all constant, type, and private variable declarations are as defined in Figure 5 except as noted here */

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

shared variables
    X: array[1..T] of 0..N − 1;
    Y, Reset: array[1..T] of Ytype initially (true, 0);
    Round: array[1..T][0..N − 1] of boolean initially false;
    Obstacle: array[0..N − 1] of 0..T initially 0;
    Acquired: array[1..T] of boolean initially false

process p ::   /* 0 ≤ p < N */
function AcquireNode(n: 1..T): Dtype
15: X[n] := p;
16: y := Y[n];
    if ¬y.free then return right fi;
17: Y[n] := (false, 0);
18: Obstacle[p] := n;
19: if X[n] ≠ p ∨
20:       Acquired[n] then
        return left
    fi;
21: Round[n][y.rnd] := true;
22: if Reset[n] ≠ y then
23:     Round[n][y.rnd] := false;
        return left
    fi;
24: Acquired[n] := true;
    return stop

procedure ReleaseNode(n: 1..T, dir: Dtype)
25: Obstacle[p] := 0;
26: if dir = right then return fi;
27: Y[n] := (false, 0);
28: X[n] := p;
29: y := Reset[n];
30: Reset[n] := (false, y.rnd);
31: if (dir = stop ∨ ¬Round[n][y.rnd]) ∧
32:       Obstacle[y.rnd] ≠ n then
33:     Reset[n] := (true, y.rnd + 1 mod N);
34:     Y[n] := (true, y.rnd + 1 mod N)
    fi;
35: if dir = stop then Round[n][y.rnd] := false fi

procedure ClearNode(n: 1..T)
36: Acquired[n] := false
```

Figure 6: Algorithm B: adaptive algorithm with $\Theta(N^2)$ space complexity. Statements 1–14 are identical to Algorithm U and not shown here.

prevented by requiring process p to erect an "obstacle" that prevents $Y[i].rnd$ from being incremented beyond the value p. More precisely, note that before reaching statement 21[i], process p must first assign $Obstacle[p] := i$ at statement 18[i]. Note further that before a process can increment $Y[i].rnd$ from r to $r + 1$ **mod** N (statement 34[i]), it must first read $Obstacle[r]$ (statement 32[i]) and find it to have a value different from i. This check prevents $Y[i].rnd$ from being incremented beyond the value p while p executes within $AcquireNode(i)$. Note that process p resets $Obstacle[p]$ to 0 at statement 25. This is done to ensure that p's own obstacle does not prevent it from incrementing a splitter's round number. The discussion so far is formalized in the following property.

Property 5: If a process p executes statement 16[i] during round $\mathcal{R}(i, r)$ and subsequently reaches statement 21[i], then $\mathcal{R}(i, p + 1 \textbf{ mod } N)$ does not start until p returns from $AcquireNode(i)$.

Proof: For the sake of contradiction, assume that a process q executes statements 25[i]–34[i] during $\mathcal{R}(i, p)$, and starts round $\mathcal{R}(i, p + 1 \textbf{ mod } N)$, while $p@\{17[i]..24[i]\}$ holds. We consider three cases.

First, if q executes statement 28[i] before p executes statement 15[i], then q establishes $Y[i] = (false, 0)$ at statement 27[i] before p executes statement 16[i]. Moreover, because we assume the correctness of the `ENTRY` and `EXIT` routines, no other process may reopen splitter i (by establishing $Y[i].free = true$) until q executes 34[i]. It follows that p is deflected right at statement 16[i], and does not reach statement 21[i].

Second, assume that q executes statement 28[i] while $p@\{16[i]..19[i]\}$ holds. In this case, p finds $X[i] \neq p$ at statement 19[i] and is deflected left without reaching statement 21[i].

Finally, assume that q executes statement 28[i] *after* p executes statement 19[i]. In this case, p has executed statement 18[i] and established $Obstacle[p] = i$, which continues to hold while p executes inside

AcquireNode(i). Therefore, q reads *Obstacle*$[p] = i$ at statement $32[i]$, and does not execute statements $33[i]$ and $34[i]$. (Note that, by Property 2, q reads *Reset*$[n].rnd = p$ at statement $29[i]$.) □

Since round numbers cycle modulo-N, we have the following: if a process p executes statement $16[i]$ during round $\mathcal{R}(i, r)$ and reaches statement $21[i]$, then $Y[i].rnd = r$ is not re-established until p returns from *AcquireNode*(i). Therefore, the recycling of round numbers does not affect the correctness of Algorithm B.

The only statement not explained so far is statement 35, which simply resets the *Round* variable to be used again. From the discussion above, we have the following lemma.

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

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

2.4 Algorithm L: Linear-space Algorithm

The final algorithm we present is depicted in Figure 7. We refer to this algorithm as Algorithm L (for linear space). In Algorithm L, a common pool of round numbers ranging over $\{1, \ldots, S\}$ is used for all splitters in the renaming tree. As we shall see, $O(N)$ round numbers suffice. In Algorithm B, our key requirement for round numbers was that they not be reused "prematurely." With a common pool of round numbers, a process should not choose r as the next round number for some splitter if there is a process *anywhere* in the renaming tree that "thinks" that r is the current round number of some splitter it has accessed.

Fortunately, since each process selects new round numbers within its critical section, it is fairly easy to ensure this requirement. All that is needed are a few extra data structures that track which round numbers are currently in use. These data structures replace the *Obstacle* array of Algorithm B. The main new data structure is a queue *Free* of round numbers. In addition, there is a new shared array *Inuse*, and a new shared variable *Check*. We assume that the *Free* queue can be manipulated by the following operations.

- *Enqueue*(*Free*, $i : 1..S$): Enqueues the integer i onto the end of *Free*. (We assume that i is not already contained in *Free*.)
- *Dequeue*(*Free*): $1..S$: Dequeues the element at the head of *Free*, and returns that element.
- *MoveToTail*(*Free*, $i : 1..S$): If i is in *Free*, then it is moved to the end of the queue; otherwise, do nothing.

If the *Free* queue is implemented as a doubly-linked list, then all of these operations can be performed in constant time. We stress that all of these operations are executed *only* within critical sections, *i.e.*, *Free* is really a *sequential* data structure.

When comparing Algorithms B and L, the only difference before the critical section is statement $18[i]$: instead of updating *Obstacle*$[p]$, process p now marks the round number r it just read from $Y[i]$ as being "in use" by assigning *Inuse*$[p] := r$. The only other differences are in *ReleaseNode*. Statements 31–34 are executed to ensure that no round number currently "in use" can propagate to the head of the *Free* queue. In particular, if a process p is delayed after having obtained r as the current round number for some splitter, then while it is delayed, r will be moved to the end of the *Free* queue by every N^{th} critical section execution. (A similar mechanism is used in the constant-time implementation of load-linked and store-conditional from read and compare-and-swap of Anderson and Moir [9].) With $S = T + 2N$ round numbers, this is sufficient to prevent r from reaching the head of the queue while p is delayed. (Among the $S = T + 2N$ round numbers, T of them are assigned to the splitters. The *Free* queue needs $2N$ round numbers because the calls to *Dequeue* and *MoveToTail* can cause a round number to migrate toward the head of the *Free* queue by two positions per critical section execution. See Figure 8 for an example.)

```
/* all constant, type, and private variable declarations are as defined in Figure 6 except as noted here */

const
    $S = T + 2N$                    /* number of possible round numbers = $O(N)$ */

type
    $Ytype$ = record  $free$: boolean; $rnd$: $0..S$  end   /* stored in one word */

shared variables
    $X$: array[$1..T$] of $0..N-1$;
    $Y$, $Reset$: array[$1..T$] of $Ytype$;
    $Round$: array[$1..S$] of boolean initially $false$;
    $Free$: queue of integers;
    $Inuse$: array[$0..N-1$] of $0..S$ initially 0;
    $Check$: $0..N-1$ initially 0;
    $Acquired$: array[$1..T$] of boolean initially $false$

initially
    $(\forall i : 1 \leq i \leq T :: Y[i] = (true,\ i) \wedge$
        $Reset[i] = (true,\ i)) \wedge$
        $(Free = (T+1) \to (T+2) \to \cdots \to S)$

private variables
    $ptr$: $0..N-1$;     $nextrnd$: $1..S$;     $usedrnd$: $0..S$

process $p$ ::    /* $0 \leq p < N$ */
function $AcquireNode(n$: $1..T)$: $Dtype$
15: $X[n] := p$;
16: $y := Y[n]$;
    if $\neg y.free$ then return $right$ fi;
17: $Y[n] := (false,\ 0)$;
18: $Inuse[p] := y.rnd$;
19: if $X[n] \neq p\ \vee$
20:       $Acquired[n]$ then
        return $left$
    fi;
21: $Round[y.rnd] := true$;
22: if $Reset[n] \neq y$ then
23:     $Round[y.rnd] := false$;
        return $left$
    fi;
24: $Acquired[n] := true$;
    return $stop$

procedure $ReleaseNode(n$: $1..T,\ dir$: $Dtype)$
25: if $dir = right$ then return fi;
26: $Y[n] := (false,\ 0)$;
27: $X[n] := p$;
28: $y := Reset[n]$;
29: $Reset[n] := (false,\ y.rnd)$;
30: if $(dir = stop\ \vee\ \neg Round[y.rnd])$ then
31:     $ptr := Check$;
32:     $usedrnd := Inuse[ptr]$;
33:     if $usedrnd \neq 0$ then
            $MoveToTail(Free,\ usedrnd)$ fi;
34:     $Check := ptr + 1 \bmod N$;
35:     $Enqueue(Free,\ y.rnd)$;
36:     $nextrnd := Dequeue(Free)$;
37:     $Reset[n] := (true,\ nextrnd)$;
38:     $Y[n] := (true,\ nextrnd)$
    fi;
    if $dir = stop$ then
39:     $Round[y.rnd] := false$;
40:     $Inuse[p] := 0$
    fi

procedure $ClearNode(n$: $1..T)$
41: $Acquired[n] := false$
```

Figure 7: Algorithm L: adaptive algorithm with $\Theta(N)$ space complexity. Statements 1–14 are identical to Algorithm U and not shown here.

Statement 35[i] enqueues the current round number for splitter i onto the *Free* queue. Statement 36[i] dequeues a new round number from *Free*. The mechanism explained above (statements 31–34) guarantees that the newly dequeued round number is not used anywhere in the renaming tree. The rest of the algorithm is the same as before. From the discussion so far, we have the following property.

Property 6: If a process p executes statement 16[i] during round $\mathcal{R}(i, r)$ and subsequently reaches statement 21[i], then the following holds until p returns from $AcquireNode(i)$.

> *Either round $\mathcal{R}(i, r)$ continues, or the round number r is contained in Free. Moreover, r is not used for any other splitter.*

Proof: (Formally, Property 6 is implied by invariants (I5)–(I8) and (I15), stated in the appendix.) $\mathcal{R}(i, r)$ may end only if a process q starts a new round at splitter i by executing statement 38[i]. In this case, q enqueues r at statement 35[i].

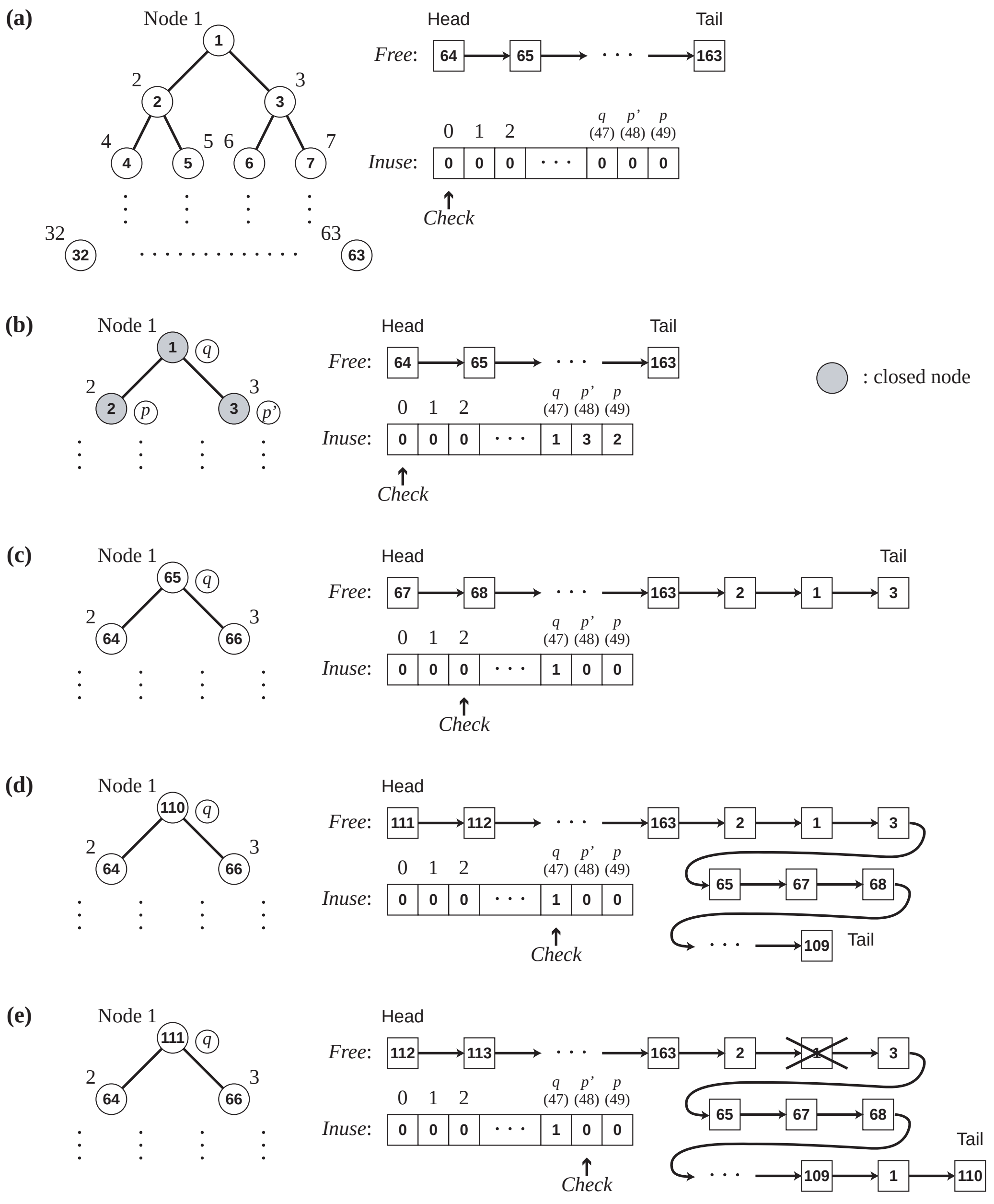


Figure 8: Change of the queue during an example execution. We assume $N = 50$, $L = 5$, $T = 63$, and $S = 163$. **(a)** Initial usage of round numbers. (Round numbers are shown in **boldface**.) **(b)** Processes p $(= 49)$ and p' $(= 48)$ start their entry sections, interact with each other at the root node and stop at nodes 2 and 3, respectively. Nodes 1, 2, and 3 are now closed. Process q $(= 47)$ executes its entry section concurrently with p and p', and is stalled at line 21[1]. Note that we have $Inuse[q] = 1$. **(c)** Process p and p' execute their critical and exit sections, in the order of p and then p'. Process p resets node 2 and 1; p' resets node 3. (Since p' has moved right at node 1, it does not reset node 1. See also Property 4.) Note that round number 1 is now in the *Free* queue, even though it is being used by q. **(d)** While q is being stalled, p executes its critical section 44 times in isolation, advancing *Check* from 3 to 47. (During this period, 44 round numbers (67–110) are dequeued; all but the last one (110) are enqueued back.) **(e)** Process p executes its critical section one more time, reads $Check = 47$ at line 31[1], executes $MoveToTail(Free, 1)$, and then resets node 1. Note that, as long as q remain stalled between lines 19[1]–29[1], round number 1 is moved to the tail by every N^{th} critical-section execution.

We now show that r is not used for any other splitter. For the sake of contradiction, assume that r *is* used by some other splitter. Moreover, assume that r is the first round number in which this happens.

Since each splitter initially has a distinct round number, this may happen only if some process q' dequeues r from *Free* at statement 36[h] (for some splitter h), and then starts round $\mathcal{R}(h, r)$ at statement 38[h]. Since *Free* has length $2N$, *ReleaseNode* must have been executed at least N times since q has enqueued r at statement 35[i]. (Recall that invocations of *ReleaseNode* are protected by `ENTRY`/`EXIT` calls and do not overlap.) Moreover, among the *last* N invocations of *ReleaseNode* (up to and including the invocation by q'), no execution must have invoked $\mathit{MoveToTail}(\mathit{Free}, r)$ at statement 33.

Since *Check* is incremented modulo-N, among these last N invocations of *ReleaseNode*, there exists one, by some process q'', in which statements 31 and 32 are executed while $\mathit{Check} = p$ holds.

First, assume that q'' executes statement 32 *after* p executes statement 18[i]. In this case, q'' subsequently invokes $\mathit{MoveToTail}(\mathit{Free}, r)$ at statement 33, a contradiction. Second, assume that q'' executes statement 32 *before* p executes statement 18[i]. In this case, since q executes *ReleaseNode* before q'' does, q has already executed statements 26[i] and 27[i]. Hence, p must either find $Y[i] = (\mathit{false}, 0)$ at statement 16[i], or find $X[i] \neq p$ at statement 19[i]. (The reasoning for this is the same as in the proof of Property 5.) Thus, p cannot reach statement 21[i], a contradiction. □

Since round numbers are never reused "prematurely," it follows that ALGORITHM L is a correct, starvation-free mutual exclusion algorithm.

Proof of contention sensitivity. We now prove that ALGORITHM L is contention-sensitive. In our proof, it is necessary to track each process's current location in the renaming tree so that we can determine when a process will be deflected left or right from some splitter. The location of a process p during its entry and exit section is depicted in Figure 9. (A formal definition will be given later.) In its entry section, p can only deflect other processes at the splitter it is attempting to acquire. Thus, it has a single location (Figure 9(a)). However, at any given instant inside its exit section, p can deflect other processes at *two* splitters — the splitter that it has acquired in its entry section (which is indicated by $p.\mathit{node}$) and the splitter it is currently resetting (which is indicated by $p.n$ inside *ReleaseNode*; see Figure 9(b)). In the former case, other processes may be deflected because $\mathit{Acquired}[p.\mathit{node}]$ is true (Figure 9(c)). In the latter case, p may deflect other processes by changing the value of Y, X, or *Reset* (statements 26–29 and 37; Figure 9(d)). In order to facilitate the following discussion, we associate a *shadow process* with an identifier of $p + N$ with each process p. When p is in its entry section, p and $p + N$ are always located at the same splitter indicated by $p.\mathit{node}$. However, when p is in its exit section, we say that p is located at the splitter indicated by $p.n$, while $p + N$ is located at the splitter indicated by $p.\mathit{node}$. (Thus, p ascends the renaming tree in its exit section, while $p + N$ remains stationary until the exit section terminates.)

Definition: We define the *contention* of splitter i, denoted $C(i)$, as the number of processes p (shadow processes included) that are located at a splitter (or a child of a leaf splitter) in the subtree rooted at i.

Consider a process p in its entry section. At any given state t, if p is located in the subtree rooted at i, then we define $\mathsf{PC}[p, i]$, the *point contention of splitter i experienced by p*, as the maximum value of $C(i)$ since p descended to splitter i up to state t. In particular, when p moves down to splitter i, $\mathsf{PC}[p, i]$ is initialized to be $C(i)$. From that point onward, $\mathsf{PC}[p, i]$ tracks the point contention of the subtree rooted at i as seen by process p. (That is, whenever $C(i)$ changes, we assign "$\mathsf{PC}[p, i] := \max(\mathsf{PC}[p, i],\ C(i))$.") □

For example, consider a non-root splitter $i \geq 2$. If a process p in its entry section descends to splitter i, then $C(i)$ increases by two (because p and $p + N$ descend together). When p ascends from i to i's parent in its exit section, $C(i)$ decreases by one. Finally, when p finishes its exit section (by executing statement 41), both p and $p + N$ leave the renaming tree. Since $p + N$ has been located at p's acquired node (which is inside the subtree rooted at i), this reduces $C(i)$ by one again. On the other hand, $C(1)$ increases by two when p starts its execution, and decreases by two when p finishes its execution. Thus, $C(1)$ equals twice the actual point contention at any given state.

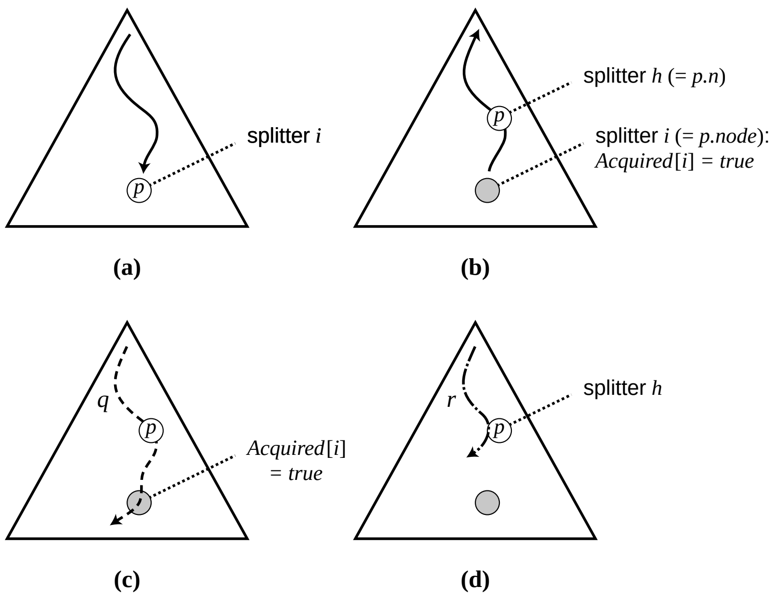


Figure 9: The location of a process p during its entry and exit section. **(a)** p acquires splitter i: both p and $p+N$ are located at i. **(b)** In its exit section, p resets each node h in its path by calling $ReleaseNode(h)$. p is located at h, while $p+N$ is located at i. **(c)** Another process q in its entry section descends to splitter i, and is deflected left because it finds $Acquired[i] = true$ (statement 20[i]). **(d)** Yet another process r in its entry section descends to splitter h, and is deflected left or right because p updated $Y[h]$, $X[h]$, or $Reset[h]$ (statements 26[h].p, 27[h].p, 29[h].p, and 37[h].p).

Proof strategy. In Property 8 below, we show that, whenever a process p is deflected to a node i, there exists another process (or a shadow process) q that causes p's deflection. Moreover, q's location is always such that q is inside the subtree rooted at $\lfloor i/2 \rfloor$ (*i.e.*, the parent of i), but *not* inside the subtree rooted at i. (See Figure 10.)[6]

By definition, this implies that $C(i) < C(\lfloor i/2 \rfloor)$ holds at that instant. If this inequality *were* an invariant, then we could easily prove contention sensitivity, by induction over the path p has taken. That is, if p eventually stops at a node i (= i_l) after visiting nodes i_0 (= 1), i_1, i_2, …, i_{l-1}, then we would have the following identity:

$$C(1) = C(i_0) > C(i_1) > \cdots > C(i_l) = C(i) > 0,$$

and hence, we would have that point contention is $C(1)/2 > l/2$. ($C(i) > 0$ holds because p itself contributes to $C(i)$.)

Unfortunately, $C(i) < C(\lfloor i/2 \rfloor)$ is *not* an invariant: it can be falsified, for example, when process q finishes its exit section and removes itself from the tree. However, if we consider $\mathsf{PC}[p,i]$ instead of $C(i)$, then Property 9 below shows that $\mathsf{PC}[p,i] < \mathsf{PC}[p,\lfloor i/2 \rfloor]$ *is* an invariant. Thus, by repeatedly applying this property over the path p has taken, we can prove contention sensitivity. □

For the sake of the proof, it is necessary to formally define the "location" of process p. Unfortunately, the definition is rather contrived, and is best described with the introduction of an auxiliary variable $\mathsf{Loc}[p]$.

In Figure 11, Algorithm L is shown with these added auxiliary variables. ($\mathsf{PC}[p,i]$ is already described; Dist is used only for the formal proof in the appendix, and is not discussed here.) We have marked the

[6]Our proof critically depends on the fact that, even if a process p acquires node i, the proper ancestors of node i can be reopened by other processes while p is still active. If this were not the case, then the algorithm would not be contention-sensitive. For example, the following "worst-case" scenario might happen: a process p may stop at the rightmost node i of level l (by moving right at every level), during a period of high contention. While p is stalled, all other active processes finish execution. If all proper ancestors of node i remain closed, then yet another process q may enter the tree, read $Y = (false, 0)$ and move right at each node it accesses (which must be ancestors of i), and eventually stop at level $l+1$. Thus, q executes $\Omega(l)$ operations, even though it experiences contention of only two. However, in Algorithm L, when a period of high contention ends, the exiting processes reopen the proper ancestors of node i. Therefore, such a hypothetical "worst-case" scenario does not happen.

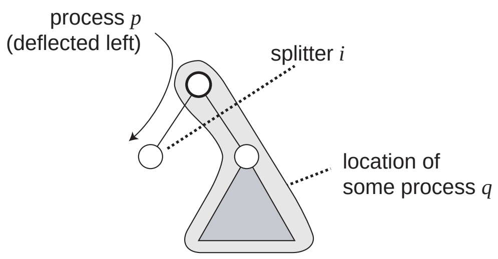


Figure 10: Deflection of a process p to splitter i. If p is deflected left (right), then some process q (or $q + N$) is located either at i's parent or in the subtree rooted at i's right (left) sibling.

lines of code that refer to auxiliary variables with a dash "—" to make them easier to distinguish. These dashed lines are assumed to execute sequentially (For example, when a process p executes statement 25, p assigns $\mathsf{Loc}[p] := p.n$ if and only if $p.dir \neq right$ holds. Similarly, the two dashed lines after line 1 are executed sequentially.) In addition, we assume that a dashed line is executed immediately after the preceding numbered statement, as a part of a single atomic statement execution. (For example, the dashed line after line 14 is executed as a part of either statement 14 or 41.)

The auxiliary procedure $\mathsf{UpdateLoc}(P, i)$ is called when a process enters its entry section (statement 1) and when a set P of processes moves to splitter i while in their entry sections (statements 15, 16, 20, 27, and 29). This procedure updates the value of $\mathsf{Loc}[q]$ and $\mathsf{Loc}[q+N]$ to be i for each $q \in P$ (recall that both should have the same value while q is in its entry section), and sets the value of $\mathsf{PC}[q, i]$ as the current contention of the subtree rooted at i. It also updates the PC values of other processes to reflect the movement of processes in P.

