

Adaptive Mutual Exclusion with Local Spinning*

James H. Anderson and Yong-Jik Kim
Department of Computer Science
University of North Carolina at Chapel Hill

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. All previous so-called adaptive mutual exclusion algorithms employ busy-waiting loops that can generate an unbounded number of remote memory references. Thus, they have unbounded time complexity under this measure.

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

*Work supported by NSF grants CCR 9732916 and CCR 9972211.

1 Introduction

The mutual exclusion problem has been studied for many years, dating back to the seminal paper of Dijkstra [5]. In this problem, each of a set of processes repeatedly cycles through four sections of code, called *noncritical*, *entry*, *critical*, and *exit* sections, respectively. A process may halt within its noncritical section but not within its critical section. The objective is to design the entry and exit sections to ensure that at most one process executes its critical section at any time, and that each process in its entry section eventually enters its critical section.

In this paper, we consider adaptive solutions to the mutual exclusion problem under read/write atomicity. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes. Two notions of contention have been considered in the literature: “interval contention” and “point contention” [3]. These two notions are defined with respect to a history H . 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 consider only point contention. Throughout the paper, we let N denote the number of processes in the system, and let k denote point contention. We mostly consider per-process time complexities, so when referring to point contention, we mean point contention over a history that starts when some particular process becomes active and ends when that process once again becomes inactive.

In previous work on adaptive mutual exclusion algorithms, two time complexity measures have been considered: “remote step complexity” and “system response time.” The *remote step complexity* of an algorithm is 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 [10]. The *system response time* is the length of time between critical section entries, assuming each enabled read or write operation is executed within some constant time bound [4]. Several read/write mutual exclusion algorithms have been presented that are adaptive to some degree under these time complexity measures. One of the first such algorithms was an algorithm of Styer that has $O(\min(N, k \log N))$ remote step complexity and $O(k \cdot \min(N, k \log N))$ response time [10]. Choy and Singh later improved upon Styer’s result by presenting an algorithm with $O(N)$ remote step complexity and $O(k)$ response time [4]. More recently, Attiya and Bortnikov presented an algorithm with $O(k)$ remote step complexity and $O(\log k)$ response time [3].

Unfortunately, the time complexity measures used in previous work on adaptive algorithms have not proven to be a good predictor of real performance, and thus they may be of questionable practical utility. Recent work on scalable local-spin mutual exclusion algorithms has shown that *the* most crucial factor in determining an algorithm’s performance is the amount of interconnect traffic it generates [2, 6, 8, 11]. In light of this, we define the *time complexity* of a mutual exclusion algorithm to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section. A *remote memory reference* is a shared variable access that requires an interconnect traversal.

In local-spin algorithms, all busy-waiting loops are required to be read-only loops in which only locally-accessible shared variables are accessed that do not require an interconnect traversal. On a distributed shared-memory multiprocessor, a shared variable is locally accessible if it is stored in a local memory module. On a cache-coherent multiprocessor, a shared variable is locally accessible if it is stored in a local cache line.

The first local-spin algorithms were algorithms in which read-modify-write instructions are used to enqueue blocked processes onto the end of a “spin queue” [2, 6, 8]. Each of these algorithms has $O(1)$ time complexity; thus, adaptivity is clearly a non-issue if appropriate read-modify-write instructions are available. Yang and Anderson were the first to consider local-spin algorithms under read/write atomicity [11]. They presented a read/write mutual exclusion algorithm with $\Theta(\log N)$ time complexity in which instances of a local-spin mutual exclusion algorithm for two processes are embedded within a binary arbitration tree. Yang and Anderson also presented a variant of this algorithm that includes a “fast path” that allows the tree to be bypassed in the absence of contention. This algorithm has the desirable property that contention-free time complexity is $O(1)$. Unfortunately, it has the undesirable property that time complexity under contention is $\Theta(N)$ in the worst case, rather than $\Theta(\log N)$. 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 with $O(1)$ time complexity in the absence of contention and $\Theta(\log N)$ time complexity under contention, when used in conjunction with Yang and Anderson’s algorithm [1].

All of the previously-cited adaptive algorithms are not local-spin algorithms, and thus they have unbounded time complexity under the remote-memory-references time measure. As such, one might question whether these algorithms are truly adaptive. Indeed, Styer’s algorithm was shown to perform poorly under contention in a performance study conducted by Yang and Anderson to compare local-spin algorithms with non-local-spin algorithms [11]. It is our belief that for an algorithm to be considered truly adaptive, it must be adaptive under the remote-memory-references time complexity measure. After all, the underlying hardware does *not* distinguish between remote memory references generated by **await** statements and remote memory references generated by other statements. 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. In this paper, we close this problem by presenting an algorithm that has $O(k)$ time complexity under the remote-memory-references measure. The space complexity of our algorithm is $\Theta(N)$, which is clearly optimal. Thus, our algorithm is both space- and time-efficient.

Our algorithm can be seen as an extension of the fast-path algorithm of Anderson and Kim [1]. This algorithm was devised by thinking about connections between fast-path mechanisms and long-lived renaming [9]. 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. Much of the code that must be executed to acquire and release the fast-path name can be executed within a process’s critical section. This makes the problem of acquiring and releasing a name much easier than is the case when designing a *wait-free* renaming algorithm. 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.

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 Section 2.2, 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 [9]. This splitter element was actually first used in Lamport’s fast mutual exclusion algorithm [7]. The splitter element is defined by the code fragment shown in Figure 1. (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.) Each process that invokes this code fragment 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 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.

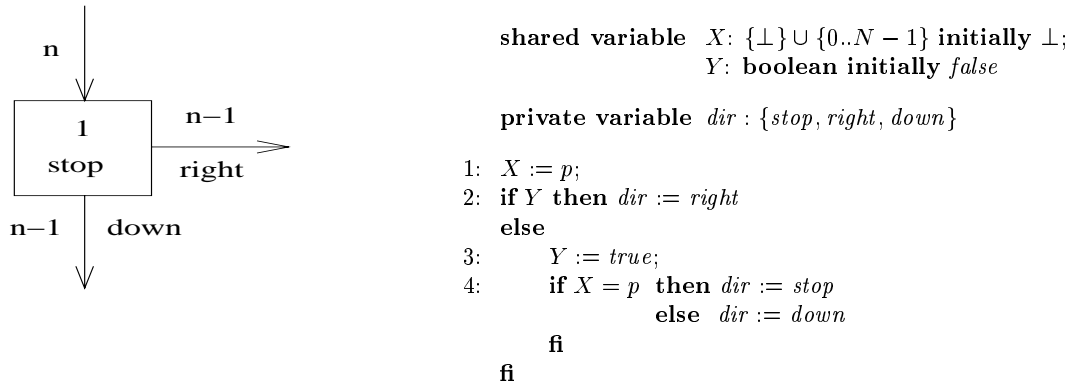


Figure 1: The splitter element and the code fragment that implements it.

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 2(a). 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. When a process stops at a splitter, it acquires the name associated with that splitter. In the *long-lived* renaming problem [9], processes must have the ability to release the names they acquire. In the grid algorithm, a process can release its name by resetting each splitter on the path traversed by it in acquiring its name. 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 whether 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, developing a more efficient reset mechanism was a major issue to be faced in the design of 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 2(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 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 [11]. We explain below why a three-process algorithm is needed instead of a two-process algorithm (as one would expect to have in an arbitration tree).