```
procedure UpdateLoc($P \subseteq \{0..N-1\}$, $i$: $1..T$)
u1: for all $q \in P$ do $\mathsf{Loc}[q]$, $\mathsf{Loc}[q+N] := i,\ i$ od;
u2: for all $q \in P$ do $\mathsf{PC}[q, i] := C(i)$ od;
u3: /* adjust $\mathsf{PC}[r, h]$ for every process $r$ in its entry section,
           and for every node $h$ on the path $r$ has taken so far */
    od
```

Note that a process p's location may change even *before* p actually discovers that it has to move down (left or right). For example, assume that processes p and q try to acquire some node i, initially open. Consider the following scenario: **(i)** p executes statement 15[i], and is stalled at line 16; **(ii)** meanwhile, q executes statement 15[i]. By executing statement 15[i], q eliminates any possibility that p may stop at node i. (p will either move right at statement 16[i], or move left at statement 19[i]. These are the only possibilities.) In order to represent this fact, q actually changes p's location from i to $2i$ by calling UpdateLoc. (See Figure 12(a).)

It is possible that q may have guessed wrong. If q executes statement 17[i] before p executes statement 16[i], then p is actually deflected right, not left. Thus, in this case, p's location changes from i to $2i$ to $2i + 1$. (See Figure 12(b).) Although this seems rather counterintuitive, it is necessary in order to prove Property 8.

We now consider each statement that may change the location of a process p.

- **Statement 1.*p*.** As p starts its entry section, both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p + N]$ are changed from 0 (not in the tree) to 1 (the root node).

- **Statement 25.*p*.** This statement either returns immediately or changes $\mathsf{Loc}[p]$ to i ($= p.n$) before resetting node i. (See Figure 9(b).)

  Note that statement 25.p always changes $\mathsf{Loc}[p]$ from some node to its ancestor (skipping the nodes at which p has moved right during its entry section), since p is traversing its path upward (the **for** loop at line 9). In particular, it cannot update $\mathsf{PC}[q, h]$ for any process q (in its entry section) and any node h.

```
/* all constant, type, and variable declarations are as defined in Figure 7 except as noted here */
shared auxiliary variables
    Loc: array[0..2N − 1] of 0..2T + 1 initially 0;
    Dist: array[1..S] of (⊥, 0..S − 1);
    PC: array[0..N − 1, 1..T] of 0..2N initially 0

private auxiliary variable
    m: 1..T

initially
    (∀j : 1 ≤ j ≤ T :: Dist[j] = ⊥) ∧
        (∀j : T < j ≤ S :: Dist[j] = j − T − 1)

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  node, level := 1, 0;
    — for m := 1 to T do PC[p, m] := 0 od;
    — UpdateLoc({p}, 1);

    /* descend renaming tree */
    repeat
2:      dir := AcquireNode(node);
3:      path[level] := (node, dir);
        if dir = left then
            level, node := level + 1, 2 · node
        elseif dir = right then
            level, node := level + 1, 2 · node + 1
        fi
    until (level > L) ∨ (dir = stop);

    if level ≤ L then      /* got a name */
        /* compete in renaming tree, then 2-proc. alg. */
        for j := level downto 0 do
4:          ENTRY_3(path[j].node, path[j].dir)
        od;
5:      ENTRY_2(0)
    else     /* did not get a name */
        /* compete in overflow tree, then 2-proc. alg. */
6:      ENTRY_N(p);
7:      ENTRY_2(1)
    fi;

8:  Critical Section;

    /* reset splitters */
    for j := min(level, L) downto 0 do
        n, dir := path[j].node, path[j].dir;
9:      ReleaseNode(n, dir)
    od;

    /* execute appropriate exit sections */
    if level ≤ L then
10:     EXIT_2(0);
        for j := 0 to level do
11:         EXIT_3(path[j].node, path[j].dir)
        od;
12:     ClearNode(node)
    else
13:     EXIT_2(1);
14:     EXIT_N(p)
    fi;
    — Loc[p], Loc[p + N] := 0, 0
od

function AcquireNode(n: 1..T): Dtype
15: X[n] := p;
    — UpdateLoc({q |
          q@{16..19} ∧ q ≠ p ∧ q.n = n}, 2n);
16: y := Y[n];
    if ¬y.free then
    —   UpdateLoc({p}, 2n + 1);
        return right
    fi;
17: Y[n] := (false, 0);
18: Inuse[p] := y.rnd;
19: if X[n] ≠ p ∨
20:       Acquired[n] then
    —   UpdateLoc({p}, 2n);
        return left
    fi;
21: Round[y.rnd] := true;
22: if Reset[n] ≠ y then
23:     Round[y.rnd] := false;
        return left
    fi;
24: Acquired[n] := true;
    return stop

procedure ReleaseNode(n: 1..T, dir: Dtype)
25: if dir = right then return fi;
    — Loc[p] := n;
26: Y[n] := (false, 0);
27: X[n] := p;
    — UpdateLoc({q | q@{16..19} ∧ q.n = n}, 2n);
28: y := Reset[n];
29: Reset[n] := (false, y.rnd);
    — UpdateLoc({q | q@{17..22} ∧ q.n = n}, 2n);
30: if (dir = stop ∨ ¬Round[y.rnd]) then
31:     ptr := Check;
32:     usedrnd := Inuse[ptr];
33:     if usedrnd ≠ 0 then
            MoveToTail(Free, usedrnd)
        fi;
34:     Check := ptr + 1 mod N;
35:     Enqueue(Free, y.rnd);
36:     nextrnd := Dequeue(Free);
37:     Reset[n] := (true, nextrnd);
38:     Y[n] := (true, nextrnd)
    fi;
    if dir = stop then
39:     Round[y.rnd] := false;
40:     Inuse[p] := 0
    fi