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 a node is reached where *stop* is returned. 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** Starting with its acquired node, p then competes within the arbitration tree. A process competes within the arbitration tree by executing each of the three-process entry sections on the path from its acquired node to the root. Note that a node’s entry section may be invoked by the process that obtained *stop* at that node, and one process from each of the left and right subtrees beneath that node.

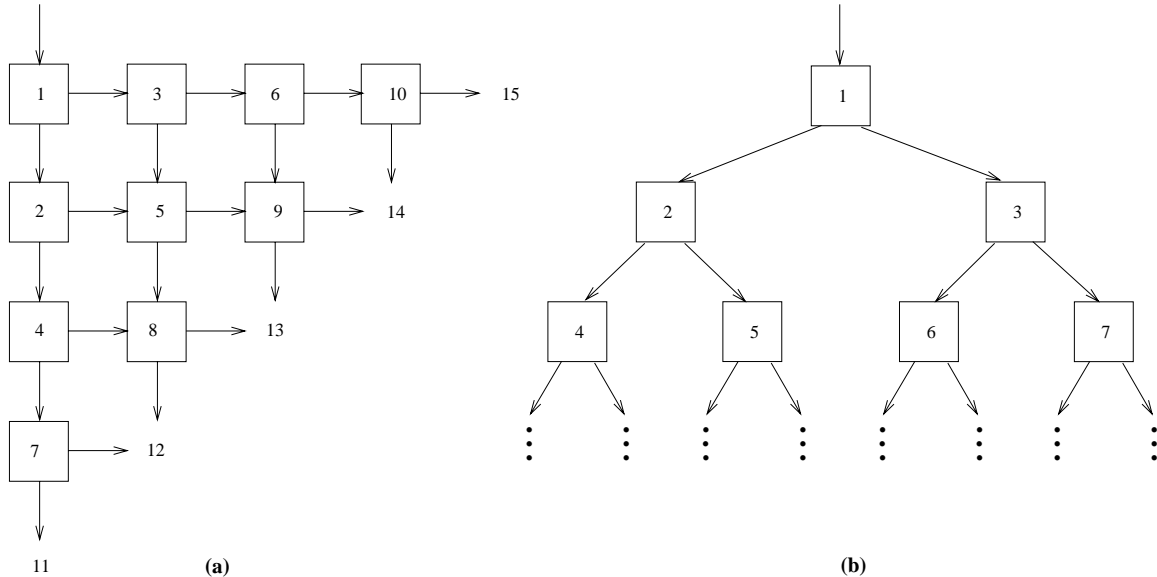


Figure 2: (a) Renaming grid (depicted for $N = 5$). (b) Renaming tree.

This is why a three-process algorithm is needed instead of a two-process algorithm.

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 [11]. 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 3(a). Figure 3(b) illustrates the steps that might be taken by a process p in acquiring a new name if contention is $O(\log N)$. Figure 3(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 the grid points, 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 1. If a process does succeed 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 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, in the context of a mutual exclusion algorithm, much of the reset code can be executed within a process’s critical section [1]. 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

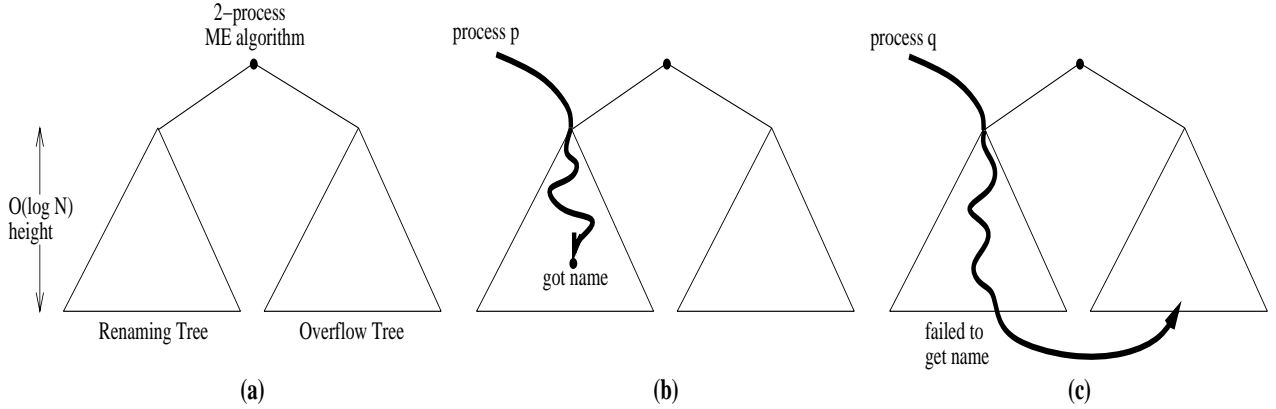


Figure 3: (a) Renaming tree and overflow tree. (b) Process p gets a name in the renaming tree. (c) Process q fails to get a name and must compete within the overflow tree.

acquire the fast-path name. In our adaptive algorithm, every name is associated with some “path” to the critical section, so we must efficiently manage acquisitions and releases for a set of names.

2.2 Detailed Description

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. First, we consider a version of the algorithm in which unbounded memory is used to reset splitters in constant time. Second, we consider a variant of the algorithm with $\Theta(N^2)$ space complexity in which all variables are bounded. Third, we present another variant that has $\Theta(N)$ space complexity. In explaining these algorithms, we actually present proof sketches for some of the key properties of each algorithm. Our intent is to use these proof sketches as a means for explaining the basic mechanisms of each algorithm in a way that is as intuitive as possible. A formal correctness proof for the final algorithm is presented in an appendix.

Algorithm U. The first algorithm, which we call Algorithm U (for unbounded), is shown in Figure 4. Before describing how this algorithm works, we first examine its basic structure. At the top of Figure 4, definitions of two constants are given: L , which gives the number of levels 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 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 1, 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, we allow these round numbers to 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 identify a potential “winning” process that has succeeded in acquiring name i in round r . $Acquired[i]$ is set to record that name i has been acquired by some process.

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 executing statements 2-10 with $node = i$. Of these, statements 2-5 correspond to the splitter code in Figure 1. Statements 6-9 are executed as part of a handshaking mechanism that prevents a process that is releasing a

```

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.. $\infty$ ;
  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, n: 1..2T + 1;
  level, j: 0..L + 1;
  y: Ytype;
  dir: Dtype;
  path: array[0..L] of Ptype

process p :: /* 0 ≤ p < N */
while true do
0: Noncritical Section;
1: node, level := 1, 0;

   /* descend renaming tree */
repeat
2:   X[node], dir := p, stop;
3:   y := Y[node];
   if ¬y.free then dir := right
   else
4:   Y[node] := (false, 0);
5:   if X[node] ≠ p ∨
6:     Acquired[node] then
       dir := left
   else
7:   Round[node][y.rnd] := true;
8:   if Reset[node] ≠ y then
9:     Round[node][y.rnd], dir := false, left
   fi
   fi
10: path[level] := (node, dir);
   if dir ≠ stop then
     level, node := level + 1, 2 · node;
     if dir = right then node := node + 1 fi
   fi
until (level > L) ∨ (dir = stop);

   if level ≤ L then /* got a name */
     /* compete in renaming tree, then 2-proc. alg. */
11:   Acquired[node] := true;
     for j := level downto 0 do
12:     ENTRY3(path[j].node, path[j].dir)
     od;
13:   ENTRY2(0)
     else /* didn't get a name */
       /* compete in overflow tree, then 2-proc. alg. */
14:   ENTRYN(p);
15:   ENTRY2(1)
   fi;

16: Critical Section;

   /* reset splitters */
   for j := min(level, L) downto 0 do
     if path[j].dir ≠ right then
17:       n := path[j].node;
18:       y := Reset[n];
19:       Reset[n] := (false, y.rnd);
20:       if j = level ∨ ¬Round[n][y.rnd] then
21:         Reset[n] := (true, y.rnd + 1);
22:         Y[n] := (true, y.rnd + 1)
       fi
     fi
   od;

   /* execute appropriate exit sections */
   if level ≤ L then
23:     EXIT2(0);
     for j := 0 to level do
24:       EXIT3(path[j].node, path[j].dir)
     od;
25:     Acquired[node] := false
   else
26:     EXIT2(1);
27:     EXITN(p)
   fi
od

```

Figure 4: Algorithm U: Adaptive algorithm with unbounded memory.

name from adversely interfering with processes attempting to acquire that name; this mechanism is discussed in detail below. Statement 10 simply prepares for the next iteration of the **repeat** loop (if there is one).

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 (statements 11-12). Each of these three-process entry sections is denoted “ENTRY₃(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 3(a)) using “0” as a process identifier (statement 13). This entry section is denoted “ENTRY₂(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 14), which is implemented using Yang and Anderson’s N -process arbitration-tree algorithm. The entry section of this algorithm is denoted ENTRY _{N} (p). Note that p uses its own process identifier in this algorithm. After competing within the overflow tree, p competes within the two-process algorithm “on top” of both tree using “1” as a process identifier (statement 15). This entry section is denoted “ENTRY₂(1).”

After completing the appropriate two-process entry section, process p executes its critical section (statement 16). It then resets each of the splitters that it visited while descending the renaming tree (statements 17-22). This reset mechanism is discussed in detail below. Process p then executes the exit sections corresponding to the entry sections it executed previously (statements 23-27). The exit sections are specified in a manner that is similar to the entry sections.

We now consider in detail the code fragments that are executed to acquire (statements 2-10) or reset (statements 18-22) some splitter i . To facilitate this discussion, we will index these statements by i . For example, when we refer to the execution of statement 4[i] by process p , we mean the execution of statement 4 by p when its local variable *node* equals i . Similarly, the execution of statement 18[i] by p refers to the execution of statement 18 by p when its local variable n equals i .

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.

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 r at node i . Of the processes that participate in round r at node i , at least one will read $Y[i] = (true, r)$ at statement 3[i] and assign $Y[i] := (false, 0)$ at statement 4[i]. By the correctness of the original splitter code, of the processes that assign $Y[i]$, at most one will reach statement 7[i]. A process that reaches statement 7[i] will either stop at node i or be deflected left.

This gives us two cases to analyze: of the processes that read $Y[i] = (true, r)$ at statement 3[i] and assign $Y[i]$ at statement 4[i], either all are deflected left, or one, say p , stops at splitter i . In the former case, at least one of the left-moving processes finds $Round[i][r]$ to be false at statement 20[i], and then reopens splitter i by executing statements 21[i] and 22[i], which establish $Y[i] = (true, r + 1) \wedge Y[i] = Reset[i]$. To see why at least one process executes statements 21[i] and 22[i], note that each process under consideration reads $Y[i] = (true, r)$ at statement 3[i], and thus its *y.rnd* variable equals r while executing within statements 4[i]-9[i]. Note also that $Round[i][r] = true$ is established only by statement 7[i]. Furthermore, each process that is deflected left at statement 9[i] first assigns $Round[i][r] := false$. Thus, at least one of the left-moving processes finds $Round[i][r]$ to be false at statement 20[i].

In the case that there is a winning process p at splitter i during round r , we must argue that **(i)** p reopens

splitter i upon leaving it, and **(ii)** no left- or right-moving process “prematurely” reopens splitter i before p has left it. Establishing (i) is straightforward. Process p will reopen the splitter by executing statements 18[i]-22[i] and 25, which establish $Y[i] = (\text{true}, r + 1) \wedge \text{Acquired}[i] = \text{false} \wedge Y[i] = \text{Reset}[i]$. Note that the assignment to *Acquired* at statement 25 prevents the reopening of splitter i from actually taking effect until after p has finished executing its exit section.

To establish (ii), suppose, to the contrary, that some left- or right-moving process reopens splitter i by executing statement 22[i] while p is executing within statements 10[i]-13 and 16-25. (Note that, because p stops at splitter i , it doesn’t iterate again within the **repeat** loop.) Let q be the first left- or right-moving process to execute statement 22[i]. Since we are assuming that the **ENTRY** and **EXIT** calls are correct, q cannot execute statement 22[i] while p is executing within statements 16-22. Moreover, if p is executing within statements 12-13 or 23-25, then *Acquired* is true, and hence the splitter is closed. The remaining possibility is that p is enabled to execute statement 10[i] or 11. (Note that, in this case, if q were to reopen splitter i then we could end up with two processes invoking $\text{ENTRY}_3(i, \text{stop})$ at statement 12, i.e., both processes use *stop* as a “process identifier.” The **ENTRY** calls obviously cannot be assumed to work correctly if they can be invoked concurrently by different processes with the same process identifier.)

So, assume that q executes statement 22[i] while p is enabled to execute statement 10[i] or 11. For this to happen, q must have read $\text{Round}[i][r] = \text{false}$ at statement 20[i] before p assigned $\text{Round}[i][r] := \text{true}$ at statement 7[i]. (Recall that all the processes under consideration read $Y[i] = (\text{true}, r)$ at statement 2[i]. This is why p writes to $\text{Round}[i][r]$ instead of some other element of $\text{Round}[i]$. q reads from $\text{Round}[i][r]$ at statement 20[i] because it is the first process to attempt to reset splitter i , which implies that q reads $\text{Reset}[i] = (\text{true}, r)$ at statement 18[i].) Because q executes statement 20[i] before p executes statement 7[i], statement 19[i] is executed by q before statement 8[i] is executed by p . Thus, p must have found $\text{Reset}[i] \neq y$ at statement 7[i], i.e., it was deflected left at splitter i , which is a contradiction. It follows from the explanation given here that splitter i is eventually reset for round $r + 1$, i.e., $Y[i] = (\text{true}, r + 1) \wedge Y[i] = \text{Reset}[i] \wedge \text{Acquired}[i] = \text{false}$ is eventually established.

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. Thus, we have the following.

Lemma U1: Algorithm U is a correct, starvation-free mutual exclusion algorithm. □

Having dispensed with basic correctness, we now informally argue that Algorithm U is contention sensitive. 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 [9], 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 [11], the ENTRY_2 , EXIT_2 , ENTRY_3 , and EXIT_3 calls take constant time, and the ENTRY_N and EXIT_N calls take $\Theta(\log N)$ time. Note that the ENTRY_N and EXIT_N routines are called by a process only if its point contention is $\Omega(\log N)$. Thus, we have the following.