procedure ClearNode(n: 1..T)
41: Acquired[n] := false
```

Figure 11: ALGORITHM L with auxiliary variables added.

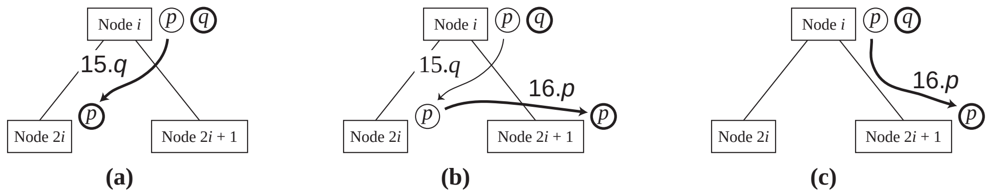


Figure 12: Ways in which a process may change its location. **(a)** p executes statement 15, and then q executes statement 15. **(b)** After (a), q executes statements 16 and 17, and then p executes statement 16, reads $Y[i] = (\mathit{false}, 0)$, and moves right. **(c)** In a separate execution, q first executes statements 15–17, and then p executes statements 15 and 16, reads $Y[i] = (\mathit{false}, 0)$, and moves right.

- **The line after statement 14.** This line can be executed as a part of either statement 14.p or 41.p. In either case, p finishes its exit section, and both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are reset to 0.

Before considering the remaining cases, we state the following property, whose proof will be given later.

Property 7: If $p@\{15\}$ and $p.n = i$, then we have $\mathsf{Loc}[p] = \mathsf{Loc}[p+N] = i$. □

In other words, when process p starts to execute $\mathit{AcquireNode}(i)$, its location equals i.

We now consider the remaining statements that may update p's location, each of which occurs while p is executing inside $\mathit{AcquireNode}(i)$.

- **Statement 16[*i*].*p*.** As explained before, there are two possible cases.
  - Both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are changed from i to $2i+1$ (see Figure 12(c)).
  - Both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are changed from $2i$ to $2i+1$ (see Figure 12(b)).
- **Statement 19[*i*].*p*.** If p reads $X[i] \neq p$, then some process, say, q, must have executed statement 15[i] or 27[i], after p executed statement 15[i] but before p executes statement 19[i]. In either case, q changes p's location to $2i$ by executing 15[i].q or 27[i].q. It follows that, if p reads $X[i] \neq p$, then $\mathsf{Loc}[p] = \mathsf{Loc}[p+N] = 2i$ already holds before the execution of 19[i].p. Thus, statement 19[i].p does not update p's location.
- **Statement 20[*i*].*p*.** Again, there are two possible cases.
  - Both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are changed from i to $2i$ (see Figure 12(a)).
  - Both $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are already equal to $2i$, and do not change.
- **Statements 15[*i*].*q* and 27[*i*].*q*, where *q* is any arbitrary process.** These statements change $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ from i to $2i$ (unless they are already equal to $2i$), if executed when $p@\{16..19\} \wedge p.n = i$ holds.

  In this case, as explained before, q eliminates any possibility that p may acquire node i. The change of p's location signifies this fact.
- **Statement 29[*i*].*q*, where *q* is any arbitrary process.** This statement may update $\mathsf{Loc}[p]$ (and $\mathsf{Loc}[p+N]$) only if executed when $p@\{17..22\} \wedge p.n = i$ holds. In this case, we may assume that p has read $Y[i] = (\mathit{true}, r)$ at statement 16[i], for some round number r. (Otherwise, p would not have reached statement 17[i].)

  We now show that p eventually moves left at node i. (Clearly, p cannot move right, because it has reached statement 17[i].) For the sake of contradiction, assume that p eventually stops at node i. In order for that to happen, p must read $\mathit{Reset}[i] = (\mathit{true}, r)$ at statement 22[i]. Thus, we have the following sequence of events: **(E1)** p reads $Y[i] = (\mathit{true}, r)$ at statement 16[i], **(E2)** q writes $\mathit{Reset}[i] = (\mathit{false}, r)$ at statement 29[i], and **(E3)** p reads $\mathit{Reset}[i] = (\mathit{true}, r)$ at statement 22[i].

Hence, there exists a process q' (which may be q or some other process) that writes $Reset[n] := (true, r)$ by executing statement $37[i]$, after (E2) but before (E3). Since executions of *ReleaseNode* are serialized, this in turn implies that q' dequeued r from *Free* (by executing statement $36[i]$) after (E2) but before (E3).

However, by Property 6, the following must continuously hold between events (E1) and (E3): *either round $\mathcal{R}(i, r)$ is active, or r is contained in Free.* We have thus reached a contradiction.

It follows that, as well as statements $15[i].q$ and $27[i].q$, statement $29[i].q$ eliminates any possibility that p may stop at node i. Thus, $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ are changed from i to $2i$ (unless they are already equal to $2i$).

The discussion so far can be formalized as the following invariants. (In (I55) below, we define that a node is an ancestor (descendant) of itself.)

invariant $p@\{0,1\} \Rightarrow \mathsf{Loc}[p] = 0 \wedge \mathsf{Loc}[p+N] = 0$ (I48)

invariant $p@\{2,15\} \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node$ (I49)

invariant $p@\{16\} \Rightarrow \big(X[p.node] = p \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node\big) \wedge$
$\big(X[p.node] \neq p \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node\big)$ (I50)

invariant $p@\{17..19\} \wedge X[p.node] = p \wedge Reset[p.node] = p.y \Rightarrow$
$\mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node$ (I51a)

invariant $p@\{17..19\} \wedge \neg(X[p.node] = p \wedge Reset[p.node] = p.y) \Rightarrow$
$\mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node$ (I51b)

invariant $p@\{20..22\} \wedge Reset[p.node] = p.y \Rightarrow$
$\mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node$ (I51c)

invariant $p@\{20..22\} \wedge \neg Reset[p.node] = p.y \Rightarrow$
$\mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node$ (I51d)

invariant $p@\{3\} \Rightarrow \big(p.dir = stop \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node\big) \wedge$
$\big(p.dir = left \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node\big) \wedge$
$\big(p.dir = right \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node + 1 \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node + 1\big)$ (I52)

invariant $p@\{4..14, 25..41\} \Rightarrow \mathsf{Loc}[p+N] = p.node$ (I53)

invariant $p@\{4..8\} \Rightarrow \mathsf{Loc}[p] = p.node$ (I54)

invariant $p@\{9, 25\} \Rightarrow \mathsf{Loc}[p]$ is a descendant of $p.n$ and an ancestor of $p.node$ (I55)

invariant $p@\{26..40\} \Rightarrow \mathsf{Loc}[p] = p.n$ (I56)

(The appendix contains formal proofs of these invariants. Note that invariants (I51a)–(I51d) are replaced by a single invariant (I51) in the appendix, which is equivalent to the conjunction of (I51a)–(I51d).)

We now give the proof of Property 7.

Proof of Property 7. This property follows trivially from invariant (I49). □

From the discussion so far, it follows that there are three ways in which $\mathsf{Loc}[p]$ (and $\mathsf{Loc}[p+N]$) may change when p is in its entry section (in particular, during the execution of *AcquireNode*(i) for some i): **(i)** from node i to its left child $(2i)$; **(ii)** from the left child $(2i)$ to the right child $(2i+1)$; **(iii)** from node i to its right child $(2i+1)$. (Each case is depicted in Figure 12.)

We are now in a position to state and prove Property 8, which is the crux of our contention-sensitivity proof.

Property 8: Consider a process p in its entry section. If p's location changes to a non-root splitter h, then there exists another process (or shadow process) q that is currently located at either h's parent, or in a subtree rooted at h's sibling. (In particular, $C(h) < C(\lfloor h/2 \rfloor)$ holds at that instant.)

Proof: Let $i = \lfloor h/2 \rfloor$. From the preceding discussion, it follows that p's location changes only by the execution of one of the statements $16[i].p$, $20[i].p$, $15[i].q$, $27[i].q$, and $29[i].q$, where q is any arbitrary process.

Statements $15[i].q$, $27[i].q$, and $29[i].q$ trivially satisfy Property 8, since we have $\mathsf{Loc}[q] = i$ at each case. (As for $15[i].q$, recall Property 7.)

Statement $20[i].p$ may establish $\mathsf{Loc}[p] = h$ $(= 2i)$ only if executed when $\mathit{Acquired}[i]$ is true, which may happen only if some process q has acquired node i. In this case, as shown in Figure 9(b), we have $\mathsf{Loc}[q + N] = i$.

Finally, since $Y[i].\mathit{Free}$ is initially *true*, statement $16[i].p$ may establish $\mathsf{Loc}[p] = h$ $(= 2i+1)$ only if some process q has executed either statement $17[i]$ or $26[i]$ before $16[i].p$ is executed.

First, assume that q has executed statement $17[i]$. In this case, q either has acquired splitter i, or has been deflected left. In either case, q is located at splitter i or in the subtree rooted at i's left child. Moreover, if q ascends to i's parent in its exit section, then q must either reopen splitter i by establishing $Y[i].\mathit{free} = \mathit{true}$ at statement $38[i]$, or read $\mathit{Round}[\mathit{Reset}[i].\mathit{rnd}] = \mathit{true}$ at statement $30[i]$. However, the latter case may arise only if some other process q' has executed statement $21[i]$ (before $30[i].q$ is executed). Moreover, q' has *not* yet executed either statement $23[i]$ or $39[i]$. (Note that, since q' has executed statement $21[i]$, q' must eventually either execute statement $23[i]$ and move left, or acquire node i and later execute statement $39[i]$. In either case, $\mathit{Round}[\mathit{Reset}[i].\mathit{rnd}]$ becomes *false* again.)

Therefore, before and after $30[i].q$ is executed, process q' is located at splitter i or in the subtree rooted at i's left child. Continuing in the same way, it follows that the following property continues to hold while $Y[i] = (\mathit{false}, 0)$ holds: *some process other than p is located at splitter i or in the subtree rooted at i's left child.* (Formally, this property follows from invariant (I57), stated in the appendix.)

Second, assume that q has executed statement $26[i]$. In this case, when p executes statement $16[i]$, either q is executing within statements $27[i]$–$38[i]$ (in which case q is located at splitter i), or it returned from *ReleaseNode* without executing statements $30[i]$–$40[i]$. (Note that statement $38[i].q$ falsifies $Y[i] = (\mathit{false}, 0)$.) However, the latter case happens only if q reads $\mathit{Round}[\mathit{Reset}[i].\mathit{rnd}] = \mathit{true}$ at statement $30[i]$, which in turn happens only if (as shown above) there exists yet another process q' that has executed statement $21[i]$. The rest of the reasoning is the same as the preceding paragraph. □

The following property states that, even if contention $C(h)$ may vary over time, the point contention $\mathsf{PC}[p, h]$ (experienced by a process p) is always strictly lower than $\mathsf{PC}[p, \lfloor h/2 \rfloor]$, the point contention of node h's parent.

Property 9: If a process p in its entry section is located in a subtree rooted at splitter $h \geq 2$, then $\mathsf{PC}[p, h] < \mathsf{PC}[p, \lfloor h/2 \rfloor]$.

Proof: Define $i = \lfloor h/2 \rfloor$. First, consider the time when p moves to splitter h. (Note that $\mathsf{PC}[p, h]$ is not defined before this time.) By Property 8, we have $C(h) < C(i)$ at that time. By definition, $\mathsf{PC}[p, i] \geq C(i)$ holds. Since $\mathsf{PC}[p, h]$ is initialized to $C(h)$, the property follows.

Second, consider the case when $\mathsf{PC}[p, h]$ or $\mathsf{PC}[p, i]$ is changed while p is already inside the subtree rooted at h. Since $\mathsf{PC}[p, h]$ and $\mathsf{PC}[p, i]$ may only increase, it suffices to consider the case when $\mathsf{PC}[p, h]$ increases. However, this may happen only if some other process q descends to splitter h. By Property 8 again, at that time, $C(h) < C(i)$ holds. Therefore, if $\mathsf{PC}[p, h]$ is updated to the new value of $C(h)$, then we still have $\mathsf{PC}[p, i] \geq C(i) > C(h) = \mathsf{PC}[p, h]$, and hence the property follows. □

We now prove contention sensitivity. (For a full proof, see invariant (I61) in the appendix.) Assume that a process p has reached splitter i at level $l = \mathsf{lev}(i)$. By repeatedly applying Property 9 over all ancestors of

i, we have

$$\begin{aligned}
\mathsf{PC}[p,1] &= \mathsf{PC}[p, p.path[0].node] \\
&> \mathsf{PC}[p, p.path[1].node] \\
&\quad \vdots \\
&> \mathsf{PC}[p, p.path[p.level-1].node] \\
&> \mathsf{PC}[p, p.node] = \mathsf{PC}[p,i] \\
&> 0.
\end{aligned}$$

Given the length of this sequence, we have $\mathsf{PC}[p,1] > l$, which implies that p has experienced contention at least $(l+1)/2$ at some point since it started execution. It follows that ALGORITHM L is contention-sensitive.

It should be noted that the order in which *ReleaseNode* is called is critical in our proof. In particular, the proof of Property 8 exploits the fact that a process in its exit section always moves up in the tree. Informally, this ensures that, given a process p in its exit section and a process q in its entry section, p may deflect q (by executing *ReleaseNode*) at most once during its exit-section execution.[7]

The space complexity of ALGORITHM L is clearly $\Theta(N)$, if we ignore the space required to implement the `ENTRY` and `EXIT` routines. (Although each process has a $\Theta(\log N)$ *path* array, these arrays are actually unneeded, as simple calculations can be used to determine the parent and children of a splitter.) If the `ENTRY`/`EXIT` routines are implemented using Yang and Anderson's arbitration-tree algorithm [25], then the overall space complexity is actually $\Theta(N \log N)$. This is because in Yang and Anderson's algorithm, each process needs a distinct spin location for each level of the arbitration tree. However, as shown in [17], it is quite straightforward to modify the arbitration-tree algorithm so that each process uses the same spin location at each level of the tree. This modified algorithm has $\Theta(N)$ space complexity.

Let us consider the exact number of shared variables used. Although we defined $L = \lfloor \log N \rfloor$ and $T = 2^{L+1} - 1$ for simplicity, the algorithm clearly remains adaptive if we let $L = \alpha \log N - 1$ (and thus $T = N^\alpha - 1$), for some fixed constant α ($0 < \alpha \le 1$). Assuming this, and counting the number of shared variables declared in Figure 7, we have the following:

$$\begin{aligned}
\text{the number of shared variables} &= T \ \{\text{for } X\} + T \ \{\text{for } Y\} + T \ \{\text{for } \textit{Reset}\} + S \ \{\text{for } \textit{Round}\} + \\
&\quad N \ \{\text{for } \textit{Inuse}\} + 1 \ \{\text{for } \textit{Check}\} + T \ \{\text{for } \textit{Acquired}\} + \\
&\quad (2S+1) \ \{\text{for } \textit{Free}\} \\
&= 7N + 7T + 2 \\
&= 7N + 7N^\alpha - 5.
\end{aligned}$$

(Since *Free* holds unique integers in the range of $1..S$, it can be implemented, as a doubly-linked circular list, with an array of S entries, each entry composed of a forward link and a backward link, plus a single pointer to the head of the queue. Also recall $S = T + 2N$.)

Additionally, if implemented using the $\Theta(N)$ arbitration tree algorithm [17] (which requires $5X + N$ shared variables for a tree with X nodes), $\mathtt{ENTRY}_2$, $\mathtt{ENTRY}_3$, and $\mathtt{ENTRY}_N$ require $N+5$, $N+10T$, and $6N-5$ shared variables, respectively. (Note that each instance of $\mathtt{ENTRY}_3$ requires *two* instances of a two-process algorithm.) Therefore, ALGORITHM L uses total $14N + 17T + 2 = 14N + 17N^\alpha - 15$ shared variables. (It can be somewhat improved by sharing spin variables used in `ENTRY`/`EXIT` subroutines, at the expense of increased remote memory references.) Each variable holds at most $S + 1 = 2N + N^\alpha$ distinct values.

We conclude this section by stating our main theorem.

[7]On the other hand, if we change ALGORITHM L such that *ReleaseNode* is now called in descending order, then the following "worst-case" scenario might happen: a process p may stop at the leftmost node i of level l, during a period of high contention. While p is stalled, all other active processes finish execution. Yet another process q then enters the tree, and then p and q execute in "lockstep" until both reach node i, with p's execution of *ReleaseNode* causing q to move left at each node. In a sense, this scenario is similar to that of Footnote 6.

Theorem 1 *N-process mutual exclusion can be implemented under read/write atomicity with RMR time complexity $O(min(k, \log N))$ and $14N+17N^{\alpha}-15$ shared variables, for some fixed constant α $(0 < \alpha \leq 1)$.* $\square$

3 Concluding Remarks

We have presented an adaptive algorithm for mutual exclusion under read/write atomicity in which all waiting is by local spinning. This is the first read/write algorithm that is adaptive under the RMR time complexity measure. Our algorithm has $\Theta(N)$ space complexity, which is clearly optimal.

In recent work, we established a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for mutual exclusion algorithms based on reads, writes, or comparison primitives such as test-and-set or compare-and-swap [8]. In another work, we also showed that that it is impossible to construct an adaptive algorithm with $o(k)$ RMR time complexity [16]. In particular, we proved the following:

> *For any k, there exists some N such that, for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.*

One may wonder whether a $\Omega(\min(k, \log N/\log\log N))$ lower bound follows from these two results. Unfortunately, that is not the case, since we have shown that $\Omega(k)$ RMR time complexity is required *provided* N is sufficiently large. We conjecture that $\Omega(\log N)$ is a tight lower bound (under the RMR measure) for mutual exclusion algorithms under read/write atomicity, and that $\Omega(\min(k, \log N))$ is also a tight lower bound for adaptive mutual exclusion algorithms under read/write atomicity (in which case the algorithm of this paper is optimal). We leave these questions for further study.

Acknowledgements: Discussions with Gary Peterson regarding the space complexity of adaptive mutual exclusion algorithms led us to develop Algorithm L. We also acknowledge the referees of this paper for their hard work in carefully checking our proofs.

References

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

[2] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 262–272. IEEE, October 1999.

[3] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[4] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 12–21. IEEE, 1992.

[5] J. Anderson. A fine-grained solution to the mutual exclusion problem. *Acta Informatica*, 30(3):249–265, May 1993.

[6] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.

[7] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

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

[9] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM, August 1995.

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

[11] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. ACM, July 2000.

[12] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 277–286. ACM, July 1998.

[13] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

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

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

[16] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15. Lecture Notes in Computer Science 2180, Springer-Verlag, October 2001.

[17] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, September 2002.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

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

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

[21] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[22] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 164–178. Lecture Notes in Computer Science 1914, Springer-Verlag, October 2000.

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

[24] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. ACM, August 1992.

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

Appendix: Correctness Proof for Algorithm L

In this appendix, we formally prove that Algorithm L satisfies the mutual exclusion property (at most one process executes its critical section at any time). In addition, we establish an invariant that implies that the algorithm is contention-sensitive. (The algorithm is easily seen to be starvation-free if the underlying algorithms used to implement the `ENTRY` and `EXIT` calls are starvation-free.)

We now define several terms that will be used in the proof. Unless stated otherwise, we assume that i and h range over $\{1, \ldots, T\}$, and that p and q range over $\{0, \ldots, N-1\}$.

Definition: We say that a process p is *a candidate to acquire splitter* i $(1 \leq i \leq T)$ if the condition $A(p, i)$, defined below, is true.

$$\begin{aligned} A(p,i) \;\equiv\; & p.node = i \;\wedge\; p@\{3..5, 8..14, 17..22, 24..41\} \;\wedge \\ & \big(p@\{17..19\} \;\Rightarrow\; X[i] = p\big) \;\wedge \\ & \big(p@\{17..22\} \;\Rightarrow\; Reset[i] = p.y\big) \;\wedge \\ & \big(p@\{3\} \;\Rightarrow\; p.dir = stop\big) \end{aligned}$$ □

By definition, if p is the winner of some round $\mathcal{R}(i, r)$ of splitter i, then p is also a candidate to acquire splitter i.

Definition: We define $i \xrightarrow{*} h$ to be true, where each of i and h $(1 \leq i, h \leq 2T + 1)$ is either a splitter or a "child" of a leaf splitter, if $i = h$ or if h is a descendent of i in the renaming tree. (When a process "falls off the end" of the renaming tree, it moves to level $L + 1$. The actual leaves of the renaming tree are at level L.) Formally, $\xrightarrow{*}$ is the transitive closure of the relation $\{(i, i),\ (i, 2i),\ (i, 2i + 1) \mid 1 \leq i \leq T\}$. □

Definition: We define $\mathsf{lev}(i)$ to be the level of splitter i in the renaming tree, *i.e.*, $\mathsf{lev}(i) = \lfloor \log i \rfloor$. □

Our proof makes use of a number of auxiliary variables, as shown in Figure 11. They are as follows.

- $\mathsf{Loc}[p]$ is the location of process p within the renaming tree, where $0 \leq p < 2N$. Note that $p.node \neq \mathsf{Loc}[p]$ is possible if p is *not* a candidate to acquire splitter $p.node$. (The exact value of $\mathsf{Loc}[p]$ is given in invariants (I48)–(I56), stated later.)
- $\mathsf{Dist}[r]$ specifies the distance of round number r from the head of *Free*. If r is not in *Free*, then $\mathsf{Dist}[r] = \perp$. $\mathsf{Dist}[r]$ is assumed to be updated within the procedures *Enqueue*, *Dequeue*, and *MoveToTail*.

For the convenience of readers, we restate the definitions of contention $C(i)$, and the auxiliary array PC, which were given in Section 2.4.

Definition: We define the *contention of splitter* i, denoted $C(i)$, as the number of processes p such that $\mathsf{Loc}[p]$ equals a splitter (or a child of a leaf splitter) in the subtree rooted at i. Formally,

$$C(i) \;\equiv\; \big|\{p : 0 \leq p < 2N :: i \xrightarrow{*} \mathsf{Loc}[p]\}\big|.$$ □

- $\mathsf{PC}[p, i]$ is the point contention experienced by process p in its entry section while at splitter i or one of its descendents in the renaming tree. In particular, when p moves down to splitter i, $\mathsf{PC}[p, i]$ is initialized to be $C(i)$, the contention of splitter i. From that point onward, $\mathsf{PC}[p, i]$ tracks the point contention of the subtree rooted at i as seen by process p, *i.e.*, the maximum value of $C(i)$ encountered since p moved down to splitter i. $\mathsf{PC}[p, i]$ is not used if process p is outside its entry section or if $i \xrightarrow{*} \mathsf{Loc}[p]$ is false.

The auxiliary procedure $\mathsf{UpdateLoc}(P, i)$ is called when a process enters its entry section (statement 1) and when a set P of processes moves to splitter i while in their entry sections (statements 15, 16, 20, 27, and 29). This procedure updates the value of $\mathsf{Loc}[q]$ and $\mathsf{Loc}[q+N]$ to be i for each $q \in P$ (recall that both should have the same value while q is in its entry section), and sets the value of $\mathsf{PC}[q, i]$ as the current contention of the subtree rooted at i. It also updates the PC values of other processes to reflect the movement of processes in P.