Lemma U2: Algorithm U has $O(\min(k, \log N))$ time complexity and unbounded space complexity. □

Of course, the problem with Algorithm U is that the *rnd* field of $Y[i]$ that is used for assigning round numbers grows without bound. We now consider a variant of Algorithm U in which space is bounded.

Algorithm B. In Algorithm B (for bounded space), $Y[i].\text{rnd}$ is incremented modulo- N , and hence does not grow without bound. Algorithm B is shown in Figure 5. With $Y[i].\text{rnd}$ being incremented modulo- N , the following potential problem arises. A process p may reach statement 8[i] in Figure 5 with $y.\text{rnd} = r$ and then get delayed. While delayed, other processes may repeatedly increment $Y[i].\text{rnd}$ (statement 27[i]) until it “cycles back” to r . At this point, another process q may reach statement 8[i] with $y.\text{rnd} = r$. This is a problem because p and q may interfere with each other in updating $\text{Round}[i][r]$.

/* all variable declarations are as defined in Figure 4 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;
  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 */
while true do
0: Noncritical Section;
1: node, level := 1, 0;

   /* descend renaming tree */
repeat
2:   X[node], dir := p, stop;
3:   y := Y[node];
   if ¬y.free then dir := right
   else
4:   Y[node] := (false, 0);
5:   Obstacle[p] := node;
6:   if X[node] ≠ p ∨
7:     Acquired[node] then
       dir := left
   else
8:   Round[node][y.rnd] := true;
9:   if Reset[node] ≠ y then
10:    Round[node][y.rnd], dir := false, left
   fi
   fi
11: path[level] := (node, dir);
   if dir ≠ stop then
     level, node := level + 1, 2 · node;
     if dir = right then node := node + 1 fi
   fi
until (level > L) ∨ (dir = stop);

   if level ≤ L then /* got a name */
     /* compete in renaming tree, then 2-proc. alg. */
12:   Acquired[node] := true;
     for j := level downto 0 do
13:     ENTRY3(path[j].node, path[j].dir)
     od;
14:   ENTRY2(0)
     else /* didn't get a name */
       /* compete in overflow tree, then 2-proc. alg. */
15:   ENTRYN(p);
16:   ENTRY2(1)
   fi;

17: Critical Section;
18: Obstacle[p] := 0;

   /* reset splitters */
   for j := min(level, L) downto 0 do
     if path[j].dir ≠ right then
19:       n := path[j].node;
20:       Y[n] := (false, 0);
21:       X[n] := p;
22:       y := Reset[n];
23:       Reset[n] := (false, y.rnd);
24:       if (j = level ∨ ¬Round[n][y.rnd]) ∧
25:         Obstacle[y.rnd] ≠ n then
26:         Reset[n] := (true, y.rnd + 1 mod N);
27:         Y[n] := (true, y.rnd + 1 mod N)
       fi;
28:       if j = level then Round[y.rnd] := false fi
     fi
   od;

   /* execute appropriate exit sections */
   if level ≤ L then
29:     EXIT2(0);
     for j := 0 to level do
30:       EXIT3(path[j].node, path[j].dir)
     od;
31:     Acquired[node] := false
   else
32:     EXIT2(1);
33:     EXITN(p)
   fi
od

```

Figure 5: Algorithm B: adaptive algorithm with $\Theta(N^2)$ space complexity.

Algorithm B prevents such a scenario from happening by preventing $Y[i].rnd$ from cycling while a process p that stops a splitter i executes within statements 8[i]-31. Informally, cycling is 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 8[i], process p must first assign $Obstacle[p] := i$ at statement 5[i]. Note further that before a process can increment $Y[i].rnd$ from r to $r + 1 \bmod N$ (statement 27[i]), it must first read $Obstacle[r]$ (statement 25[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 statements 8[i]-31. Note that process p resets $Obstacle[p]$ to 0 at statement 18. This is done to ensure that p ’s own obstacle doesn’t prevent it from incrementing a splitter’s round number.

To this point, we have explained every difference between Algorithms U and B except one: in Figure 5, there are added assignments to elements of Y and X (statements 20 and 21) after the critical section. The reason for these assignments is as follows. Suppose some process p is about to assign $Obstacle[p] := true$ at statement 5[i], but gets delayed. (In other words, p is “about to” erect an obstacle to prevent $Y[i].rnd$ from cycling.) We must ensure that if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p . Let r be the value read from $Y.rnd$ by p at statement 3[i]. For $Y.rnd$ to be incremented beyond p , some other process q that reads $Y.rnd = r$ must attempt to reopen splitter i .

So, suppose that process q reopens splitter i by executing statement 27[i] while p is delayed at statement 5[i]. If process q executes statement 21[i] after p executes statement 2[i], then p will find $X[i] \neq p$ at statement 6[i] and will be deflected left. So, assume that q executes statement 21[i] before p executes statement 2[i]. This implies that q establishes $Y[i].free = false$ by executing statement 20[i] before p reads $Y[i]$ at statement 3[i]. Note that $Y[i].free = true$ is only established within a critical section (statement 27[i]). Also, note that we have established the following sequence of statement executions (perhaps interleaved with statement executions of other processes): q executes statements 20[i] and 21[i]; p executes statements 2[i]-5[i]; q executes statement 27[i] (q ’s execution of statements 22[i]-26[i] may interleave arbitrarily with p ’s execution of statements 2[i]-5[i]). Because statements 17[i]-27[i] are executed as a critical section, this implies that p reads $Y[i].free = false$ at statement 3[i], and thus does not reach statement 5[i], which is a contradiction. We conclude from this reasoning that if p is delayed at statement 5[i], and if p ultimately reaches statement 8[i], then $Y[i].rnd$ does not get incremented beyond the value p .

From the discussion above, we have the following properties for Algorithm B.

Lemma B1: Algorithm B is a correct, starvation-free mutual exclusion algorithm. □

Lemma B2: Algorithm B has $O(\min(k, \log N))$ time complexity and $\Theta(N^2)$ space complexity. □

By examining the shared variable declarations in Figures 4 and 5, it should be clear that 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.

Algorithm L. The final algorithm we present is depicted in Figure 6. We refer to this algorithm as Algorithm L (for linear space). In Algorithm L, we use a common pool of round numbers that range over $\{1, \dots, S\}$ for all of the 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*.

/* all variable declarations are as defined in Figure 5 except as noted here */

const

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

type

$Ytype = \text{record } free : \text{boolean}; rnd : 0..S \text{ end}$ /* stored in one word */

shared variables

$Y, Reset : \text{array}[1..T] \text{ of } Ytype;$
 $Round : \text{array}[1..S] \text{ of boolean initially false};$
 $Free : \text{queue of integers};$
 $Inuse \text{ array}[0..N - 1] \text{ of } 0..S \text{ initially } 0;$
 $Check : 0..N - 1 \text{ initially } 0$

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

while $true$ **do**

0: Noncritical Section;

1: $node, level := 1, 0;$

/* descend renaming tree */

repeat

2: $X[node], dir := p, stop;$

3: $y := Y[node];$

if $\neg y.free$ **then** $dir := right$

else

4: $Y[node] := (false, 0);$

5: $Inuse[p] := y.rnd;$

6: **if** $X[node] \neq p \vee$

7: $Acquired[node]$ **then**

$dir := left$

else

8: $Round[y.rnd] := true;$

9: **if** $Reset[node] \neq y$ **then**

10: $Round[y.rnd], dir := false, left$

fi

fi

11: $path[level] := (node, dir);$

if $dir \neq stop$ **then**

$level, node := level + 1, 2 \cdot node;$

if $dir = right$ **then** $node := node + 1$ **fi**

fi

until $(level > L) \vee (dir = stop);$

if $level \leq L$ **then** /* got a name */

/* compete in renaming tree, then 2-proc. alg. */

12: $Acquired[node] := true;$

for $j := level$ **downto** 0 **do**

13: $ENTRY_3(path[j].node, path[j].dir)$

od;

14: $ENTRY_2(0)$

else /* didn't get a name */

/* compete in overflow tree, then 2-proc. alg. */

15: $ENTRY_N(p);$

16: $ENTRY_2(1)$

fi;

initially

$(\forall i : 1 \leq i \leq T :: Y[i] = (true, i) \wedge$

$Reset[i] = (true, i)) \wedge$

$(Free = (T + 1) \rightarrow \dots \rightarrow S)$

private variables

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

17: Critical Section;

/* reset splitters */

for $j := \min(level, L)$ **downto** 0 **do**

if $path[j].dir \neq right$ **then**

18: $n := path[j].node;$

19: $Y[n] := (false, 0);$

20: $X[n] := p;$

21: $y := Reset[n];$

22: $Reset[n] := (false, y.rnd);$

23: **if** $j = level \vee \neg Round[y.rnd]$ **then**

24: $ptr := Check;$

25: $usedrnd := Inuse[ptr];$

26: **if** $usedrnd \neq 0$ **then**

$MoveToTail(Free, usedrnd)$

fi;

27: $Check := ptr + 1 \bmod N;$

28: $Enqueue(Free, y.rnd);$

29: $nextrnd := Dequeue(Free);$

30: $Reset[n] := (true, nextrnd);$

31: $Y[n] := (true, nextrnd)$

fi;

if $j = level$ **then**

32: $Round[y.rnd] := false;$

33: $Inuse[p] := 0$

fi

od;

/* execute appropriate exit sections */

if $level \leq L$ **then**

34: $EXIT_2(0);$

for $j := 0$ **to** $level$ **do**

35: $EXIT_3(path[j].node, path[j].dir)$

od;

36: $Acquired[node] := false$

else

37: $EXIT_2(1);$

38: $EXIT_N(p)$

fi

od

Figure 6: Algorithm L: adaptive algorithm with $\Theta(N)$ space complexity.

- *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.

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.

There are only a few differences between Algorithms B and L. The only difference before the critical section is statement 5[i]: instead of updating *Obstacle*[p], process p now marks the round number r that it just read from $Y[i]$ as being “in use” by assigning $Inuse[p] := r$. The only other differences are in the code after the critical section and before exit sections (statements 18-33 in Figure 6). Statements 24-27 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. With $S = T + 2N$ round numbers, this is sufficient to prevent r from reaching the head of the queue while p is delayed. Statement 28[i] enqueues the current round number for splitter i onto the *Free* queue. (Note that there may be other processes within the renaming tree that “think” that the just-enqueued round number is the current round number for splitter i ; this is why we need a mechanism to prevent round numbers from prematurely reaching the head of the queue.) Statement 29[i] simply dequeues a new round number from *Free*. The rest of the algorithm is the same as before.

By examining Figures 4 through 6, it should be clear that the space complexity of Algorithm L is $\Theta(N)$, if we ignore the space required to implement the ENTRY and EXIT routines. (Note that each process has a private *path* array with $\Theta(\log N)$ entries. These arrays are not actually needed. The renaming tree’s structure is statically determined, so simple arithmetic calculations can be used to determine the parent of a splitter i , and whether i is a left or right child.) If the ENTRY and EXIT routines are implemented using Yang and Anderson’s local-spin algorithm [11], then the overall space complexity is actually $\Theta(N \log N)$. This is because in Yang and Anderson’s arbitration-tree algorithm, each process needs a distinct spin location for each level of the arbitration tree. However, as we will show in the full paper, 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.

This completes the informal explanation of our algorithm. A formal correctness proof for Algorithm L is given in the appendix. We conclude this section by stating our main theorem.

Theorem 1 *Algorithm L is a correct N -process mutual exclusion algorithm, and is starvation-free, provided the mutual exclusion algorithm used to implement the ENTRY and EXIT routines is starvation-free. Moreover, if these routines are implemented using the modified version of Yang and Anderson’s local-spin algorithm mentioned above, then the space complexity of Algorithm L is $\Theta(N)$, and each critical section access by a process p requires $O(\min(k, \log N))$ remote memory references, where k is the point contention measured over p ’s entry section. \square*

3 Concluding Remarks

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

Acknowledgement: Gary Peterson recently conjectured to us that adaptivity under the remote-memory-references time measure must necessitate $\Omega(N^2)$ space complexity. His conjecture led us to develop Algorithm L.

References

- [1] J. Anderson and Y.-J. Kim. Fast *and* scalable mutual exclusion. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 180–194, September 1999.
- [2] 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.
- [3] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*. July 2000 (to appear).
- [4] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
- [5] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
- [6] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.
- [7] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
- [8] 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.
- [9] M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, October 1995.
- [10] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. August 1992.
- [11] 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 prove that Algorithm L is correct. Our proof makes use of a number of auxiliary variables. In Figure 7, Algorithm L is shown with these added auxiliary variables. We have marked the lines of code that refer to auxiliary variables with a dash “—” to make them easier to distinguish. Before describing the auxiliary variables that have been added, we first state several notational conventions that will be used in our proof.

Notational Conventions: Unless stated otherwise, we assume i and h range over $\{1, \dots, T\}$, and that p and q range over $\{0, \dots, N - 1\}$. We use $a.p$ to denote the statement with label a of process p , and $p.b$ to represent p 's private variable b . 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 a is a statement label, then $p@{a}$ means that process p is *enabled* to execute statement a , i.e., it hasn't executed statement a yet.) As stated earlier, we assume that each labeled sequence of statements in Figure 7 is atomic. (For example, statement $6.p$ establishes $p@{7}$ if $X[p.node] = p$ holds, and $p@{11}$ if $X[p.node] \neq p$ holds. In the latter case, the execution of $6.p$ includes the update to $p.dir$ and the call to `UpdateLoc` listed below statement $7.p$.) \square

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. In its entry section, a process p can only deflect other processes at the splitter it is attempting to acquire. However, in its exit section, p can deflect other processes at *two* splitters — the splitter indicated by the value of $p.node$ and the splitter indicated by the value of $p.n$. In the former case, other processes may be deflected because $Acquired[p.node]$ is true. In the latter case, p may deflect other processes by changing the value of Y , X , or $Reset$ (statements 19-22). When tracking each process's current location, we therefore find it convenient to conceptually view each process p as being *two* processes, p and $p + N$. When p is in its entry section, p and $p + N$ are always located at the same splitter. However, when p is in its exit section, p and $p + N$ may be located at different splitters.

We now define several terms that will be used in the proof.

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

$$\begin{aligned}
 A(p, i) \equiv & p.node = i \wedge p@\{4..9, 11..38\} \wedge \\
 & (p@\{4..6\} \Rightarrow X[i] = p) \wedge \\
 & (p@\{4..9\} \Rightarrow Reset[i] = p.y) \wedge \\
 & (p@\{11\} \Rightarrow p.dir = stop)
 \end{aligned}
 \quad \square$$

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 .) \square

Definition: We define $\text{lev}(i)$ as the level of splitter i in the renaming tree, i.e., $\text{lev}(i) = \lfloor \log i \rfloor$. \square

We are now in a position to describe the auxiliary variables used in our proof. They are as follows.

- $\text{Loc}[p]$ is the location of process p within the renaming tree, where $0 \leq p < 2N$.
- $\text{Dist}[r]$ specifies the distance of round number r from the head of $Free$. If r is not in $Free$, then $\text{Dist}[r] = \perp$. $\text{Dist}[r]$ is assumed to be updated within the procedures *Enqueue*, *Dequeue*, and *MoveToTail*.

One additional definition is needed before we consider the final set of auxiliary variables.

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

$$C(i) \equiv |\{p : 0 \leq p < 2N :: i \xrightarrow{*} \text{Loc}[p]\}| \quad \square$$

The last set of auxiliary variables is as follows.

- $\text{PC}[p, i]$ 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 , $\text{PC}[p, i]$ is initialized to be $C(i)$, the contention of splitter i . From that point onward, $\text{PC}[p, i]$ 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 spitter i . $\text{PC}[p, i]$ is not used if process p is outside its entry section or if $i \xrightarrow{*} \text{Loc}[p]$ is false.

The auxiliary procedure $\text{UpdateLoc}(P, i)$ is called when a set P of processes moves to splitter i while in their entry sections. This procedure updates the value of $\text{Loc}[p]$ and $\text{Loc}[p + N]$ to be i , for each $p \in P$, and sets the value of $\text{PC}[p, i]$ 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  $p \in P$  do  $\text{Loc}[p]$ ,  $\text{Loc}[p + N] := i$ ,  $i$  od;
u2: for all  $p \in P$  do  $\text{PC}[p, i] := C(i)$  od;
u3: for all  $q, h$  s.t.  $q \in \{2..11\} \wedge h \xrightarrow{*} \text{Loc}[q]$  do
      if  $\text{PC}[q, h] < C(h)$  then  $\text{PC}[q, h] := C(h)$  fi
od