```
procedure UpdateLoc($P \subseteq \{0..N-1\}$, $i$: $1..T$)
u1: for all $q \in P$ do  $\mathsf{Loc}[q],\ \mathsf{Loc}[q+N] := i,\ i$  od;
u2: for all $q \in P$ do  $\mathsf{PC}[q,i] := C(i)$  od;
u3: for all $r,\ h$ s.t. $r@\{2,3,15..24\} \ \wedge\ h \xrightarrow{*} \mathsf{Loc}[r]$ do
        if $\mathsf{PC}[r,h] < C(h)$ then $\mathsf{PC}[r,h] := C(h)$ fi
    od
```

Notice that the value of Loc array is changed directly at statements $14.p$, $25.p$, and $41.p$ (by our atomicity assumption, each of $14.p$ and $41.p$ establishes $p@\{0\}$ and includes the assignments to $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+n]$ that appear after statement $14.p$). Because these statements are not within p's entry section, there is no need to update $\mathsf{PC}[p,i]$ (for any i). In addition, statement $25.p$ updates $\mathsf{Loc}[p]$ to move from a splitter to its ancestor, and statements $14.p$ and $41.p$ reinitialize $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$ to 0 (*i.e.*, p is no longer within the renaming tree). Thus, contention does not increase within any subtree when these statements are executed.

List of Invariants

We will establish the mutual-exclusion and contention-sensitivity properties 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.

(I) Invariants that give conditions that must hold if a process p is a candidate to acquire splitter i.

invariant $A(p,i) \ \wedge\ \big(p@\{3..8,18..22,24\} \ \vee\ (p@\{9,25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow\ Y[i] = (\mathit{false}, 0)$ (I1)

invariant $A(p,i) \ \wedge\ \big(p@\{3..8,22,24\} \ \vee\ (p@\{9,25..38\} \ \wedge\ p.n = i)\big) \ \Rightarrow\ \mathit{Round}[p.y.rnd] = \mathit{true}$ (I2)

invariant $A(p,i) \ \wedge\ \big(p@\{3..8,24\} \ \vee\ (p@\{9,25..37\} \ \wedge\ p.n = i)\big) \ \Rightarrow\ \mathit{Reset}[i].rnd = p.y.rnd$ (I3)

invariant $\big(\exists p :: A(p,i) \ \wedge\ p@\{3..14,25..41\}\big) = \big(\mathit{Acquired}[i] = \mathit{true}\big)$ (I4)

(II) Invariants that prevent "interference" of *Round* entries. These invariants are used to show that if $p@\{20..23\}$ holds and process p is *not* a candidate to acquire splitter $p.node$, then either $p.y.rnd$ is not in the *Free* queue, or it is "trapped" in the tail region of the queue. Therefore, there is no way $p.y.rnd$ can reach the head of *Free* and get assigned to another splitter.

invariant $p@\{17..19\} \ \wedge\ X[p.node] = p \ \Rightarrow$
$\quad \mathit{Reset}[p.node] = p.y \ \wedge\ \mathsf{Dist}[p.y.rnd] = \bot \ \wedge\ (\forall q :: q@\{37,38\} \ \Rightarrow\ p.y.rnd \neq q.nextrnd)$ (I5)

invariant $p@\{20..23\} \ \wedge\ (\exists q :: q@\{34\}) \ \Rightarrow$
$\quad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\quad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p) \ \mathbf{mod}\ N\big) - 2$ (I6)

invariant $p@\{20..23\} \ \wedge\ (\exists q :: q@\{35,36\}) \ \Rightarrow$
$\quad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\quad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 2$ (I7)

invariant $p@\{20..23\} \ \wedge\ \neg(\exists q :: q@\{34..36\}) \ \Rightarrow$
$\quad \mathit{Reset}[p.node].rnd = p.y.rnd \ \vee$
$\quad \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((\mathit{Check} - p - 1) \ \mathbf{mod}\ N\big) - 3$ (I8)

invariant $p@\{20..23\} \ \wedge\ q@\{37,38\} \ \Rightarrow\ p.y.rnd \neq q.nextrnd$ (I9)

invariant $p@\{23\} \ \wedge\ \mathit{Reset}[p.node].rnd = p.y.rnd \ \Rightarrow\ \mathit{Reset}[p.node].\mathit{free} = \mathit{false}$ (I10)

invariant $Y[i].\mathit{free} = \mathit{true} \ \Rightarrow\ \mathsf{Dist}[Y[i].rnd] = \bot \ \wedge\ (\forall p :: p@\{37,38\} \ \Rightarrow\ Y[i].rnd \neq p.nextrnd)$ (I11)

invariant $p@\{37\} \ \Rightarrow\ (\forall i :: \mathit{Reset}[i].rnd \neq p.nextrnd)$ (I12)

invariant $p@\{38,39\} \ \Rightarrow\ (\forall i :: \mathit{Reset}[i].rnd \neq p.y.rnd)$ (I13)

invariant $\mathsf{Dist}[\mathit{Reset}[i].rnd] = \bot \ \vee$
$\quad (\exists p :: p@\{36\} \ \wedge\ p.n = i \ \wedge\ \mathsf{Dist}[\mathit{Reset}[i].rnd] = 2N) \ \vee$
$\quad (\exists p :: p@\{37\} \ \wedge\ p.n = i \ \wedge\ \mathsf{Dist}[\mathit{Reset}[i].rnd] = 2N - 1)$ (I14)

invariant $i \neq h \Rightarrow Reset[i].rnd \neq Reset[h].rnd$ (I15)

invariant $p@\{20..23\} \wedge q@\{33\} \wedge (Check = p) \Rightarrow$
$Reset[p.node].rnd = p.y.rnd \vee q.usedrnd = p.y.rnd$ (I16)

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

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow$
$\neg[p@\{17..20\} \wedge (q@\{3..8, 17..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i))]$ (I17)

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow \neg(p@\{3..14, 21, 22, 24..41\} \wedge q@\{3..14, 21, 22, 24..41\})$ (I18)

invariant $A(p,i) \wedge q.n = i \Rightarrow \neg(p@\{17..19\} \wedge q@\{28..38\})$ (I19)

invariant $A(p,i) \wedge q.n = i \Rightarrow \neg(p@\{20, 21\} \wedge q@\{30..38\})$ (I20)

invariant $A(p,i) \wedge q.n = i \Rightarrow \neg(p@\{3..10, 22, 24\} \wedge q@\{31..38\})$ (I21)

invariant $A(p,i) \wedge (p@\{3..8, 20..22, 24\} \vee (p@\{9, 25..38\} \wedge p.n = i)) \wedge q@\{20..23\} \wedge p \neq q \Rightarrow$
$p.y.rnd \neq q.y.rnd$ (I22)

(IV) Miscellaneous invariants that are either trivial or follow almost directly from the mutual exclusion property (I60). In (I37) and (I38), $\|Free\|$ denotes the length of the *Free* queue.

invariant $p@\{29\} \Rightarrow Reset[p.n] = p.y$ (I23)

invariant $p@\{30..37\} \Rightarrow Reset[p.n] = (false, p.y.rnd)$ (I24)

invariant $p@\{27..38\} \Rightarrow Y[p.n] = (false, 0)$ (I25)

invariant $Y[i].free = true \Rightarrow Y[i] = Reset[i]$ (I26)

invariant $Reset[i].rnd \neq 0$ (I27)

invariant $p@\{17..23\} \Rightarrow p.y.free = true$ (I28)

invariant $p@\{17..23, 29..40\} \Rightarrow p.y.rnd \neq 0$ (I29)

invariant $p@\{19..23\} \Rightarrow Inuse[p] = p.y.rnd$ (I30)

invariant $p@\{2, 3, 15..24\} \Rightarrow (0 \leq p.level \leq L) \wedge (1 \leq p.node \leq T)$ (I31)

invariant $p@\{9, 25..40\} \Rightarrow$
$1 \leq p.n \leq T \wedge p.n = p.path[p.j].node \wedge p.dir = p.path[p.j].dir \wedge p.j = \mathsf{lev}(p.n)$ (I32)

invariant $p@\{9, 25..40\} \wedge p.dir = stop \Rightarrow p.n = p.node$ (I33)

invariant $p@\{9, 25..40\} \wedge p.n = p.node \Rightarrow p.dir = stop$ (I34)

invariant $p@\{2..41\} \Rightarrow (0 \leq p.level \leq L+1) \wedge (\mathsf{lev}(p.node) = p.level)$ (I35)

invariant $p@\{39, 40\} \Rightarrow p.n = p.node$ (I36)

invariant $\neg(\exists p :: p@\{36\}) \Rightarrow \|Free\| = 2N$ (I37)

invariant $(\exists p :: p@\{36\}) \Rightarrow \|Free\| = 2N+1$ (I38)

invariant $p@\{32..34\} \Rightarrow p.ptr = Check$ (I39)

invariant $p@\{37, 38\} \Rightarrow \mathsf{Dist}[p.nextrnd] = \perp$ (I40)

invariant $p@\{36\} \Rightarrow \mathsf{Dist}[p.y.rnd] = 2N$ (I41)

invariant $p@\{37\} \Rightarrow \mathsf{Dist}[p.y.rnd] = 2N-1$ (I42)

invariant $(1 \leq i \leq S) \wedge Round[i] = true \Rightarrow$
$(\exists p :: p.y.rnd = i \wedge (p@\{3..5, 8, 22..24\} \vee (p@\{9, 25..39\} \wedge p.n = p.node)) \wedge$
$(p@\{3..5, 8\} \Rightarrow (p.dir = stop \wedge 1 \leq p.node \leq T)))$ (I43)

invariant $p@\{4..14, 25..41\} \wedge (p.node \leq T) \Rightarrow p.path[p.level] = (p.node, stop)$ (I44)

invariant $p@\{2..41\} \wedge (2i \xrightarrow{*} p.node) \Rightarrow p.path[\mathsf{lev}(i)] = (i, left)$ (I45)

invariant $p@\{3\} \wedge (p.level > 0) \Rightarrow$
$p.path[0].node = 1 \wedge$
$(\forall l : 0 \leq l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node) \wedge$

$$p.path[p.level-1].node \xrightarrow{*} p.node \tag{I46}$$

invariant $p@\{26..38\} \Rightarrow (p.n = p.node) \vee (2 \cdot p.n \xrightarrow{*} p.node)$ (I47)

(V) Invariants that give the value of the auxiliary variable Loc.

invariant $p@\{0,1\} \Rightarrow \mathsf{Loc}[p] = 0 \wedge \mathsf{Loc}[p+N] = 0$ (I48)

invariant $p@\{2,15\} \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node$ (I49)

invariant $p@\{16\} \Rightarrow \big(X[p.node] = p \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node\big) \wedge$
$\big(X[p.node] \neq p \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node\big)$ (I50)

invariant $p@\{17..24\} \Rightarrow \big(A(p, p.node) \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node\big) \wedge$
$\big(\neg A(p, p.node) \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node\big)$ (I51)

invariant $p@\{3\} \Rightarrow \big(p.dir = stop \Rightarrow \mathsf{Loc}[p] = p.node \wedge \mathsf{Loc}[p+N] = p.node\big) \wedge$
$\big(p.dir = left \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node\big) \wedge$
$\big(p.dir = right \Rightarrow \mathsf{Loc}[p] = 2 \cdot p.node + 1 \wedge \mathsf{Loc}[p+N] = 2 \cdot p.node + 1\big)$ (I52)

invariant $p@\{4..14, 25..41\} \Rightarrow \mathsf{Loc}[p+N] = p.node$ (I53)

invariant $p@\{4..8\} \Rightarrow \mathsf{Loc}[p] = p.node$ (I54)

invariant $p@\{9, 25\} \Rightarrow p.n \xrightarrow{*} \mathsf{Loc}[p] \xrightarrow{*} p.node$ (I55)

invariant $p@\{26..40\} \Rightarrow \mathsf{Loc}[p] = p.n$ (I56)

(VI) Invariants that limit the maximum number of processes allowed within a given subtree of the renaming tree.

invariant $Y[i].free = false \Rightarrow \big(\exists p :: \big(p@\{3..8, 18..24\} \vee (p@\{9, 25..38\} \wedge p.n = i)\big) \wedge$
$(p.node = i) \wedge (p@\{3\} \Rightarrow p.dir \neq right)\big) \vee$
$\big(\exists p :: p@\{2..9, 15..40\} \wedge (2i \xrightarrow{*} p.node) \wedge$
$(p@\{9, 25..40\} \Rightarrow 2i \xrightarrow{*} p.n)\big) \vee$
$\big(\exists p :: p@\{9, 25..38\} \wedge p.n = i \wedge (p@\{9, 25\} \Rightarrow 2i \xrightarrow{*} \mathsf{Loc}[p])\big)$ (I57)

invariant $p@\{2, 3, 15..24\} \wedge (i \xrightarrow{*} \mathsf{Loc}[p]) \Rightarrow \mathsf{PC}[p, i] \geq C(i)$ (I58)

invariant $p@\{2, 3, 15..24\} \wedge (i \xrightarrow{*} \mathsf{Loc}[p]) \wedge (2 \leq i \leq T) \Rightarrow \mathsf{PC}[p, i] < \mathsf{PC}[p, \lfloor i/2 \rfloor]$ (I59)

(VII) Invariants that prove mutual exclusion and contention sensitivity.

invariant **(Mutual exclusion)** $\big|\{p :: p@\{8..10, 13, 25..40\}\}\big| \leq 1$ (I60)

invariant **(Contention sensitivity)** $p@\{4..8\} \Rightarrow \mathsf{lev}(p.node) < \mathsf{PC}[p, 1]$ (I61)

We now prove that each of (I1)–(I61) 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. The following lemma is used in several of the proofs.

Lemma A1: If $1 \leq i \leq T$ holds and t and u are consecutive states such that $A(p, i)$ is false at t but true at u, and if all the invariants stated above hold at t, then the following are true.

- u is reached from t via the execution of statement $16.p$.
- $p@\{17\}$ is established at u.
- $p.node = i \wedge Y[i].free = true$ holds at both t and u.

Proof: The only statements that could potentially establish $A(p,i)$ are the following.

- 1.p and 3.p, which could establish $p.node = i$.
- 7.p, 16.p, and 23.p, which establish $p@\{3..5, 8..14, 17..22, 24..41\}$.
- 15.p and 27.p, which could establish $X[i] = p$.
- 19.p, which falsifies $p@\{17..19\}$.
- 16.p, 28.p, 29.q, and 37.q, where q is any arbitrary process, which could establish $Reset[i] = p.y$.
- 19.p, 20.p, and 22.p, which could falsify $p@\{17..22\}$.
- 24.p, which could establish $p@\{3\} \land p.dir = stop$.

We now show that none of these statements other than 16.p can establish $A(p,i)$, and 16.p can do so only if the conditions specified in the lemma are met.

Statements 1.p and 15.p establish $p@\{2, 16\}$. Therefore, they cannot establish $A(p,i)$.

Statement 3.p establishes $p@\{2, 4, 6\}$. If it establishes $p@\{2, 6\}$, then $A(p,i)$ is false at u. If it establishes $p@\{4\}$, then $p.dir = stop$ holds at t. In this case, 3.p can establish $A(p,i)$ only if $p.node = i$ also holds at t. But this implies that $A(p,i)$ holds at t, which is a contradiction.

Statement 7.p can establish $A(p,i)$ by establishing $p@\{8\}$ only if $p.node = i$ holds at t. However, note that statement 7.p can be executed only if $p.level > L$ holds. By (I35), this implies $p.node > T$, a contradiction.

Statement 16.p can establish $A(p,i)$ by establishing $p@\{17\}$ only if $p.node = i \land Y[i].free = true$ holds at t. But then this condition also holds at u.

Statement 19.p establishes either $p@\{3\}$ or $p@\{20\}$. If it establishes $p@\{3\}$, then it also establishes $p.dir = left$, in which case $A(p,i)$ is false at u. If 19.p establishes $p@\{20\}$, then $A(p,i)$ holds at u only if $p.node = i \land X[i] = p \land Reset[i] = p.y$ holds at t. But this contradicts our assumption that $A(p,i)$ is false at t.

Statements 20.p and 23.p can establish $A(p,i)$ only if they establish $p@\{3\}$. In this case, they also establish $p.dir = left$, which implies that $A(p,i)$ is false.

Statement 22.p establishes either $p@\{23\}$ or $p@\{24\}$. If it establishes $p@\{23\}$, then $A(p,i)$ is false at u. On the other hand, if it establishes $p@\{24\}$, then $A(p,i)$ holds at u only if $p.node = i \land Reset[i] = p.y$ holds at t. But this contradicts our assumption that $A(p,i)$ is false at t.

Statements 24.p, 27.p, 28.p, 29.p, and 37.p can establish $A(p,i)$ only if $p.node = i$ holds at both t and u. But then $A(p,i)$ holds at t, a contradiction.

Statement 29.q, where $q \neq p$, can establish $A(p,i)$ only by establishing $Reset[i] = p.y$ when $p@\{17..22\} \land q.n = i$ holds. In this case, 29.q establishes $Reset[i].free = false$. By (I28), if $p@\{17..22\}$ holds at t, then $p.y.free = true$ holds at t, and hence also at u. Therefore, 29.q cannot establish $Reset[i] = p.y$.

Statement 37.q can establish $A(p,i)$ only by establishing $Reset[i] = p.y$ when the expression $p@\{17..22\} \land p.node = i \land q@\{37\} \land q.n = i$ holds at t. We consider the two cases $p@\{17..19\}$ and $p@\{20..22\}$ separately.

Suppose that $p@\{17..19\}$ holds at t. In this case, 37.q can establish $A(p,i)$ only if $X[i] = p$ holds at t. By (I5), this implies that $Reset[i] = p.y$ holds as well. Thus, we have $p@\{17..19\} \land p.node = i \land X[i] = p \land Reset[i] = p.y$ at state t, which implies that $A(p,i)$ is true at t, a contradiction.

Finally, suppose that $p@\{20..22\}$ holds at t. In this case, by (I9), $p.y.rnd \neq q.nextrnd$ also holds. Therefore, $Reset[i].rnd \neq p.y.rnd$ holds after the execution of 37.q, which implies that $A(p,i)$ is false. □

Proof of Mutual Exclusion

We begin by proving those invariants needed to establish the mutual exclusion property (I60).

invariant $A(p,i) \wedge \big(p@\{3..8, 18..22, 24\} \vee (p@\{9, 25..38\} \wedge p.n = i)\big) \Rightarrow Y[i] = (\mathit{false}, 0)$ (I1)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is $16.p$. However, if statement $16.p$ establishes $A(p,i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I1).

The only statements that can establish $p@\{3..8, 18..22, 24\}$ are $16.p$, $17.p$, and $23.p$. Statement $17.p$ establishes the consequent. Statements $16.p$ and $23.p$ can establish $p@\{18..22, 3..8\}$ only by establishing $p@\{3\}$, in which case they also establish $p.dir \neq stop$. Thus, if these statements establish $p@\{18..22, 3..8\}$, then they also falsify $A(p,i)$.

The only statement that can establish $p@\{9, 25..38\} \wedge p.n = i$ while $A(p,i)$ holds is $8.p$. (Note that $A(p,i)$ implies $p.node = i$ by definition. Thus, if statement $40.p$ establishes $p@\{9\}$, then $p.n \neq i$ holds after its execution.) In this case, the antecedent holds before the execution of $8.p$. Thus, although statement $8.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statement $38.q$, where q is any arbitrary process. If $q = p$, then statement $38.q$ establishes $(p@\{9\} \wedge p.n \neq i) \vee p@\{10, 13, 39\}$, which implies that the antecedent is false. Suppose that $q \neq p$. Statement $38.q$ can falsify the consequent only if executed when $q.n = i$ holds. However, by (I19), (I20), and (I21), the antecedent and $q@\{38\} \wedge q.n = i$ cannot hold simultaneously. □

invariant $A(p,i) \wedge \big(p@\{3..8, 22, 24\} \vee (p@\{9, 25..38\} \wedge p.n = i)\big) \Rightarrow Round[p.y.rnd] = \mathit{true}$ (I2)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is $16.p$. However, if statement $16.p$ establishes $A(p,i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I2).

The condition $p@\{3..8, 22, 24\}$ may be established only by statements $16.p$, $19.p$, $20.p$, $21.p$, and $23.p$. Statement $21.p$ establishes the consequent. If statements $16.p$, $19.p$, $20.p$, and $23.p$ establish $p@\{3..8, 22, 24\}$, then they also establish $p@\{3\} \wedge p.dir \neq stop$, which implies that $A(p,i)$ is false after the execution of each of these statements.

The only statement that can establish $p@\{9, 25..38\} \wedge p.n = i$ while $A(p,i)$ holds is $8.p$. (Note that $A(p,i)$ implies $p.node = i$ by definition.) In this case, the antecedent holds before the execution of $8.p$. Thus, although statement $8.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which may change the value of $p.y.rnd$) and $23.q$ and $39.q$ (which assign the value *false* to an element of the *Round* array), where q is any arbitrary process. Statement $16.p$ establishes $p@\{17\} \vee (p@\{3\} \wedge p.dir = right)$, which implies that the antecedent is false. If both the antecedent and $p@\{28\}$ hold, then by (I3), $Reset[i].rnd = p.y.rnd$ holds. It follows that statement $28.p$ cannot change the value of $p.y.rnd$ when the antecedent is true, and hence cannot falsify (I2).

If $q = p$, then statement $23.q$ establishes $p@\{3\} \wedge p.dir = left$ and statement $39.q$ establishes $p@\{40\}$, both of which imply that the antecedent is false.

Suppose that $q \neq p$. In this case, statements $23.q$ and $39.q$ may falsify the consequent only if $p.y.rnd = q.y.rnd$ holds. By (I22), $p.y.rnd = q.y.rnd \wedge q@\{23\}$ implies that the antecedent of (I2) is false. Thus, statement $23.q$ cannot falsify (I2).

By (I60), if the antecedent and $q@\{39\}$ hold, then we have $A(p,i) \wedge p@\{3..7, 22, 24\}$. By the definition of $A(p,i)$, $A(p,i) \wedge p@\{22\}$ implies that $Reset[i].rnd = p.y.rnd$ holds. By (I3), $A(p,i) \wedge p@\{3..7, 24\}$ also implies that $Reset[i].rnd = p.y.rnd$ holds. By (I13), $Reset[i].rnd = p.y.rnd \wedge q@\{39\}$ implies that $p.y.rnd \neq q.y.rnd$. Thus, statement $39.q$ cannot falsify (I2). □

invariant $A(p,i) \wedge \big(p@\{3..8, 24\} \vee (p@\{9, 25..37\} \wedge p.n = i)\big) \Rightarrow Reset[i].rnd = p.y.rnd$ (I3)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is $16.p$. However, if statement $16.p$ establishes $A(p,i)$, then it also establishes $p@\{17\}$. Hence, it cannot falsify (I3).

The condition $p@\{3..8, 24\}$ may be established only by $16.p$, $19.p$, $20.p$, $22.p$, and $23.p$. If statements $16.p$, $19.p$, $20.p$, and $23.p$ establish $p@\{3..8, 24\}$, then they also establish $p@\{3\} \wedge p.dir \neq stop$, which implies that

$A(p, i)$ is false after the execution of each of these statements. Statement 22.p can establish the antecedent only if $Reset[i] = p.y$ holds. Hence, it preserves (I3).

The only statement that can establish $p@\{9, 25..37\} \wedge p.n = i$ while $A(p, i)$ holds is 8.p. (Note that $A(p, i)$ implies $p.node = i$ by definition.) In this case, the antecedent holds before the execution of 8.p. Thus, although statement 8.p may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements 16.p and 28.p (which update $p.y.rnd$) and 29.q and 37.q (which update $Reset[i].rnd$), where q is any arbitrary process. The antecedent is false after the execution of 16.p (as explained above) and 37.p (which establishes $p@\{38\}$). If statement 28.p or 29.p is executed while the antecedent holds, then the consequent is preserved.

This leaves only statements 29.q and 37.q, where $q \neq p$. By (I23), 29.q cannot change $Reset[i].rnd$, and hence, cannot falsify (I3). Statement 37.q may falsify the consequent only if $q.n = i$ holds. By (I21), $q.n = i \wedge q@\{37\}$ implies that the antecedent of (I3) is false. Thus, statement 37.q cannot falsify (I3). □

invariant $\big(\exists p :: A(p, i) \wedge p@\{3..14, 25..41\}\big) = \big(Acquired[i] = true\big)$ (I4)

Proof: By the definition of $A(p, i)$, the left-hand side of (I4) is equivalent to $\big(\exists p :: p.node = i \wedge p@\{3..5, 8..14, 25..41\} \wedge (p@\{3\} \Rightarrow p.dir = stop)\big)$. Thus, the left-hand side may be established or falsified only by statements 1.p (which may update $p.node$), 3.p (which may update $p.node$ and also falsify $p@\{3\}$), 16.p, 19.p, 20.p, 23.p, and 24.p (which may establish $p@\{3\}$ and also update $p.dir$), and 7.p, 14.p, and 41.p (which may establish or falsify $p@\{3..5, 8..14, 25..41\}$). 24.$p$ and 41.p are also the only statements that may establish or falsify the right-hand side of (I4) (p can be any process here).

The left-hand side of (I4) is false before and after the execution of each of 1.p, 16.p, 19.p, 20.p, and 23.p. (Note that, if one of 16.p, 19.p, 20.p, or 23.p establishes $p@\{3\}$, then it also establishes $p.dir \neq stop$.) If statement 3.p is executed when $p.dir = stop$ holds, then it establishes $p@\{4\}$, and does not update $p.node$. Thus, the left-hand side is true before and after its execution. On the other hand, if statement 3.p is executed when $p.dir \neq stop$ holds, then it establishes $p@\{2, 6\}$, and hence the left-hand side is false before and after its execution. It follows that statement 3.p cannot establish or falsify the left-hand side.

Statement 24.p establishes the left-hand side if and only if it also establishes the right-hand side. Statement 41.p falsifies the right-hand side if and only if executed when $p.node = i$ holds, in which case $A(p, i)$ holds by definition. By (I18), this implies that the left-hand side of (I4) is falsified.

Statements 7.p and 14.p may be executed only when $p.level > L$. By (I35), $p.level > L$ implies that $p.node > T$. Because $i \leq T$ (by assumption), this implies that $A(p, i)$ is false both before and after any of these statements is executed. Thus, these statements can neither establish nor falsify the left-hand side of (I4). □

invariant $p@\{17..19\} \wedge X[p.node] = p \Rightarrow$
$Reset[p.node] = p.y \wedge \mathsf{Dist}[p.y.rnd] = \bot \wedge (\forall q :: q@\{37, 38\} \Rightarrow p.y.rnd \neq q.nextrnd)$ (I5)

Proof: The antecedent may be established only by statements 16.p (which establishes $p@\{17..19\}$), 1.p and 3.p (which update $p.node$), and 15.p and 27.p (which may establish $X[p.node] = p$). However, statements 1.p, 3.p, 15.p, and 27.p establish $p@\{2, 4, 6, 16, 28\}$ and hence cannot establish the antecedent. Also, by (I26) and (I11), if 16.p establishes the antecedent, then it also establishes the consequent.

The consequent may be falsified only by statements 16.p and 28.p (which may change the value of $p.y$), 1.p and 3.p (which may update $p.node$), 36.r (which may establish $r@\{37, 38\} \wedge p.y.rnd = r.nextrnd$), 29.$r$ and 37.r (which may update $Reset[p.node]$), and 35.r (which may falsify $\mathsf{Dist}[p.y.rnd] = \bot$), where r is any arbitrary process. However, 16.p preserves (I5) as shown above. Furthermore, the antecedent is false after the execution of 1.p, 3.p, and 28.p and also after the execution of each of 29.r, 35.r, 36.r, and 37.r if $r = p$.

Consider statements 29.r, 35.r, and 37.r, where $r \neq p$. If the antecedent and consequent of (I5) both hold, then by (I31), $p.node \leq T$ holds, and hence $A(p, p.node)$ holds. Statements 29.r and 37.r may falsify the consequent only if $r.n = p.node$ holds. Similarly, statement 35.r may falsify the consequent only if

$r.y.rnd = p.y.rnd$. If $r@\{35\} \wedge r.y.rnd = p.y.rnd$ and the consequent both hold, then by (I24), we have $Reset[r.n].rnd = Reset[p.node].rnd$. By (I15), this implies that $r.n = p.node$. Therefore, each of these statements may falsify the consequent only if $r.n = p.node$ holds. However, by (I19), $r.n = p.node \wedge r@\{29, 35, 37\} \wedge A(p, p.node)$ implies that $p@\{17..19\}$ is false. Thus, these statements cannot falsify (I5).

Finally, statement $36.r$ may establish $p.y.rnd = r.nextrnd$ only if $p.y.rnd$ is at the head of *Free* queue, *i.e.*, $\mathsf{Dist}[p.y.rnd] = 0$. But this implies that $\mathsf{Dist}[p.y.rnd] = \perp$ is false. Because (I5) is assumed to hold prior to the execution of $36.r$, this implies that the antecedent of (I5) is false before $36.r$ is executed. Thus, the antecedent is also false after the execution of $36.r$. □

invariant $p@\{20..23\} \wedge (\exists q :: q@\{34\}) \Rightarrow$
$$Reset[p.node].rnd = p.y.rnd \ \vee \ \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p) \ \mathbf{mod} \ N\big) - 2 \qquad \text{(I6)}$$

Proof: The antecedent may be established only by statements $19.p$ and $33.q$, where q is any process. Statement $19.p$ may establish the antecedent only if executed when $X[p.node] = p$ holds, in which case $Reset[p.node].rnd = p.y.rnd$ holds, by (I5).

Statement $33.q$ may establish the consequent only if executed when $p@\{20..23\}$ holds. By (I60), $q@\{33\} \wedge p@\{20..23\}$ implies that the antecedent of (I8) holds. By the consequent of (I8),

$$\big(Reset[p.node].rnd = p.y.rnd\big) \ \vee \ \big(\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \ \mathbf{mod} \ N\big) - 3\big) \qquad (1)$$

holds as well. Now, consider the following three cases.

- $Reset[p.node].rnd = p.y.rnd$ holds before $33.q$ is executed. In this case, $Reset[p.node].rnd = p.y.rnd$ holds after $33.q$, so (I6) is preserved.
- $Check = p \wedge Reset[p.node].rnd \neq p.y.rnd$ holds before $33.q$ is executed. In this case, by (I29), $p.y.rnd \neq 0$ holds, and by (I16), $q.usedrnd = p.y.rnd$ holds. Therefore, procedure *MoveToTail* is called. By (I60) and (I37), $q@\{33\}$ implies that $\|Free\| = 2N$. Thus, statement $33.q$ establishes $\mathsf{Dist}[p.y.rnd] = 2N - 1$, which implies the consequent.
- $Check \neq p \wedge Reset[p.node].rnd \neq p.y.rnd$ holds before $33.q$ is executed. By (1), this implies $\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \ \mathbf{mod} \ N\big) - 3$. Note that the value of $\mathsf{Dist}[p.y.rnd]$ can decrease by at most one by a call to *MoveToTail*. Note also that if $Check \neq p$, then $(Check - p) \ \mathbf{mod} \ N = \big((Check - p - 1) \ \mathbf{mod} \ N\big) + 1$. Therefore, after the execution of $33.q$,

$$\begin{aligned} \mathsf{Dist}[p.y.rnd] &> \big[2N - 2 \cdot \big((Check - p - 1) \ \mathbf{mod} \ N\big) - 3\big] - 1 \\ &= 2N - 2 \cdot \big(((Check - p) \ \mathbf{mod} \ N) - 1\big) - 4 \\ &= 2N - 2 \cdot \big((Check - p) \ \mathbf{mod} \ N\big) - 2. \end{aligned}$$

The consequent may be falsified only by statements $1.p$ and $3.p$ (which may update $p.node$), $16.p$ and $28.p$ (which may change the value of $p.y$), $29.r$ and $37.r$ (which may update $Reset[p.node].rnd$), $33.r$, $35.r$, and $36.r$ (which may update $\mathsf{Dist}[p.y.rnd]$), and $34.r$ (which may change the value of *Check*), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements $1.p$, $3.p$, $16.p$, and $28.p$. By (I60), statements $29.r$, $33.r$, $35.r$, $36.r$, and $37.r$ cannot be executed while the antecedent holds. Finally, by (I60), statement $34.r$ falsifies the antecedent. □

invariant $p@\{20..23\} \wedge (\exists q :: q@\{35, 36\}) \Rightarrow$
$$Reset[p.node].rnd = p.y.rnd \ \vee \ \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \ \mathbf{mod} \ N\big) - 2 \qquad \text{(I7)}$$

Proof: The antecedent may be established only by statements $19.p$ and $34.q$, where q is any process. Statement $19.p$ may establish the antecedent only if executed when $X[p.node] = p$ holds, in which case

$Reset[p.node].rnd = p.y.rnd$ holds, by (I5). Statement $34.q$ can establish the antecedent only if $q@\{34\} \wedge p@\{20..23\}$ holds. This implies that the consequent of (I6) holds before statement $34.q$ is executed. By (I39), statement $34.q$ increments the value of *Check* by 1 modulo-N. Thus, the consequent of (I7) is established.

The consequent may be falsified only by statements $1.p$ and $3.p$ (which may update $p.node$), $16.p$ and $28.p$ (which may change the value of $p.y$), $29.r$ and $37.r$ (which may update $Reset[p.node].rnd$), $33.r$, $35.r$, and $36.r$ (which may update $\mathsf{Dist}[p.y.rnd]$), and $34.r$ (which may change the value of *Check*), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements $1.p$, $3.p$, $16.p$, and $28.p$. By (I60), statements $29.r$, $33.r$, $34.r$, and $37.r$ cannot be executed while the antecedent holds. Statements $36.r$ falsifies the antecedent by (I60).

Statement $35.r$ may update $\mathsf{Dist}[p.y.rnd]$ only if executed when $\mathsf{Dist}[p.y.rnd] = \bot$, in which case it establishes $\mathsf{Dist}[p.y.rnd] = 2N$, by (I60) and (I37). This implies that the consequent holds. □

invariant $p@\{20..23\} \wedge \neg(\exists q :: q@\{34..36\}) \Rightarrow$
$$Reset[p.node].rnd = p.y.rnd \vee \mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \textbf{ mod } N\big) - 3 \qquad \text{(I8)}$$

Proof: The antecedent may be established only by statements $19.p$ and $36.q$, where q is any process. Statement $19.p$ may establish the antecedent only if executed when $X[p.node] = p$ holds, in which case $Reset[p.node].rnd = p.y.rnd$ holds, by (I5).

Statement $36.q$ may establish the antecedent only if executed when $q@\{36\} \wedge p@\{20..23\}$ holds. If $Reset[p.node].rnd = p.y.rnd$ holds before the execution of $36.q$, then it holds afterward as well, and thus (I8) is not falsified. So, assume that $q@\{36\} \wedge p@\{20..23\} \wedge Reset[p.node].rnd \neq p.y.rnd$ holds before the execution of $36.q$. In this case, by (I7), $\mathsf{Dist}[p.y.rnd] > 0$ holds. Therefore, the function *Dequeue* decrements $\mathsf{Dist}[p.y.rnd]$ by 1. Moreover, by (I7), $\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \textbf{ mod } N\big) - 2$ holds before $36.q$ is executed, which implies that $\mathsf{Dist}[p.y.rnd] > 2N - 2 \cdot \big((Check - p - 1) \textbf{ mod } N\big) - 3$ holds afterward.

The consequent may be falsified only by statements $1.p$ and $3.p$ (which may update $p.node$), $16.p$ and $28.p$ (which may change the value of $p.y$), $29.r$ and $37.r$ (which may update $Reset[p.node].rnd$), $33.r$, $35.r$, and $36.r$ (which may update $\mathsf{Dist}[p.y.rnd]$), and $34.r$ (which may change the value of *Check*), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements $1.p$, $3.p$, $16.p$, and $28.p$. Statement $33.r$ establishes $r@\{34\}$, and hence falsifies the antecedent. By (I23), statement $29.r$ does not change the value of $Reset[p.node].rnd$. Statements $34.r$, $35.r$, and $36.r$ cannot be executed while the antecedent holds.

Finally, statement $37.r$ may falsify the consequent only if executed when $r@\{37\} \wedge Reset[p.node].rnd = p.y.rnd \wedge p.node = r.n$ holds. By (I24), this implies that $Reset[r.n].rnd = r.y.rnd$ holds. Also, by (I42), $\mathsf{Dist}[r.y.rnd] = 2N - 1$ holds. Combining these assertions, we have $p.y.rnd = r.y.rnd$, and hence $\mathsf{Dist}[p.y.rnd] = 2N - 1$. This implies that the consequent of (I8) holds after $37.r$ is executed. □

invariant $p@\{20..23\} \wedge q@\{37, 38\} \Rightarrow p.y.rnd \neq q.nextrnd$ (I9)

Proof: The antecedent may be established only by statements $19.p$ and $36.q$. Statement $19.p$ may establish the antecedent only if executed when $p@\{19\} \wedge X[p.node] = p \wedge q@\{37, 38\}$ holds. By (I5), this implies that $p.y.rnd \neq q.nextrnd$ holds. Thus, the consequent of (I9) is true after the execution of $19.p$.

Statement $36.q$ may establish the antecedent of (I9) only if executed when the antecedent of (I7) holds. By (I60), the third disjunct of (I14) does not hold before the execution of $36.q$. Thus, if $Reset[p.node].rnd = p.y.rnd$ holds before $36.q$ is executed, then $\mathsf{Dist}[p.y.rnd] = \bot \vee \mathsf{Dist}[p.y.rnd] = 2N$ holds. On the other hand, if $Reset[p.node].rnd \neq p.y.rnd$ holds before $36.q$ is executed, then by (I7), $\mathsf{Dist}[p.y.rnd] > 0$ holds. In either case, *Dequeue* must return a value different from $p.y.rnd$. Hence, the consequent of (I9) is established.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which may change the value of $p.y$) and $36.q$ (which may update $q.nextrnd$). However, $p@\{20..23\}$ is false after the execution of statements $16.p$ and $28.p$, and statement $36.q$ preserves (I9) as shown above. □

invariant $p@\{23\} \wedge \mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd} \Rightarrow \mathit{Reset}[p.\mathit{node}].\mathit{free} = \mathit{false}$ (I10)

Proof: (I10) may be falsified only by statements $22.p$ (which may establish $p@\{23\}$), $16.p$ and $28.p$ (which may change the value of $p.y.\mathit{rnd}$), $1.p$ and $3.p$ (which may change the value of $p.\mathit{node}$), and $29.q$ and $37.q$ (which may update $\mathit{Reset}[p.\mathit{node}]$), where q is any arbitrary process. However, $p@\{23\}$ is false after the execution of statements $1.p$, $3.p$, $16.p$, and $28.p$. Statement $22.p$ establishes the antecedent only if executed when $\mathit{Reset}[p.\mathit{node}] \neq p.y \wedge \mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds, which implies that $\mathit{Reset}[p.\mathit{node}].\mathit{free} \neq p.y.\mathit{free}$. By (I28), $p.y.\mathit{free} = \mathit{true}$. Thus, $\mathit{Reset}[p.\mathit{node}].\mathit{free} = \mathit{false}$ holds.

If $q = p$, then each of $29.q$ and $37.q$ establishes $p@\{30, 38\}$, which implies that the antecedent is false.

Consider statements $29.q$ and $37.q$, where $q \neq p$. Statement $29.q$ trivially establishes or preserves the consequent. Statement $37.q$ could potentially falsify (I10) only if executed when $p@\{23\} \wedge q@\{37\} \wedge q.n = p.\mathit{node}$ holds. In this case, by (I9), $p.y.\mathit{rnd} \neq \mathit{Reset}[p.\mathit{node}].\mathit{rnd}$ holds after the execution of $37.q$. Thus, statement $37.q$ cannot falsify (I10). □

invariant $Y[i].\mathit{free} = \mathit{true} \Rightarrow \mathsf{Dist}[Y[i].\mathit{rnd}] = \perp \wedge (\forall p :: p@\{37, 38\} \Rightarrow Y[i].\mathit{rnd} \neq p.\mathit{nextrnd})$ (I11)

Proof: The antecedent may be established only by statement $38.q$, where q is any arbitrary process. However, by (I60) and (I40), if $38.q$ is executed when $q.n = i$ holds, then it establishes $\mathsf{Dist}[Y[i].\mathit{rnd}] = \perp \wedge \neg(\exists p :: p@\{37, 38\})$.

The consequent may be falsified only by statements $17.q$, $26.q$, and $38.q$ (which may update $Y[i].\mathit{rnd}$), $35.q$ (which may falsify $\mathsf{Dist}[Y[i].\mathit{rnd}] = \perp$), and $36.q$ (which may update $q.\mathit{nextrnd}$, and may also establish $q@\{37, 38\}$), where q is any arbitrary process. Statements $17.q$ and $26.q$ falsify the antecedent. Statement $38.q$ preserves (I11) as shown above.

Statement $35.q$ may falsify $\mathsf{Dist}[Y[i].\mathit{rnd}] = \perp$ only if executed when $q@\{35\} \wedge Y[i].\mathit{free} = \mathit{true} \wedge q.y.\mathit{rnd} = Y[i].\mathit{rnd}$ holds. In this case, by (I26) and (I24), $Y[i].\mathit{rnd} = \mathit{Reset}[i].\mathit{rnd}$ and $\mathit{Reset}[q.n].\mathit{rnd} = q.y.\mathit{rnd}$ are both true as well. It follows that $\mathit{Reset}[q.n].\mathit{rnd} = \mathit{Reset}[i].\mathit{rnd}$ is also true, and hence $q.n = i$ holds, by (I15). By (I25), this in turn implies that $Y[i].\mathit{free} = \mathit{false}$ holds, a contradiction. It follows that statement $35.q$ cannot falsify the consequent while the antecedent holds.