```

Notice that the value of Loc array is changed directly at statements 18. p , 36. p , and 38. p (by our atomicity assumption, each of 36. p and 38. p establishes $p@0$ and includes the assignments to $\text{Loc}[p]$ and $\text{Loc}[p + n]$ that appear after statement 38. p). This is because these statements are not within p 's entry section. In addition, statement 18. p updates $\text{Loc}[p]$ to move from a splitter to its ancestor, and statements 36. p and 38. p reinitialize $\text{Loc}[p]$ and $\text{Loc}[p + N]$ 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 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 is a candidate of splitter i .

$$\text{invariant } A(p, i) \wedge (p@5..9, 11..17 \vee (p@18..31 \wedge p.j = p.level)) \Rightarrow Y[i] = (false, 0) \quad (\text{I1})$$

$$\text{invariant } A(p, i) \wedge (p@9, 11..17 \vee (p@18..31 \wedge p.j = p.level)) \Rightarrow \text{Round}[p.y.rnd] = true \quad (\text{I2})$$

$$\text{invariant } A(p, i) \wedge (p@11..17 \vee (p@18..30 \wedge p.j = p.level)) \Rightarrow \text{Reset}[i].rnd = p.y.rnd \quad (\text{I3})$$

$$\text{invariant } (\exists p :: A(p, i) \wedge p@13, 14, 17..36) = (\text{Acquired}[i] = true) \quad (\text{I4})$$

II) Invariants that prevent “interference” of Round entries. These invariants are used to show that if $p@7..10$ holds and process p is not a candidate at 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.

/* all variable declarations are as defined in Figure 6 except as noted here */

shared auxiliary variables

Loc: **array**[0..2N - 1] of 0..2T + 1 **initially** 0;
 Dist: **array**[1..S] of (\perp , 0..S - 1);
 PC: **array**[0..N - 1, 1..T] of 0..2N **initially** 0

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: *X*[*node*], *dir* := p, stop;

— UpdateLoc({q :: q@{3..6} ∧ q.node = node},
 — 2 · node)

3: *y* := *Y*[*node*];

if ¬*y*.free **then** *dir* := right

— UpdateLoc({p}, 2 · node + 1)

else

4: *Y*[*node*] := (false, 0);

5: *Inuse*[p] := *y*.rnd;

6: **if** *X*[*node*] ≠ p ∨

7: *Acquired*[*node*] **then**

dir := left;

— UpdateLoc({p}, 2 · node)

else

8: *Round*[*y*.rnd] := true;

9: **if** *Reset*[*node*] ≠ *y* **then**

10: *Round*[*y*.rnd], *dir* := false, left

fi

fi

fi;

11: *path*[*level*] := (*node*, *dir*);

if *dir* ≠ stop **then**

level, *node* := *level* + 1, 2 · *node*;

if *dir* = right **then** *node* := *node* + 1 **fi**

fi

until (*level* > L) ∨ (*dir* = stop);

if *level* ≤ L **then** /* got a name */

/* compete in renaming tree, then 2-proc. alg. */

12: *Acquired*[*node*] := true;

for j := *level* **downto** 0 **do**

13: ENTRY₃(*path*[j].*node*, *path*[j].*dir*)

od;

14: ENTRY₂(0)

else /* didn't get a name */

/* compete in overflow tree, then 2-proc. alg. */

15: ENTRY_N(p);

16: ENTRY₂(1)

fi;

private auxiliary variable

m: 1..T

initially

(∀j : 1 ≤ j ≤ T :: Dist[j] = \perp) ∧

(∀j : T < j ≤ S :: Dist[j] = j - T - 1)

17: Critical Section;

/* reset splitters */

for j := min(*level*, L) **downto** 0 **do**

if *path*[j].*dir* ≠ right **then**

18: *n* := *path*[j].*node*;

— Loc[p] := *n*;

19: *Y*[*n*] := (false, 0);

20: *X*[*n*] := p;

— UpdateLoc({q :: q@{3..6} ∧ q.node = n},
 — 2 · n)

21: *y* := *Reset*[*n*];

22: *Reset*[*n*] := (false, *y*.rnd);

— UpdateLoc({q :: q@{4..9} ∧ q.node = n},
 — 2 · n)

23: **if** j = *level* ∨ ¬*Round*[*y*.rnd] **then**

24: *ptr* := *Check*;

25: *usedrnd* := *Inuse*[*ptr*];

26: **if** *usedrnd* ≠ 0 **then**

MoveToTail(*Free*, *usedrnd*)

fi;

27: *Check* := *ptr* + 1 mod N;

28: *Enqueue*(*Free*, *y*.rnd);

29: *nextrnd* := *Dequeue*(*Free*);

30: *Reset*[*n*] := (true, *nextrnd*);

31: *Y*[*n*] := (true, *nextrnd*)

fi;

if j = *level* **then**

32: *Round*[*y*.rnd] := false;

33: *Inuse*[p] := 0

fi

od;

od;

/* execute appropriate exit sections */

if *level* ≤ L **then**

34: EXIT₂(0);

for j := 0 **to** *level* **do**

35: EXIT₃(*path*[j].*node*, *path*[j].*dir*)

od;

36: *Acquired*[*node*] := false

else

37: EXIT₂(1);

38: EXIT_N(p)

fi

— Loc[p], Loc[p + N] := 0, 0

od

Figure 7: Algorithm L with auxiliary variables added.

$$\begin{aligned}
\text{invariant } p@{4..6} \wedge X[p.node] = p &\Rightarrow \\
\text{Reset}[p.node] = p.y \wedge \text{Dist}[p.y.rnd] = \perp \wedge (\forall q :: q@{30,31} \Rightarrow p.y.rnd \neq q.nextrnd) &\quad (I5) \\
\text{invariant } p@{7..10} \wedge (\exists q :: q@{27}) &\Rightarrow \\
\text{Reset}[p.node].rnd = p.y.rnd \vee \\
\text{Dist}[p.y.rnd] > 2N - 2 \cdot ((\text{Check} - p) \bmod N) - 2 &\quad (I6) \\
\text{invariant } p@{7..10} \wedge (\exists q :: q@{28,29}) &\Rightarrow \\
\text{Reset}[p.node].rnd = p.y.rnd \vee \\
\text{Dist}[p.y.rnd] > 2N - 2 \cdot ((\text{Check} - p - 1) \bmod N) - 2 &\quad (I7) \\
\text{invariant } p@{7..10} \wedge \neg(\exists q :: q@{27..29}) &\Rightarrow \\
\text{Reset}[p.node].rnd = p.y.rnd \vee \\
\text{Dist}[p.y.rnd] > 2N - 2 \cdot ((\text{Check} - p - 1) \bmod N) - 3 &\quad (I8) \\
\text{invariant } p@{7..10} \wedge q@{30,31} \Rightarrow p.y.rnd \neq q.nextrnd &\quad (I9) \\
\text{invariant } p@{10} \wedge \text{Reset}[p.node].rnd = p.y.rnd \Rightarrow \text{Reset}[p.node].free = false &\quad (I10) \\
\text{invariant } Y[i].free = true \Rightarrow \text{Dist}[Y[i].rnd] = \perp \wedge (\forall p :: p@{30,31} \Rightarrow Y[i].rnd \neq p.nextrnd) &\quad (I11) \\
\text{invariant } p@{30} \Rightarrow (\forall i :: \text{Reset}[i].rnd \neq p.nextrnd) &\quad (I12) \\
\text{invariant } p@{31,32} \Rightarrow (\forall i :: \text{Reset}[i].rnd \neq p.y.rnd) &\quad (I13) \\
\text{invariant } \text{Dist}[\text{Reset}[i].rnd] = \perp \vee \\
(\exists p :: p@{29} \wedge p.n = i \wedge \text{Dist}[\text{Reset}[i].rnd] = 2N) \vee \\
(\exists p :: p@{30} \wedge p.n = i \wedge \text{Dist}[\text{Reset}[i].rnd] = 2N - 1) &\quad (I14) \\
\text{invariant } i \neq h \Rightarrow \text{Reset}[i].rnd \neq \text{Reset}[h].rnd &\quad (I15) \\
\text{invariant } p@{7..10} \wedge q@{26} \wedge (\text{Check} = p) \Rightarrow \\
\text{Reset}[p.node].rnd = p.y.rnd \vee q.usedrnd = p.y.rnd &\quad (I16)
\end{aligned}$$

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

$$\begin{aligned}
\text{invariant } A(p,i) \wedge A(q,i) \wedge p \neq q &\Rightarrow \\
\neg[p@{4..7} \wedge (q@{4..9,11..17} \vee (q@{18..31} \wedge q.j = q.level))] &\quad (I17) \\
\text{invariant } A(p,i) \wedge A(q,i) \wedge p \neq q &\Rightarrow \neg(p@{8,9,11..36} \wedge q@{8,9,11..36}) &\quad (I18) \\
\text{invariant } A(p,i) \wedge q.n = i &\Rightarrow \neg(p@{4..6} \wedge q@{21..31}) &\quad (I19) \\
\text{invariant } A(p,i) \wedge q.n = i &\Rightarrow \neg(p@{7,8} \wedge q@{23..31}) &\quad (I20) \\
\text{invariant } A(p,i) \wedge q.n = i &\Rightarrow \neg(p@{9,11..34} \wedge q@{24..31}) &\quad (I21) \\
\text{invariant } A(p,i) \wedge (p@{7..9,11..17} \vee (p@{18..31} \wedge p.j = p.level)) \wedge q@{7..10} \wedge p \neq q &\Rightarrow \\
p.y.rnd \neq q.y.rnd &\quad (I22)
\end{aligned}$$

IV) Miscellaneous invariants that are either trivial or follow almost directly from the mutual exclusion property (I61). In (I43) and (I44), $\|Free\|$ denotes the length of the *Free* queue.

$$\begin{aligned}
\text{invariant } p@{3..10} &\Rightarrow y.dir = stop &\quad (I23) \\
\text{invariant } p@{22} &\Rightarrow \text{Reset}[p.n] = p.y &\quad (I24) \\
\text{invariant } p@{23..30} &\Rightarrow \text{Reset}[p.n] = (false, p.y.rnd) &\quad (I25) \\
\text{invariant } p@{20..31} &\Rightarrow Y[p.n] = (false, 0) &\quad (I26) \\
\text{invariant } Y[i].free = true &\Rightarrow Y[i] = \text{Reset}[i] &\quad (I27) \\
\text{invariant } \text{Reset}[i].rnd &\neq 0 &\quad (I28) \\
\text{invariant } p@{4..10} &\Rightarrow p.y.free = true &\quad (I29) \\
\text{invariant } p@{4..10,22..33} &\Rightarrow p.y.rnd \neq 0 &\quad (I30) \\
\text{invariant } (p@{6..17} \vee (p@{18..33} \wedge p.j = p.level)) \wedge (p@{11} \Rightarrow p.dir \neq right) &\Rightarrow \\
Inuse[p] = p.y.rnd \wedge p.y.rnd \neq 0 &\quad (I31) \\
\text{invariant } Round[i] = true \Rightarrow (\exists p :: p.y.rnd = i \wedge (p@{9..17} \vee (p@{18..32} \wedge p.j = p.level))) \wedge \\
(p@{11} \Rightarrow (p.dir = stop \wedge p.level \leq L)) &\quad (I32) \\
\text{invariant } p@{2..11} &\Rightarrow (0 \leq p.level \leq L) \wedge (1 \leq p.node \leq T) &\quad (I33) \\
\text{invariant } p@{19..33} &\Rightarrow 1 \leq p.n \leq T \wedge p.n = p.path[p.j].node &\quad (I34) \\
\text{invariant } p@{19..33} \wedge p.j = p.level &\Rightarrow p.n = p.node &\quad (I35) \\
\text{invariant } p@{2..38} &\Rightarrow (0 \leq p.level \leq L + 1) \wedge (lev(p.node) = p.level) &\quad (I36) \\
\text{invariant } p@{32,33} &\Rightarrow p.j = p.level &\quad (I37)
\end{aligned}$$