If $Y[i].\mathit{free} = \mathit{false}$ holds before statement $36.q$ is executed, then it holds afterward, and hence (I11) is not falsified. If $Y[i] = \mathit{true}$ holds before $36.q$ is executed, then $\mathsf{Dist}[Y[i].\mathit{rnd}] = \perp$ holds as well, since (I11) is presumed to hold before the execution of $36.q$. Thus, $Y[i].\mathit{rnd}$ is not in the *Free* queue. This implies that a value different from $Y[i].\mathit{rnd}$ is dequeued, *i.e.*, $Y[i].\mathit{rnd} \neq q.\mathit{nextrnd}$ holds after the execution of $36.q$. □

invariant $p@\{37\} \Rightarrow (\forall i :: \mathit{Reset}[i].\mathit{rnd} \neq p.\mathit{nextrnd})$ (I12)

Proof: The antecedent may be established only by statement $36.p$, which may establish $\mathit{Reset}[i].\mathit{rnd} = p.\mathit{nextrnd}$ only if executed when $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 0$, *i.e.*, when $\mathit{Reset}[i].\mathit{rnd}$ is at the head of the *Free* queue. However, this is precluded by (I14).

The consequent may be falsified only by statement $36.p$ (which may update $p.\mathit{nextrnd}$), and statements $29.q$ and $37.q$ (which may update $\mathit{Reset}[i].\mathit{rnd}$), where q is any arbitrary process. However, statement $36.p$ preserves (I12) as shown above. By (I23), statement $29.q$ does not change the value of $\mathit{Reset}[i].\mathit{rnd}$. By (I60), the antecedent is false after the execution of statement $37.q$. □

invariant $p@\{38, 39\} \Rightarrow (\forall i :: \mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd})$ (I13)

Proof: The antecedent may be established only by statement $37.p$. Before its execution, $\mathit{Reset}[i].\mathit{rnd} \neq p.\mathit{nextrnd}$ holds, by (I12), and $\mathit{Reset}[p.n].\mathit{rnd} = p.y.\mathit{rnd}$ holds, by (I24). We consider two cases. First, suppose that $i = p.n$ holds before $37.p$ is executed. In this case, $p.y.\mathit{rnd} \neq p.\mathit{nextrnd}$ is true before the execution of $37.p$, and hence $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ is true after.

Second, suppose that $i \neq p.n$ holds before the execution of $37.p$. In this case, by (I15), $\mathit{Reset}[i].\mathit{rnd} \neq \mathit{Reset}[p.n].\mathit{rnd}$ holds as well. Because $\mathit{Reset}[p.n].\mathit{rnd} = p.y.\mathit{rnd}$ is true before $37.p$ is executed, we have $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ as well. Thus, $\mathit{Reset}[i].\mathit{rnd} \neq p.y.\mathit{rnd}$ holds after the execution of $37.p$.

The consequent may be falsified only by statements $16.p$ and $28.p$ (which may update $p.y.\mathit{rnd}$), and $29.q$ and $37.q$ (which may update $\mathit{Reset}[i].\mathit{rnd}$), where q is any arbitrary process. However, the antecedent is false after the execution of $16.p$ and $28.p$, and by (I60), $29.q$ and $37.q$ are not enabled while the antecedent holds. □

invariant $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot \ \vee$
$(\exists p :: p@\{36\} \ \wedge \ p.n = i \ \wedge \ \mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 2N) \ \vee$
$(\exists p :: p@\{37\} \ \wedge \ p.n = i \ \wedge \ \mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 2N - 1)$ (I14)

Proof: The only statements that may falsify (I14) are $29.q$ (which may update $\mathit{Reset}[i].\mathit{rnd}$), $37.q$ (which may falsify $(\exists p :: p@\{36, 37\} \ \wedge \ p.n = i)$ and may update $\mathit{Reset}[i].\mathit{rnd}$), and $33.q$, $35.q$, and $36.q$ (which may update $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}]$), where q is any arbitrary process. By (I23), statement $29.q$ does not change the value of $\mathit{Reset}[i].\mathit{rnd}$.

Statement $37.q$ may falsify (I14) only if executed when $q.n = i$ holds, in which case it establishes $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot$, by (I40).

If $33.q$ is enabled, then by (I60), $\neg(\exists p :: p@\{36, 37\})$ holds. Since (I14) is presumed to hold before the execution of $33.q$, this implies that $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot$ holds both before and after $33.q$ is executed.

Statement $35.q$ may falsify $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot$ only if executed when $q@\{35\} \ \wedge \ q.y.\mathit{rnd} = \mathit{Reset}[i].\mathit{rnd}$ holds. In this case, by (I24), $\mathit{Reset}[q.n].\mathit{rnd} = q.y.\mathit{rnd}$ holds as well. This implies that $\mathit{Reset}[q.n].\mathit{rnd} = \mathit{Reset}[i].\mathit{rnd}$ is true, and hence $q.n = i$ holds, by (I15). Because the *Enqueue* procedure puts $q.y.\mathit{rnd}$ at the tail of the *Free* queue, by (I37) and (I60), $35.q$ establishes $\mathsf{Dist}[q.y.\mathit{rnd}] = 2N$. Therefore, if statement $35.q$ falsifies $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = \bot$, then it establishes the second disjunct of (I14).

Statement $36.q$ may falsify (I14) only if executed when $q@\{36\} \ \wedge \ q.n = i \ \wedge \ \mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}] = 2N$ holds. In this case, the *Dequeue* decrements $\mathsf{Dist}[\mathit{Reset}[i].\mathit{rnd}]$ by one, establishing the third disjunct of (I14). □

invariant $i \neq h \ \Rightarrow \ \mathit{Reset}[i].\mathit{rnd} \neq \mathit{Reset}[h].\mathit{rnd}$ (I15)

Proof: The only statements that may falsify (I15) are $29.p$ and $37.p$, where p is any arbitrary process. By (I23), statement $29.p$ does not change the value of $\mathit{Reset}[i].\mathit{rnd}$, and hence cannot falsify (I15). Statement $37.p$ may falsify (I15) only if $p.n = h \ \wedge \ \mathit{Reset}[i].\mathit{rnd} = p.\mathit{nextrnd}$ holds prior to its execution. However, this is precluded by (I12). □

invariant $p@\{20..23\} \ \wedge \ q@\{33\} \ \wedge \ (\mathit{Check} = p) \ \Rightarrow$
$\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd} \ \vee \ q.\mathit{usedrnd} = p.y.\mathit{rnd}$ (I16)

Proof: The antecedent may be established only by statements $19.p$, $32.q$, and $34.r$, where r is any arbitrary process. However, by (I60), statement $34.r$ cannot be executed while $q@\{33\}$ holds.

Statement $19.p$ may establish $p@\{20\}$ only if executed when $X[p.\mathit{node}] = p$ holds. By (I5), this implies that $\mathit{Reset}[p.\mathit{node}].\mathit{rnd} = p.y.\mathit{rnd}$ holds both before and after $19.p$ is executed.

Statement $32.q$ may establish the antecedent only if executed when $p@\{20..23\} \ \wedge \ q@\{32\} \ \wedge \ \mathit{Check} = p$ holds. By (I30) and (I39), this implies that both $\mathit{Inuse}[p] = p.y.\mathit{rnd}$ and $q.\mathit{ptr} = p$ are also true. Thus, statement $32.q$ establishes $q.\mathit{usedrnd} = p.y.\mathit{rnd}$, and hence does not falsify (I16).

The consequent may be falsified only by statements $1.p$, $3.p$, $16.p$, and $28.p$ (which may update either $p.\mathit{node}$ or $p.y.\mathit{rnd}$), $32.q$ (which may change the value of $q.\mathit{usedrnd}$), and $29.r$ and $37.r$ (which may update $\mathit{Reset}[p.\mathit{node}].\mathit{rnd}$), where r is any arbitrary process. However, $p@\{20..23\}$ is false after the execution of statements 1.p, 3.p, 16.p, and 28.p. Statement 32.q preserves (I16), as shown above. By (I60), Statements 29.r and 37.r cannot be executed while $q@\{33\}$ holds. □

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow$
$\neg\big[p@\{17..20\} \wedge \big(q@\{3..8, 17..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)\big)\big]$ (I17)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is 16.p. The only statements that may falsify the consequent while the antecedent holds are 16.p (which may establish $p@\{17..20\}$), and 16.q and 23.q (which may establish $q@\{3..8, 17..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)$). Note that if statements 16.q and 23.q establish $q@\{3\}$, then they also establish $q.dir \neq stop$, which implies that $A(q,i)$ is false. Therefore, (I17) could potentially be falsified only if either 16.p or 16.q is executed, establishing $p@\{17\}$ or $q@\{17\}$, respectively. Without loss of generality, it suffices to consider only statement 16.p.

If $A(q,i) \wedge q@\{17..19\}$ holds before 16.p is executed, where $q \neq p$, then by the definition of $A(q,i)$, $X[i] = q$ holds as well. This implies that $X[i] \neq p$ holds both before and after 16.p is executed. Hence, $A(p,i)$ is false after the execution of 16.p.

Next, suppose that $A(q,i) \wedge \big(q@\{3..8, 20..22, 24\} \vee (q@\{9, 25..38\} \wedge q.n = i)\big)$ holds before 16.p is executed, where $q \neq p$. In this case, by (I1), $Y[i].free = false$. This implies that 16.p does not establish $p@\{17\}$. □

invariant $A(p,i) \wedge A(q,i) \wedge p \neq q \Rightarrow \neg(p@\{3..14, 21, 22, 24..41\} \wedge q@\{3..14, 21, 22, 24..41\})$ (I18)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is 16.p. However, if 16.p establishes $A(p,i)$, it also establishes $p@\{17\}$, and hence it cannot falsify (I18). Similar argument applies to 16.q.

By symmetry, when considering statements that might falsify the consequent, it suffices to consider only 16.p, 19.p, 20.p, and 23.p (which may establish $p@\{3..14, 21, 22, 24..41\}$). Statements 16.$p$, 19.$p$, and 23.$p$ can establish $p@\{3..14, 21, 22, 24..41\}$ only by establishing $p@\{3\}$, in which case they also establish $p.dir \neq stop$. This implies that $A(p,i)$ is false. Thus, these statements cannot falsify (I18). Similar reasoning applies if statement 20.p establishes $p@\{3\}$.

Statement 20.p could also establish $p@\{3..14, 21, 22, 24..41\}$ by establishing $p@\{21\}$. In this case, we have $Acquired[i] = false$ before its execution. If $A(p,i)$ is false before 20.p is executed, then by Lemma A1, it is also false afterward, and hence (I18) is not falsified. So, suppose that $A(p,i)$ is true before 20.p is executed. By (I17), this implies that $A(q,i) \wedge q@\{3..8, 21, 22, 24\}$ is false. Thus, 20.p could potentially falsify (I18) only if executed when $A(q,i) \wedge q@\{9..14, 25..41\}$ holds. However, in this case, by (I4), we have $Acquired[i] = true$. Thus, statement 20.p cannot falsify (I18). □

invariant $A(p,i) \wedge q.n = i \Rightarrow \neg(p@\{17..19\} \wedge q@\{28..38\})$ (I19)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is 16.p, which may do so only if $Y[i].free = true$. By (I25), this implies that $q@\{28..38\} \wedge q.n = i$ is false. Therefore, statement 16.p cannot falsify (I19). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I19).

The only other statement that may falsify (I19) is 27.q (which establishes $q@\{28..38\}$), which may do so only if executed when $q.n = i$ holds. In this case, it falsifies $X[i] = p$, which implies that $A(p,i) \wedge p@\{17..19\}$ is false as well. □

invariant $A(p,i) \wedge q.n = i \Rightarrow \neg(p@\{20, 21\} \wedge q@\{30..38\})$ (I20)

Proof: By Lemma A1, the only statement that can establish $A(p,i)$ is 16.p, which establishes $p@\{17, 3\}$. Hence, it cannot falsify (I20). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I20).

The only other statements that may falsify (I20) are $19.p$ (which may establish $p@\{20, 21\}$) and $29.q$ (which establishes $q@\{30..38\}$). Statement $19.p$ may falsify (I20) only by establishing $p@\{20\}$, which it does only if $X[p.node] = p$. By (I5), this implies that $p@\{19\} \land X[p.node] = p \land \mathit{Reset}[p.node] = p.y$ holds before its execution. Thus, $19.p$ may potentially falsify (I20) only if executed when $A(p, p.node)$ holds. If $i \neq p.node$, then $A(p, i)$ is clearly false both before and after the execution of $19.p$. If $i = p.node$, then $A(p, i)$ holds before $19.p$ is executed, which implies that $q@\{30..38\} \land q.n = i$ is false, by (I19). Thus, statement $19.p$ cannot falsify (I20).

Statement $29.q$ may falsify (I20) only if executed when $q.n = i \land p@\{20, 21\}$ holds. By (I28), this implies that $p.y.\mathit{free} = \mathit{true}$ holds. Because statement $29.q$ establishes $\mathit{Reset}[i].\mathit{free} = \mathit{false}$, this implies that $p@\{20, 21\} \land \mathit{Reset}[i] \neq p.y$ is established, *i.e.*, $A(p, i)$ is false after its execution. □

invariant $A(p, i) \land q.n = i \Rightarrow \neg(p@\{3..10, 22, 24\} \land q@\{31..38\})$ (I21)

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $16.p$. However, if $16.p$ establishes $A(p, i)$, then it also establishes $p@\{17\}$, and hence cannot falsify (I21). Any statement that updates $q.n$ also establishes $q@\{9, 15\}$, and hence cannot falsify (I21).

The only other statements that may falsify (I21) are $16.p$, $19.p$, $20.p$, $21.p$, and $23.p$ (which may establish $p@\{3..10, 22, 24\}$), and $30.q$ (which may establish $q@\{31..38\}$).

Statements $16.p$, $19.p$, $20.p$, and $23.p$ can establish $p@\{3..10, 22, 24\}$ only by establishing $p@\{3\}$, in which case they also establish $p.\mathit{dir} \neq \mathit{stop}$. This implies that $A(p, i)$ is false. Statement $21.p$ can neither establish nor falsify $A(p, i)$. Thus, it may falsify (I21) only if executed when $A(p, i) \land q.n = i \land q@\{31..38\}$ holds, but this is precluded by (I20).

Statement $30.q$ may falsify (I21) only if executed when

$$q@\{30\} \land A(p, i) \land p@\{3..10, 22, 24\} \land q.n = i \land (q.\mathit{dir} = \mathit{stop} \lor \mathit{Round}[q.y.\mathit{rnd}] = \mathit{false})$$

holds. By (I60), $p@\{8..10\}$ and $q@\{30\}$ cannot hold simultaneously. So, assume that the following assertion holds prior to the execution of $30.q$.

$$q@\{30\} \land A(p, i) \land p@\{3..7, 22, 24\} \land q.n = i \land (q.\mathit{dir} = \mathit{stop} \lor \mathit{Round}[q.y.\mathit{rnd}] = \mathit{false}) \tag{2}$$

If $q.\mathit{dir} = \mathit{stop}$ holds, then by (I33), $q.n = q.\mathit{node}$. Because $q.n = i$, this implies that $A(q, i)$ holds. By (2), this implies that $A(p, i) \land A(q, i) \land p@\{3..7, 22, 24\} \land q@\{30\}$ holds. However, this is precluded by (I18).

The only other possibility is that

$$q@\{30\} \land A(p, i) \land p@\{3..7, 22, 24\} \land q.n = i \land \mathit{Round}[q.y.\mathit{rnd}] = \mathit{false}$$

holds before $30.q$ is executed. In this case, $\mathit{Reset}[q.n].\mathit{rnd} = q.y.\mathit{rnd}$ holds, by (I24), and $\mathit{Reset}[i].\mathit{rnd} = p.y.\mathit{rnd}$ also holds, by the definition of $A(p, i)$ and (I3). Thus, $q.y.\mathit{rnd} = p.y.\mathit{rnd}$. In addition, by (I2), $\mathit{Round}[p.y.\mathit{rnd}] = \mathit{true}$. Thus, $\mathit{Round}[q.y.\mathit{rnd}] = \mathit{true}$, which is a contradiction. □

invariant $A(p, i) \land \big(p@\{3..8, 20..22, 24\} \lor (p@\{9, 25..38\} \land p.n = i)\big) \land q@\{20..23\} \land p \neq q \Rightarrow$
$p.y.\mathit{rnd} \neq q.y.\mathit{rnd}$ (I22)

Proof: The only statements that may falsify the consequent are $16.p$ and $28.p$ (which may change the value of $p.y.\mathit{rnd}$) and $16.q$ and $28.q$ (which may change the value of $q.y.\mathit{rnd}$). However, the antecedent is false after the execution of $16.q$ and $28.q$. Also, $16.p$ either establishes $p@\{17\}$ or $p@\{3\} \land p.\mathit{dir} = \mathit{right}$. The latter implies that $A(p, i)$ is false. Thus, statement $16.p$ cannot falsify (I22).

Statement $28.p$ may falsify (I22) only if executed when $A(p, i) \land p.n = i$ holds. In this case, $\mathit{Reset}[i].\mathit{rnd} = p.y.\mathit{rnd}$ holds, by (I3). It follows that statement $28.p$ does not change the value of $p.y.\mathit{rnd}$ if executed when the antecedent holds.

The antecedent may be established only by statements $16.p$ (which, by Lemma A1, may establish $A(p,i)$ and may also establish $p@\{3..8, 20..22, 24\}$), $19.p$ and $23.p$ (which may establish $p@\{3..8, 20..22, 24\}$), and $19.q$ (which may establish $q@\{20..23\}$). However, as shown above, statement $16.p$ cannot falsify (I22).

Statement $19.q$ can establish $q@\{20\}$ only if $q@\{19\} \wedge X[q.node] = q$ holds. If $i = q.node$, then by (I5), $A(q,i)$ holds as well. However, by (I17) (with p and q exchanged), this implies that the antecedent of (I22) is false. Thus, $19.q$ cannot falsify (I22) in this case.

On the other hand, if $q@\{19\} \wedge X[q.node] = q \wedge i \neq q.node$ holds, then we have $q.y = Reset[q.node]$, by (I5), and $Reset[q.node].rnd \neq Reset[i].rnd$, by (I15). If the antecedent of (I22) is true, then we have either $A(p,i) \wedge p@\{3..8, 20..22, 24\} \vee (p@\{9, 25..37\} \wedge p.n = i)$ or $p@\{38\}$. In the former case, by (I3) and the definition of $A(p,i)$, we have $p.y.rnd = Reset[i].rnd$. In the latter case, by (I13), we have $p.y.rnd \neq Reset[q.node].rnd$. Thus, in either case, $p.y.rnd \neq q.y.rnd$ holds. Hence, statement $19.q$ cannot falsify (I22).

Statements $19.p$ and $23.p$ are the remaining statements to consider. Statement $23.p$ establishes $p@\{3\} \wedge p.dir = left$, which implies that $A(p,i)$ is false. Statement $19.p$ establishes either $p@\{3\} \wedge p.dir = left$ or $p@\{20\}$. In the former case, $A(p,i)$ is false. In the latter case, note that statement $19.p$ may falsify (I22) only if executed when $q@\{20..23\}$ holds. In addition, by Lemma A1, $19.p$ can establish $A(p,i) \wedge p@\{20\}$ only if executed when $A(p,i) \wedge p@\{19\} \wedge X[i] = p$ is true. By (I28) and the definition of $A(p,i)$, we therefore have the following prior to the execution of $19.p$.

$$q@\{20..23\} \wedge A(p,i) \wedge p@\{19\} \wedge X[i] = p \wedge p.y = Reset[i] \wedge Reset[i].free = true \tag{3}$$

By applying (I6), (I7), and (I8) to $q@\{20..23\}$, we also have either $q.y.rnd = Reset[q.node].rnd$ or $\mathsf{Dist}[q.y.rnd] \neq \bot$. This gives us two cases to analyze.

- $q.y.rnd = Reset[q.node].rnd$. Note that statement $19.p$ can falsify (I22) only if executed when the following holds.

  $$p.y.rnd = q.y.rnd \tag{4}$$

  By (3), (4), and our assumption that $q.y.rnd = Reset[q.node].rnd$ holds, we have $Reset[i].rnd = Reset[q.node].rnd$. By (I15), this implies that $i = q.node$.

  If $q@\{20..22\}$ holds before the execution of $19.p$, then by (3) and (I17), $A(q,i)$ is false. By the definition of $A(q,i)$ this implies that $Reset[i] \neq q.y$. By (3), this implies that $p.y \neq q.y$. In addition, by (I28), we have $q.y.free = true$. By (3), this implies that $p.y.rnd \neq q.y.rnd$, which contradicts (4).

  On the other hand, if $q@\{23\}$ holds before the execution of $19.p$, then we have $Reset[q.node].free = false$, by (I10). Because $i = q.node$, this contradicts (3).

- $\mathsf{Dist}[q.y.rnd] \neq \bot$. By Lemma A1, statement $19.p$ may establish the antecedent only if $A(p,i)$ holds before its execution. By (I19), this implies that $\neg(\exists r :: r@\{36, 37\} \wedge r.n = i)$ holds. Hence, by (I14), $\mathsf{Dist}[Reset[i].rnd] = \bot$ holds as well. By (3), we therefore have $\mathsf{Dist}[p.y.rnd] = \bot$. Because $\mathsf{Dist}[p.y.rnd] = \bot$ and $\mathsf{Dist}[q.y.rnd] \neq \bot$ both hold, we have $p.y.rnd \neq q.y.rnd$. It follows that statement $19.p$ cannot falsify (I22). $\square$

invariant $p@\{29\} \Rightarrow Reset[p.n] = p.y$ (I23)
invariant $p@\{30..37\} \Rightarrow Reset[p.n] = (false, p.y.rnd)$ (I24)
invariant $p@\{27..38\} \Rightarrow Y[p.n] = (false, 0)$ (I25)
invariant $Y[i].free = true \Rightarrow Y[i] = Reset[i]$ (I26)
invariant $Reset[i].rnd \neq 0$ (I27)
invariant $p@\{17..23\} \Rightarrow p.y.free = true$ (I28)
invariant $p@\{17..23, 29..40\} \Rightarrow p.y.rnd \neq 0$ (I29)
invariant $p@\{19..23\} \Rightarrow Inuse[p] = p.y.rnd$ (I30)

Proof: By (I60), all writes to *Reset* (statements 29 and 37) and to Y, except for statement 17 (statements 26 and 38), and all operations involving the *Free* queue (statements 33, 35, and 36) occur within mutually exclusive regions of code. Given this and the initial condition $(\forall i, p :: Y[i] = (true, i) \wedge Reset[i] = (true, i)) \wedge$

Free $= (T+1) \rightarrow \cdots \rightarrow S$, each of these invariants easily follows. Note that statement 17 establishes $Y[i] = (\mathit{false}, 0)$, and hence cannot falsify either (I25) or (I26). Note also that (I25) implies that statements 29 and 37 cannot falsify (I26). □

invariant $p@\{2,3,15..24\} \Rightarrow (0 \le p.\mathit{level} \le L) \wedge (1 \le p.\mathit{node} \le T)$ (I31)

invariant $p@\{9,25..40\} \Rightarrow$
$1 \le p.n \le T \wedge p.n = p.\mathit{path}[p.j].\mathit{node} \wedge p.\mathit{dir} = p.\mathit{path}[p.j].\mathit{dir} \wedge p.j = \mathsf{lev}(p.n)$ (I32)

invariant $p@\{9,25..40\} \wedge p.\mathit{dir} = \mathit{stop} \Rightarrow p.n = p.\mathit{node}$ (I33)

invariant $p@\{9,25..40\} \wedge p.n = p.\mathit{node} \Rightarrow p.\mathit{dir} = \mathit{stop}$ (I34)

invariant $p@\{2..41\} \Rightarrow (0 \le p.\mathit{level} \le L+1) \wedge (\mathsf{lev}(p.\mathit{node}) = p.\mathit{level})$ (I35)

invariant $p@\{39,40\} \Rightarrow p.n = p.\mathit{node}$ (I36)

Proof: These invariants easily follow from the program text and structure of the renaming tree. □

invariant $\neg(\exists p :: p@\{36\}) \Rightarrow \|\mathit{Free}\| = 2N$ (I37)

invariant $(\exists p :: p@\{36\}) \Rightarrow \|\mathit{Free}\| = 2N+1$ (I38)

invariant $p@\{32..34\} \Rightarrow p.\mathit{ptr} = \mathit{Check}$ (I39)

invariant $p@\{37,38\} \Rightarrow \mathsf{Dist}[p.\mathit{nextrnd}] = \perp$ (I40)

invariant $p@\{36\} \Rightarrow \mathsf{Dist}[p.y.\mathit{rnd}] = 2N$ (I41)

invariant $p@\{37\} \Rightarrow \mathsf{Dist}[p.y.\mathit{rnd}] = 2N-1$ (I42)

Proof: Note that every statement that accesses the *Free* queue (statements 33, 35, and 36) executes within a mutually exclusive region of code. From this and the initial condition, $\|\mathit{Free}\| = 2N$, these invariants easily follow. Note also that by (I24), if $p@\{35\}$ holds, then $p.y.\mathit{rnd} = \mathit{Reset}[p.n].\mathit{rnd}$ holds. By (I14) and (I60), this in turn implies that $\mathsf{Dist}[p.y.\mathit{rnd}] = \perp$ holds. Hence, statement 35.p does not enqueue a duplicate entry onto the *Free* queue. □

invariant (Mutual Exclusion) $\big|\{p :: p@\{8..10,13\}\}\big| \le 1$ (I60)

Proof: From the specification of the `ENTRY` and `EXIT` routines, (I60) could be falsified only if two processes p and q stop at the same splitter in the renaming tree, *i.e.*, we have $p \ne q \wedge p@\{4,5,8..11\} \wedge q@\{4,5,8..11\} \wedge p.\mathit{node} = q.\mathit{node}$. However, this is precluded by (I18). □

Proof of Contention Sensitivity

The remaining invariants are needed to establish contention sensitivity (I61). These invariants formalize the following rather intuitive reasoning: for a process p to reach splitter i in the renaming tree by moving right (left) from i's parent, some other process must have either stopped or moved left (right) at i's parent. From this, it follows that the depth to which p descends in the renaming tree is proportional to the point contention that p experiences.

invariant $(1 \le i \le S) \wedge \mathit{Round}[i] = \mathit{true} \Rightarrow$
$\big(\exists p :: p.y.\mathit{rnd} = i \wedge \big(p@\{3..5,8,22..24\} \vee (p@\{9,25..39\} \wedge p.n = p.\mathit{node})\big) \wedge$
$\big(p@\{3..5,8\} \Rightarrow (p.\mathit{dir} = \mathit{stop} \wedge 1 \le p.\mathit{node} \le T)\big)\big)$ (I43)

Proof: The antecedent may be established only by statement 21.p, which also establishes the consequent.

Suppose that

$$p.y.\mathit{rnd} = i \wedge \big(p@\{3..5,8,22..24\} \vee (p@\{9,25..39\} \wedge p.n = p.\mathit{node})\big) \wedge \big(p@\{3..5,8\} \Rightarrow (p.\mathit{dir} = \mathit{stop} \wedge 1 \le p.\mathit{node} \le T)\big) \quad (5)$$

holds. We consider each condition separately. The only statements that may falsify $p.y.rnd = i$ are 16.p and 28.p. Statement 16.p cannot be executed while (5) holds. If statement 28.p is executed while (5) holds, then by (I3), it does not change the value of $p.y.rnd$.

The only statements that may falsify $p@\{3..5, 8, 22..24\}$ are 3.p and 8.p. If statement 3.p is executed while (5) holds, then by (I35), we also have $p.level \leq L$, and hence statement 3.p establishes $p@\{4\}$. Similarly, if statement 8.p is executed while (5) holds, then it establishes $p@\{9\} \wedge p.n = p.node$. Thus, these statements cannot falsify (5).

The only statements that may falsify $p@\{9, 25..39\} \wedge p.n = p.node$ are 25.p, 30.p, 38.p, and 39.p. By (I34), statements 25.p, 30.p, and 38.p establish $p@\{26, 31, 39\}$. If $p.y.rnd = i$, then statement 39.p falsify the antecedent of (I43).

The only statements that may falsify $p@\{3..5, 8\} \Rightarrow (p.dir = stop \wedge 1 \leq p.node \leq T)$ are 1.p and 3.p (which may update $p.node$), 7.p (which establishes $p@\{3..5, 8\}$), and 16.p, 19.p, 20.p, 23.p, and 24.p (which may establish $p@\{3..5, 8\}$ and also update $p.dir$). Statements 1.p, 7.p, 16.p, 19.p, and 20.p cannot be executed while (5) holds. If statement 3.p is executed while $p.dir = stop$ holds, then it does not update $p.node$. If $p.y.rnd = i$, then statement 23.p falsifies the antecedent of (I43). By (I31), statement 24.p establishes $p.dir = stop \wedge 1 \leq p.node \leq T$, and hence preserves (5). □

invariant $p@\{4..14, 25..41\} \wedge (p.node \leq T) \Rightarrow p.path[p.level] = (p.node, stop)$ (I44)

Proof: The antecedent of (I44) can only be established by statement 3.p. By (I31), it does so only if $p.level = L \vee p.dir = stop$ holds prior to its execution. If $p.dir = stop$ holds, then 3.p establishes the consequent of (I44). If $p.level = L \wedge p.dir \neq stop$ holds before 3.p is executed, then by (I35), $p.node > T$ holds afterward. No statement can falsify the consequent of (I44) while the antecedent holds. □

invariant $p@\{2..41\} \wedge (2i \xrightarrow{*} p.node) \Rightarrow p.path[\mathsf{lev}(i)] = (i, left)$ (I45)

Proof: The only statements that may falsify (I45) are 1.p and 3.p. Statement 1.p establishes $p.node = 1$, which implies that $2i \xrightarrow{*} p.node$ is false. Statement 3.p may establish $2i \xrightarrow{*} p.node$ only if it also establishes $2i = p.node$, which can happen only if it is executed when $p.node = i \wedge p.dir = left$ holds. In this case, by (I35), statement 3.p establishes the consequent. □

invariant $p@\{3\} \wedge (p.level > 0) \Rightarrow$
$$p.path[0].node = 1 \wedge$$
$$(\forall l : 0 \leq l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node) \wedge$$
$$p.path[p.level - 1].node \xrightarrow{*} p.node$$ (I46)

invariant $p@\{26..38\} \Rightarrow (p.n = p.node) \vee (2 \cdot p.n \xrightarrow{*} p.node)$ (I47)

Proof: Each iteration of the **repeat** loop of statements 2–3 descends one level in the renaming tree, starting with splitter 1 (the root). When descending from level l, $p.path[l]$ is updated (statement 3) to indicate the splitter visited at level l. From this, invariant (I46) easily follows.

The **repeat** loop of statements 2–3 terminates only if $p.level \geq L \vee p.dir = stop$ holds prior to the execution of statement 3.p. If $p.dir = stop$ holds when 3.p is executed, then when p executes within statements 4–14 and 25–41, $p.node$ equals the splitter at which it stopped. If $p.level \geq L \wedge p.dir \neq stop$ holds when 3.p is executed, then when p executes within statements 4–14 and 25–41, $p.node$ equals a splitter at level $L + 1$ (in which case there is no actual splitter corresponding to $p.node$). In either case, the **for** loop at statement 9 will ascend the renaming tree, visiting only $p.node$ and its ancestors. The corresponding splitter is indicated by the variable $p.n$. Moreover, if p moved right while descending the renaming tree, then p returns from *ReleaseNode* at statement 25. If p stopped at that splitter, then $p.n = p.node$ holds. If p moved left from that splitter, then $2 \cdot p.n \xrightarrow{*} p.node$ holds. From these observations, it should be clear that (I47) is an invariant. □

invariant $p@\{0,1\} \;\Rightarrow\; \mathsf{Loc}[p]=0 \;\wedge\; \mathsf{Loc}[p+N]=0$ (I48)

invariant $p@\{2,15\} \;\Rightarrow\; \mathsf{Loc}[p]=p.node \;\wedge\; \mathsf{Loc}[p+N]=p.node$ (I49)

invariant $p@\{16\} \;\Rightarrow\; \big(X[p.node]=p \;\Rightarrow\; \mathsf{Loc}[p]=p.node \;\wedge\; \mathsf{Loc}[p+N]=p.node\big) \;\wedge$
$\big(X[p.node]\neq p \;\Rightarrow\; \mathsf{Loc}[p]=2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N]=2\cdot p.node\big)$ (I50)

invariant $p@\{17..24\} \;\Rightarrow\; \big(A(p,p.node) \;\Rightarrow\; \mathsf{Loc}[p]=p.node \;\wedge\; \mathsf{Loc}[p+N]=p.node\big) \;\wedge$
$\big(\neg A(p,p.node) \;\Rightarrow\; \mathsf{Loc}[p]=2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N]=2\cdot p.node\big)$ (I51)

invariant $p@\{3\} \;\Rightarrow\; \big(p.dir=stop \;\Rightarrow\; \mathsf{Loc}[p]=p.node \;\wedge\; \mathsf{Loc}[p+N]=p.node\big) \;\wedge$
$\big(p.dir=left \;\Rightarrow\; \mathsf{Loc}[p]=2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N]=2\cdot p.node\big) \;\wedge$
$\big(p.dir=right \;\Rightarrow\; \mathsf{Loc}[p]=2\cdot p.node+1 \;\wedge\; \mathsf{Loc}[p+N]=2\cdot p.node+1\big)$ (I52)

invariant $p@\{4..14,25..41\} \;\Rightarrow\; \mathsf{Loc}[p+N]=p.node$ (I53)

invariant $p@\{4..8\} \;\Rightarrow\; \mathsf{Loc}[p]=p.node$ (I54)

invariant $p@\{9,25\} \;\Rightarrow\; p.n \xrightarrow{*} \mathsf{Loc}[p]$ (I55)

invariant $p@\{26..40\} \;\Rightarrow\; \mathsf{Loc}[p]=p.n$ (I56)

Proof: These invariants easily follow from the program text and the structure of the renaming tree. Note that $p@\{16\}$ is established only if $X[p.node]=p$ holds. Also note that whenever $X[p.node]=p$ is falsified by either $15.q$ or $27.q$, where q is any arbitrary process, q also establishes $\mathsf{Loc}[p]=2\cdot p.node \;\wedge\; \mathsf{Loc}[p+N]=2\cdot p.node$.

Statement $16.p$ may establish $p@\{17\}$ only if executed when $Y[p.node].free = true$ holds. By (I26), this implies that $Y[p.node]=Reset[p.node]$ holds. From this condition, the consequent of (I50), and the definition of $A(p,i)$, it follows that if statement $16.p$ establishes $p@\{17\}$, then the consequent of (I51) is true.

$A(p,p.node)$ potentially could be falsified by some process $q\neq p$ only by executing one of the statements $15.q$, $27.q$, $29.q$, or $37.q$. However, by (I24), $q@\{37\}$ implies that $Reset[q.n].free = false$ holds. Moreover, by (I28), $p@\{17..22\}$ implies $p.y.free = true$. It follows that statement $37.q$ cannot change the value of $A(p,p.node)$ from true to false. If one of $15.q$, $27.q$, and $29.q$ falsifies $A(p,p.node)$, then it also assigns the value $2\cdot p.node$ to each of $\mathsf{Loc}[p]$ and $\mathsf{Loc}[p+N]$. □

invariant $Y[i].free=false \;\Rightarrow\; \mathcal{A}\text{: } \big(\exists p :: \big(p@\{3..8,18..24\} \;\vee\; (p@\{9,25..38\} \;\wedge\; p.n=i)\big) \;\wedge$
$(p.node=i) \;\wedge\; (p@\{3\} \;\Rightarrow\; p.dir\neq right)\big) \;\vee$
$\mathcal{B}\text{: } \big(\exists p :: p@\{2..9,15..40\} \;\wedge\; (2i \xrightarrow{*} p.node) \;\wedge$
$(p@\{9,25..40\} \;\Rightarrow\; 2i \xrightarrow{*} p.n)\big) \;\vee$
$\mathcal{C}\text{: } \big(\exists p :: p@\{9,25..38\} \;\wedge\; p.n=i \;\wedge\; (p@\{9,25\} \;\Rightarrow\; 2i \xrightarrow{*} \mathsf{Loc}[p])\big)$ (I57)

Proof: Note that we may assume $1\le i\le T$, because $Y[i]$ is defined only for values of i in this range.

The only statements that can establish the antecedent are $17.q$ and $26.q$, where q is any arbitrary process. Statement $17.q$ establishes disjunct $\mathcal{A}$. If statement $26.q$ establishes the antecedent, then disjunct $\mathcal{C}$ holds both before and after its execution. We now consider statements that may falsify each of the three disjuncts of the consequent.

Disjunct $\mathcal{A}$: Suppose that the following assertion holds.

$$\big(p@\{3..8,18..24\} \;\vee\; (p@\{9,25..38\} \;\wedge\; p.n=i)\big) \;\wedge\; (p.node=i) \;\wedge\; (p@\{3\} \;\Rightarrow\; p.dir\neq right) \qquad (6)$$

We consider each condition separately. The only statements that may falsify $p@\{3..8,18..24\}$ are $3.p$ and $8.p$. If $3.p$ is executed while $p.node=i \;\wedge\; p.dir=stop$ holds, then it establishes $p@\{4\} \;\wedge\; p.node=i$, and hence preserves (6). On the other hand, if $3.p$ is executed while $p.node=i \;\wedge\; p.dir=left$ holds, then it establishes $p@\{2,6\} \;\wedge\; p.node=2i$, and hence disjunct $\mathcal{B}$ is established. If statement $8.p$ is executed while (6) holds, then since $i\le T$, by (I35), $p.level\le L$ holds before and after its execution. Thus, in this case, $8.p$ establishes $p@\{9\} \;\wedge\; p.n=i$. It follows that these statements cannot falsify (6).

The only statements that may falsify $p@\{9,25..38\} \;\wedge\; p.n=i$ are $25.p$, $30.p$, and $38.p$. By (I34), statements $25.p$ and $30.p$ establish $p@\{26,31\}$. If $p.n=i$, then statement $38.p$ falsifies the antecedent of (I57).

The only statements that may update $p.node$ are $1.p$ and $3.p$. Statement $1.p$ cannot be executed while (6) holds. Statement $3.p$ preserves (I57) as shown above.

The only statement that may falsify $p@\{3\} \Rightarrow p.dir \neq right$ is $16.p$, which cannot be executed while (6) holds.

Disjunct $\mathcal{B}$: Suppose that the following assertion holds.

$$p@\{2..9, 15..40\} \wedge (2i \xrightarrow{*} p.node) \wedge (p@\{9, 25..40\} \Rightarrow 2i \xrightarrow{*} p.n) \tag{7}$$

We consider each condition separately. The only statements that may falsify $p@\{2..9, 15..40\}$ are $25.p$, $30.p$, $38.p$, and $40.p$ (which may return from *ReleaseNode* and terminate the **for** loop at statement 9). However, since $2i \xrightarrow{*} p.n$ implies $\mathsf{lev}(p.n) > 1$, the **for** loop must iterate again, establishing $p@\{9\}$. If these statements preserve $2i \xrightarrow{*} p.n$, then (7) is preserved. Otherwise, p ascends the renaming tree by a level, and $p@\{9\} \wedge p.n = i$ is established. Moreover, by (I55) and (I56), $2i \xrightarrow{*} \mathsf{Loc}[p]$ holds before and after the execution of each of these statements. This implies that disjunct $\mathcal{C}$ holds.

The only statements that may update $p.node$ are $1.p$ and $3.p$. Statement $1.p$ cannot be executed while (7) holds. Statement $3.p$ only updates $p.node$ as the renaming tree is descended. Thus, it cannot falsify $2i \xrightarrow{*} p.node$.

The only statements that may falsify $p@\{9, 25..40\} \Rightarrow 2i \xrightarrow{*} p.n$ are $8.p$ (which may establish $p@\{9, 25..40\}$ and also update $p.n$) and $25.p$, $30.p$, $38.p$, and $40.p$ (which may return from *ReleaseNode* and update $p.n$). If statement $8.p$ establishes $p@\{9\}$, then it also establishes $p.n = p.node$. Thus, it cannot falsify (7). Statements $25.p$, $30.p$, $38.p$, and $40.p$ preserve (I57) as shown above.

Disjunct $\mathcal{C}$: Suppose that the following assertion holds.

$$p@\{9, 25..38\} \wedge p.n = i \wedge (p@\{9, 25\} \Rightarrow 2i \xrightarrow{*} \mathsf{Loc}[p]) \tag{8}$$

This assertion implies that $p@\{9, 25..38\}$ holds. Thus, it may be falsified only by statement $25.p$ (which updates $\mathsf{Loc}[p]$ and may falsify $p@\{9, 25..38\}$ — note that no other process can modify $\mathsf{Loc}[p]$ while $p@\{9, 25..38\}$ holds), and statements $30.p$ and $38.p$ (which may falsify $p@\{9, 25..38\}$, establish $p@\{9, 25\}$, or modify $p.n$). Statement $25.p$ may falsify $p@\{9, 25..38\}$ only if executed when $p.dir = right$ holds. By (I45) and (I55), $p@\{25\} \wedge 2i \xrightarrow{*} \mathsf{Loc}[p]$ implies $p.path[\mathsf{lev}(i)] = (i, \mathit{left})$ — informally, p moved left from splitter i when descending the renaming tree. Thus, by (I32), $p.dir = \mathit{left}$ holds before the execution of $25.p$. It follows that statement $25.p$ establishes $p@\{26\}$, and hence it cannot falsify (8). Statement $38.p$ falsifies the antecedent of (I57) if executed when $p.n = i$ holds.

Statement $30.p$ may falsify (8) only if executed when $\mathit{Round}[p.y.rnd] = \mathit{true}$ holds. So, assume that $p@\{30\} \wedge \mathit{Round}[p.y.rnd] = \mathit{true}$ holds. Then, by (I43), there exists a process q such that

$$\begin{aligned}&\big(q@\{3..5, 8, 22..24\} \vee (q@\{9, 25..39\} \wedge q.n = q.node)\big) \wedge\\ &\big(q@\{3..5, 8\} \Rightarrow (q.dir = stop \wedge 1 \leq q.node \leq T)\big) \wedge\\ &q.y.rnd = p.y.rnd.\end{aligned} \tag{9}$$

If $q = p$, then by (9) and our assumption that $p@\{30\}$ holds, we have $p.n = p.node$, which implies, by (I34), that statement $30.p$ establishes $p@\{31\}$, preserving (8).

So, suppose that $q \neq p$. By (I24), we have $\mathit{Reset}[i].rnd = p.y.rnd$, which by (9) implies that

$$\mathit{Reset}[i].rnd = q.y.rnd.$$

In addition, by (I60) and our assumption that $p@\{30\}$ holds, we have $q@\{3..5, 22..24\}$. We now prove that $\mathit{Reset}[q.node].rnd = q.y.rnd$ holds by considering the two cases $q@\{22, 23\}$ and $q@\{3..5, 24\}$ separately. First, suppose that $q@\{22, 23\}$ holds. Then, by (I8) and (I60), we have either $\mathit{Reset}[q.node].rnd = q.y.rnd$ or $\mathsf{Dist}[q.y.rnd] \neq \bot$. However, because $\mathit{Reset}[i].rnd = q.y.rnd \wedge p@\{30\}$ holds, by (I14) and (I60), $\mathsf{Dist}[q.y.rnd] = \bot$ holds. Thus, in this case, we have $\mathit{Reset}[q.node].rnd = q.y.rnd$.

On the other hand, if $q@\{3..5, 24\}$ holds, then by (I31) and (9), $A(q, q.node)$ holds as well. By (I3), this implies that $Reset[q.node].rnd = q.y.rnd$ holds.

Putting these assertions together, we have $Reset[i].rnd = Reset[q.node].rnd$. By (I15), this implies that $q.node = i$. Therefore, if $q \neq p$, we have $q@\{3..5, 22..24\} \ \wedge \ q.node = i \ \wedge \ (q@\{3\} \ \Rightarrow \ q.dir = stop)$, which implies that disjunct $\mathcal{A}$ holds both before and after the execution of $30.p$. □

The following lemma is used in proving invariants (I58) and (I59).

Lemma A2: If t and u are consecutive states such that $p@\{2, 3, 15..24\}$ holds at t, the condition $i \xrightarrow{*} \mathsf{Loc}[p]$ holds at u but not at t, and all the invariants in this appendix hold at t, then the following are true:

- The value of $\mathsf{Loc}[p]$ is changed by a call to $\mathsf{UpdateLoc}(P, i)$ such that $p \in P$.
- $i = \mathsf{Loc}[p] \ \wedge \ C(\lfloor i/2 \rfloor) > C(i)$ holds at u.

Proof: The only statements that may establish $i \xrightarrow{*} \mathsf{Loc}[p]$ while $p@\{2, 3, 15..24\}$ holds are $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. It should be obvious that these statements can update $\mathsf{Loc}[p]$ only by calling $\mathsf{UpdateLoc}(P, i)$, where $p \in P$. Now we show that if state u is reached via the execution of any of these statements, then $i = \mathsf{Loc}[p]$ is established.

By using (I50) and (I51), we can tabulate all the possible ways in which $\mathsf{Loc}[p]$ can be changed by one of these statements. Such a tabulation is given below. Note that the table only shows the ways in which $\mathsf{Loc}[p]$ may *change* value. For example, by (I50), if $p@\{16\}$ holds, then $\mathsf{Loc}[p]$ is either $p.node$ or $2 \cdot p.node$. In both cases, statement $16.p$ may change the value of $\mathsf{Loc}[p]$ only by establishing $\mathsf{Loc}[p] = 2 \cdot p.node + 1$. As another example, by (I51), if $p@\{19\}$ holds, then $\mathsf{Loc}[p]$ is either $p.node$ or $2 \cdot p.node$. Because $19.p$ can change the value of $\mathsf{Loc}[p]$ only by establishing $\mathsf{Loc}[p] = 2 \cdot p.node$, we do not include an entry for $\mathsf{Loc}[p] = 2 \cdot p.node$ in the column for state t.

statement	at state t	at state u
$16.p$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node + 1$
	$\mathsf{Loc}[p] = 2 \cdot p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node + 1$
$19.p$, $20.p$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$
$15.q$, $27.q$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$
$29.q$	$\mathsf{Loc}[p] = p.node$	$\mathsf{Loc}[p] = 2 \cdot p.node$

From this table, we see that there are only three ways in which the value of $\mathsf{Loc}[p]$ can be changed: **(i)** from a parent ($p.node$) to its left child ($2 \cdot p.node$), **(ii)** from a parent ($p.node$) to its right child ($2 \cdot p.node + 1$), and **(iii)** from a left child ($2 \cdot p.node$) to its right sibling ($2 \cdot p.node + 1$). It follows that $i \xrightarrow{*} \mathsf{Loc}[p]$ holds at u but not at t if and only if $i = \mathsf{Loc}[p]$ is established in transiting from t to u.

Our remaining proof obligation is to show that $C(\lfloor i/2 \rfloor) > C(i)$ holds at u. Let $h = \lfloor i/2 \rfloor$. Note that h is splitter i's parent. (Note further that, because $i \xrightarrow{*} \mathsf{Loc}[p]$ does not hold at t, and because the root is an ancestor of every splitter, splitter i is not the root, *i.e.*, its parent does exist.) We will establish $C(h) > C(i)$ by showing that some process "located" within the subtree rooted at h is either at h or in the subtree rooted at i's sibling. Note that in each of cases **(i)** through **(iii)**, we have

$$h = p.node \ \wedge \ (i = 2 \cdot p.node \ \vee \ i = 2 \cdot p.node + 1). \tag{10}$$

We now show that $C(\lfloor i/2 \rfloor) > C(i)$ holds at u by considering each of statements $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$ separately. Statement $16.p$ may establish $\mathsf{Loc}[p] = i$, where

$$i = 2 \cdot p.node + 1 \tag{11}$$

only if executed when $Y[h].free = false$, in which case the antecedent of (I57) holds. Thus, one of the three disjuncts of the consequent of (I57) holds.

- If disjunct $\mathcal{A}$ holds, then by (I51)–(I53), there exists q such that $q@\{3..9, 18..38\} \wedge (\mathsf{Loc}[q+N] = h \vee \mathsf{Loc}[q+N] = 2h)$.
- If disjunct $\mathcal{B}$ holds, then by (I49)–(I53), there exists q such that $q@\{2..9, 15..40\} \wedge 2h \xrightarrow{*} \mathsf{Loc}[q+N]$.
- If disjunct $\mathcal{C}$ holds, then by (I56), there exists q such that $(q@\{9,25\} \wedge 2h \xrightarrow{*} \mathsf{Loc}[q]) \vee (q@\{26..38\} \wedge \mathsf{Loc}[q] = q.n \wedge q.n = h)$.

If $q = p$, then the condition $p@\{16\}$ implies that disjunct $\mathcal{B}$ must hold. By (10), this implies that $2h = 2 \cdot p.node \xrightarrow{*} q.node$ holds, which contradicts $q = p$. Hence, $q \neq p$, and thus (by the program text) statement $16.p$ does not change the value of either $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q+N]$. Because one of $\mathcal{A}$ through $\mathcal{C}$ holds before $16.p$ is executed, the following assertion holds both before and after the execution of $16.p$.

$$q \neq p \wedge \big(\mathsf{Loc}[q] = h \vee \mathsf{Loc}[q+N] = h \vee 2h \xrightarrow{*} \mathsf{Loc}[q] \vee 2h \xrightarrow{*} \mathsf{Loc}[q+N]\big)$$

In other words, when $16.p$ is executed (which moves $\mathsf{Loc}[p]$ to the subtree rooted at i), at least one of $\mathsf{Loc}[q]$ and $\mathsf{Loc}[q+N]$ is equal to h, or a splitter within the left subtree of h. Furthermore, because $16.p$ does not alter $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q+N]$, this is also true after $16.p$ is executed, *i.e.*, in state u. Note that, by (10) and (11), i is the right child of h. This implies that $C(h) > C(i)$ holds at state u.

Statement $19.p$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $X[h] \neq p$ holds. However, this implies that $A(p,h)$ is false, and hence $\mathsf{Loc}[p] = 2h$, by (I51). Therefore, statement $19.p$ cannot change the value of $\mathsf{Loc}[p]$.

Statement $20.p$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $Acquired[h] = true$ holds. By (I4), this implies that there exists a process q such that $q@\{3..14, 25..41\} \wedge A(q,h)$ holds. Thus, by (I24) and (I53), $\mathsf{Loc}[q+N] = h$ holds both before and after the execution of $20.p$. This implies that $C(h) > C(i)$.

Statement $15.q$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{16..19\} \wedge p.node = h \wedge q.node = h$ holds. In this case, by (I49), $\mathsf{Loc}[q] = h$ holds at t. Because statement $15.q$ does not update $\mathsf{Loc}[q]$, this condition also holds at u, which implies that $C(h) > C(i)$.

Similarly, statements $27.q$ and $29.q$ may establish $\mathsf{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{16..22\} \wedge p.node = h \wedge q.n = h$ holds. In this case, by (I56), $\mathsf{Loc}[q] = h$ holds at t, and hence at u. This implies that $C(h) > C(i)$. □

invariant $p@\{2,3,15..24\} \wedge (i \xrightarrow{*} \mathsf{Loc}[p]) \Rightarrow \mathsf{PC}[p,i] \geq C(i)$ (I58)

Proof: The only statement that can establish $p@\{2,3,15..24\}$ is $1.p$. But this establishes $\mathsf{Loc}[p] = 1$ and $\mathsf{PC}[p,1] = C(1)$. Hence the consequent is established (or preserved) for $i = 1$, and the antecedent is falsified for $i \neq 1$.

If $p@\{2,3,15..24\}$ holds, then the only statements that can establish $i \xrightarrow{*} \mathsf{Loc}[p]$ are $16.p$, $19.p$, $20.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. By Lemma A2, if one of these statements establishes $i \xrightarrow{*} \mathsf{Loc}[p]$, it establishes it by calling $\mathsf{UpdateLoc}(P,i)$ such that $p \in P$. In this case, line u2 of UpdateLoc establishes $\mathsf{PC}[p,i] = C(i)$.

The value of $\mathsf{PC}[p,i]$ may be changed only by statement $1.p$, or by a call to UpdateLoc. However, statement $1.p$ establishes the consequent as shown above, and whenever UpdateLoc updates $\mathsf{PC}[p,i]$, it establishes $\mathsf{PC}[p,i] = C(i)$.

The value of $C(i)$ may be changed only by statements $14.q$, $25.q$, and $41.q$ (by updating $\mathsf{Loc}[q]$ or $\mathsf{Loc}[q+N]$ directly), or by a call to UpdateLoc. However, by (I55), statement $25.q$ always changes $\mathsf{Loc}[q]$ from a splitter to its ancestor. It follows that statement $25.q$ cannot cause $C(i)$ to increase for any i. Similarly, statements $14.q$ and $41.q$ establishes $\mathsf{Loc}[q] = 0 \wedge \mathsf{Loc}[q+N] = 0$, and hence they also cannot cause $C(i)$ to increase for any i.

The only remaining case is when $C(i)$ is changed by a call to UpdateLoc. However, line u3 of UpdateLoc ensures that (I58) is always preserved in this case. □

invariant $p@\{2, 3, 15..24\} \ \wedge \ (i \xrightarrow{*} \mathsf{Loc}[p]) \ \wedge \ (2 \leq i \leq T) \ \Rightarrow \ \mathsf{PC}[p, i] < \mathsf{PC}[p, \lfloor i/2 \rfloor]$ (I59)

Proof: The only statement that can establish $p@\{2, 3, 15..24\}$ is $1.p$, but this establishes $\mathsf{Loc}[p] = 1$, and hence falsifies the antecedent.

The only other statements that may establish the antecedent (by changing the value of $\mathsf{Loc}[p]$) are $14.p$, $16.p$, $19.p$, $20.p$, $25.p$, $41.p$, $15.q$, $27.q$, and $29.q$, where q is any arbitrary process. Statements $14.p$, $25.p$, and $41.p$, falsify the antecedent. The other statements may establish the antecedent only by calling $\mathsf{UpdateLoc}$ when $p@\{2, 3, 15..24\}$ holds. Similarly, the only statements that may falsify the consequent (by changing the value of $\mathsf{PC}[p, i]$ or $\mathsf{PC}[p, \lfloor i/2 \rfloor]$) are $1.q$, $15.q$, $16.q$, $19.q$, $20.q$, $27.q$, and $29.q$, where q is any arbitrary process. These statements also may falsify (I59) only by calling $\mathsf{UpdateLoc}$ when $p@\{2, 3, 15..24\}$ holds. Therefore, it suffices to consider such an invocation of $\mathsf{UpdateLoc}$. We consider two cases. Let h be the parent of i, *i.e.*, $h = \lfloor i/2 \rfloor$.

Case 1: $i \xrightarrow{*} \mathsf{Loc}[p]$ is established by $\mathsf{UpdateLoc}(P, f)$.

By Lemma A2, $i \xrightarrow{*} \mathsf{Loc}[p]$ can be established only if $p \in P \ \wedge \ f = i$, and in this case $i = \mathsf{Loc}[p] \ \wedge \ C(\lfloor i/2 \rfloor) > C(i)$ is also established. Note that line $\mathsf{u2}$ of $\mathsf{UpdateLoc}$ establishes $\mathsf{PC}[p, i] = C(i)$. Also, because $p@\{2, 3, 15..24\}$ holds, line $\mathsf{u3}$ ensures that $\mathsf{PC}[p, h] \geq C(h)$ holds after the call to $\mathsf{UpdateLoc}$. Thus, $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ holds after $\mathsf{UpdateLoc}$ is called.

Case 2: $i \xrightarrow{*} \mathsf{Loc}[p]$ holds before the call to $\mathsf{UpdateLoc}(P, f)$.

In this case, the antecedent of (I59) holds before the call to $\mathsf{UpdateLoc}$, and hence $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ holds as well. If the value of $\mathsf{PC}[p, h]$ is changed by line $\mathsf{u2}$ of $\mathsf{UpdateLoc}$, then $p \in P$ and $f = h$. But this implies that line $\mathsf{u1}$ establishes $\mathsf{Loc}[p] = h$, which falsifies the antecedent of (I59). (In fact, such a case can never occur, because it implies that a process moves *upward* within the renaming tree while in its entry section.) If the value of $\mathsf{PC}[p, h]$ is changed by line $\mathsf{u3}$, then because line $\mathsf{u3}$ never causes a PC entry to decrease, the condition $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$ cannot be falsified.

The remaining possibility to consider is that the value of $\mathsf{PC}[p, i]$ is changed when $\mathsf{UpdateLoc}(P, f)$ is called. Note that, if $\mathsf{PC}[p, i]$ is changed, either by line $\mathsf{u2}$ or line $\mathsf{u3}$, then

$$\mathsf{PC}[p, i] = C(i) \tag{12}$$

is established. From (I58), it follows that the value of $\mathsf{PC}[p, i]$ can increase only if $C(i)$ also increases. By the definition of $C(i)$, this can happen only if $\mathsf{UpdateLoc}(P, f)$ establishes either $i \xrightarrow{*} \mathsf{Loc}[r]$ or $i \xrightarrow{*} \mathsf{Loc}[r + N]$ for some $r \in P$. In this case, either $\mathsf{UpdateLoc}(P, f)$ is called by statement $1.r$, or $r@\{2, 3, 15..24\}$ holds before the execution of the statement calling $\mathsf{UpdateLoc}(P, f)$ (this can be seen by examining each call to $\mathsf{UpdateLoc}$ in Figure 11). However, statement $1.r$ calls $\mathsf{UpdateLoc}(\{r\}, 1)$, and hence cannot establish $i \xrightarrow{*} \mathsf{Loc}[r]$ for any $i \geq 2$. Therefore, $r@\{2, 3, 15..24\}$ holds before $\mathsf{UpdateLoc}$ is called. By (I49)–(I52), this implies that $\mathsf{Loc}[r] = \mathsf{Loc}[r + N]$ also holds. Therefore, it is enough to consider the case in which $i \xrightarrow{*} \mathsf{Loc}[r]$ is established.

By Lemma A2, $\mathsf{UpdateLoc}(P, f)$ establishes $i \xrightarrow{*} \mathsf{Loc}[r]$ only if it also establishes $C(\lfloor i/2 \rfloor) > C(i)$, *i.e.*, $C(h) > C(i)$. As noted earlier, line $\mathsf{u3}$ ensures that $\mathsf{PC}[p, h] \geq C(h)$ holds after the call to $\mathsf{UpdateLoc}$. Thus, by (12), we again have $\mathsf{PC}[p, h] > \mathsf{PC}[p, i]$. □

invariant (Contention Sensitivity) $p@\{4..8\} \ \Rightarrow \ \mathsf{lev}(p.node) < \mathsf{PC}[p, 1]$ (I61)

Proof: The antecedent is established only by statement $3.p$. We begin by showing that if $3.p$ establishes the antecedent by establishing $p@\{4, 6\}$, then $\mathsf{PC}[p, 1] \geq \mathsf{PC}[p, p.node] + p.level$ holds after its execution. Note that $3.p$ establishes $p@\{4, 6\}$ only if executed when $p.dir = stop \ \vee \ p.level \geq L$ holds. If $3.p$ is executed when $p.level = 0 \ \wedge \ p.dir = stop$ holds, then by (I35), $p.node = 1$ holds as well. $3.p$ does not update either $p.node$ or $p.level$ in this case, and hence, $\mathsf{PC}[p, 1] \geq \mathsf{PC}[p, p.node] + p.level$ holds after its execution.

The remaining possibility to consider is that $(p.dir = stop \ \wedge \ p.level > 0) \ \vee \ p.level \geq L$ holds before $3.p$ is executed. In this case, by (I46), $3.p$ establishes the following.

$$p.path[0].node = 1 \ \wedge \ p.path[p.level-1].node \xrightarrow{*} p.node \ \wedge$$
$$(\forall l : 0 \le l < p.level - 1 :: p.path[l].node \xrightarrow{*} p.path[l+1].node).$$

By (I59), the following is established as well.

$$\mathsf{PC}[p,1] = \mathsf{PC}[p, p.path[0].node] > \mathsf{PC}[p, p.path[1].node] > \cdots > \mathsf{PC}[p, p.path[p.level-1].node] > \mathsf{PC}[p, p.node]$$

Given the length of this sequence, we have $\mathsf{PC}[p,1] \ge \mathsf{PC}[p, p.node] + p.level$, as claimed.

By (I35), this implies that $\mathsf{PC}[p,1] \ge \mathsf{PC}[p, p.node] + \mathsf{lev}(p.node)$ holds after $3.p$ is executed. Note that $p@\{4,6\}$ implies that $\mathsf{PC}[p, p.node] \ge 1$ holds (the point contention for some process at a splitter must at least include that process). Hence, if $3.p$ establishes the antecedent of (I61), then $\mathsf{PC}[p,1] > \mathsf{lev}(p.node)$ holds after its execution.

While the antecedent holds, the value of $\mathsf{lev}(p.node)$ cannot be changed. Moreover, if $\mathsf{PC}[p,1]$ is changed within UpdateLoc, then its value is increased. □

Note that $\mathsf{PC}[p,1]$ is initialized by p at statement $1.p$ to match the current contention within the renaming tree, assuming that each process q is counted twice, as "q" and as "$q+N$." While $p@\{2,3,15..24\}$ continues to hold, as other processes enter and leave the renaming tree, if the current contention ever exceeds the current the value of $\mathsf{PC}[p,1]$, then line u3 of UpdateLoc ensures that $\mathsf{PC}[p,1]$ is updated accordingly. Thus, if m is the maximum value attained by $\mathsf{PC}[p,1]$ while $p@\{2,3,15..24\}$ holds, then m is at most twice the actual point contention experienced by p in its entry section.

It should be clear that the number of remote memory references executed by p to enter and then exit its critical section is $\Theta(n)$, where n is the value of $\mathsf{lev}(p.node)$ when p reaches statement 4 or 6. By (I61), we have $n < m$. Clearly, we also have $n \le L+1$. Thus, p executes $O(min(k, \log N))$ remote memory references to enter and then exit its critical section, where k is the point contention it experiences in its entry section.