$$\text{invariant } p@\{12..38\} \wedge (p.\text{node} \leq T) \Rightarrow p.\text{path}[p.\text{level}] = (p.\text{node}, \text{stop}) \wedge p.\text{dir} = \text{stop} \quad (\text{I38})$$

$$\text{invariant } p@\{2..38\} \wedge (2i \xrightarrow{*} p.\text{node}) \Rightarrow p.\text{path}[\text{lev}(i)] = (i, \text{left}) \quad (\text{I39})$$

$$\text{invariant } (p@\{11\} \wedge p.\text{dir} = \text{stop}) \vee p@\{12..38\} \Rightarrow 1 = p.\text{path}[0] \wedge p.\text{path}[p.\text{level} - 1] \xrightarrow{*} p.\text{node} \wedge (\forall l : 0 \leq l < p.\text{level} - 1 :: p.\text{path}[l] \xrightarrow{*} p.\text{path}[l + 1]) \quad (\text{I40})$$

$$\text{invariant } p@\{11\} \wedge p.\text{dir} \neq \text{stop} \wedge p.\text{level} \geq L \Rightarrow 1 = p.\text{path}[0] \wedge p.\text{path}[p.\text{level}] \xrightarrow{*} 2 \cdot p.\text{node} \wedge (\forall l : 0 \leq l < p.\text{level} :: p.\text{path}[l] \xrightarrow{*} p.\text{path}[l + 1]) \quad (\text{I41})$$

$$\text{invariant } p@\{19..31\} \Rightarrow (p.n = p.\text{node}) \vee (2 \cdot p.n \xrightarrow{*} p.\text{node}) \quad (\text{I42})$$

$$\text{invariant } \neg(\exists p :: p@\{29\}) \Rightarrow \|\text{Free}\| = 2N \quad (\text{I43})$$

$$\text{invariant } (\exists p :: p@\{29\}) \Rightarrow \|\text{Free}\| = 2N + 1 \quad (\text{I44})$$

$$\text{invariant } p@\{25..27\} \Rightarrow p.\text{ptr} = \text{Check} \quad (\text{I45})$$

$$\text{invariant } p@\{30, 31\} \Rightarrow \text{Dist}[p.\text{next}.\text{rnd}] = \perp \quad (\text{I46})$$

$$\text{invariant } p@\{29\} \Rightarrow \text{Dist}[p.y.\text{rnd}] = 2N \quad (\text{I47})$$

$$\text{invariant } p@\{30\} \Rightarrow \text{Dist}[p.y.\text{rnd}] = 2N - 1 \quad (\text{I48})$$

V) Invariants that determine the value of the auxiliary variable Loc .

$$\text{invariant } p@\{0, 1\} \Rightarrow \text{Loc}[p] = 1 \wedge \text{Loc}[p + N] = 1 \quad (\text{I49})$$

$$\text{invariant } p@\{2\} \Rightarrow \text{Loc}[p] = p.\text{node} \wedge \text{Loc}[p + N] = p.\text{node} \quad (\text{I50})$$

$$\text{invariant } p@\{3\} \Rightarrow (X[p.\text{node}] = p \Rightarrow \text{Loc}[p] = p.\text{node} \wedge \text{Loc}[p + N] = p.\text{node}) \wedge (X[p.\text{node}] \neq p \Rightarrow \text{Loc}[p] = 2 \cdot p.\text{node} \wedge \text{Loc}[p + N] = 2 \cdot p.\text{node}) \quad (\text{I51})$$

$$\text{invariant } p@\{4..10\} \Rightarrow (A(p, p.\text{node}) \Rightarrow \text{Loc}[p] = p.\text{node} \wedge \text{Loc}[p + N] = p.\text{node}) \wedge (\neg A(p, p.\text{node}) \Rightarrow \text{Loc}[p] = 2 \cdot p.\text{node} \wedge \text{Loc}[p + N] = 2 \cdot p.\text{node}) \quad (\text{I52})$$

$$\text{invariant } p@\{11\} \Rightarrow (p.\text{dir} = \text{stop} \Rightarrow \text{Loc}[p] = p.\text{node} \wedge \text{Loc}[p + N] = p.\text{node}) \wedge (p.\text{dir} = \text{left} \Rightarrow \text{Loc}[p] = 2 \cdot p.\text{node} \wedge \text{Loc}[p + N] = 2 \cdot p.\text{node}) \wedge (p.\text{dir} = \text{right} \Rightarrow \text{Loc}[p] = 2 \cdot p.\text{node} + 1 \wedge \text{Loc}[p + N] = 2 \cdot p.\text{node} + 1) \quad (\text{I53})$$

$$\text{invariant } p@\{12..38\} \Rightarrow \text{Loc}[p + N] = p.\text{node} \quad (\text{I54})$$

$$\text{invariant } p@\{12..17\} \Rightarrow \text{Loc}[p] = p.\text{node} \quad (\text{I55})$$

$$\text{invariant } p@\{18\} \Rightarrow p.\text{path}[p.j].\text{node} \xrightarrow{*} \text{Loc}[p] \quad (\text{I56})$$

$$\text{invariant } p@\{19..33\} \Rightarrow \text{Loc}[p] = p.n \quad (\text{I57})$$

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

$$\text{invariant } Y[i].\text{free} = \text{false} \Rightarrow (\exists p :: (p@\{5..17\} \vee (p@\{18..31\} \wedge p.j = p.\text{level}))) \wedge (p.\text{node} = i) \wedge (p@\{11\} \Rightarrow p.\text{dir} \neq \text{right})) \vee (\exists p :: p@\{2..33\} \wedge (2i \xrightarrow{*} p.\text{node}) \wedge (p@\{18..33\} \Rightarrow 2i \xrightarrow{*} p.\text{path}[p.j].\text{node})) \vee (\exists p :: p@\{18..31\} \wedge p.\text{path}[p.j].\text{node} = i \wedge (p@\{18\} \Rightarrow 2i \xrightarrow{*} \text{Loc}[p])) \quad (\text{I58})$$

$$\text{invariant } p@\{2..11\} \wedge (i \xrightarrow{*} \text{Loc}[p]) \Rightarrow \text{PC}[p, i] \geq C(i) \quad (\text{I59})$$

$$\text{invariant } p@\{2..11\} \wedge (h \xrightarrow{*} i \xrightarrow{*} \text{Loc}[p]) \wedge i \neq h \Rightarrow \text{PC}[p, i] < \text{PC}[p, h] \quad (\text{I60})$$

VII) Invariants that prove mutual exclusion and contention sensitivity.

$$\text{invariant } (\text{Mutual exclusion}) \quad |\{p :: p@\{17..34, 37\}\}| \leq 1 \quad (\text{I61})$$

$$\text{invariant } (\text{Contention sensitivity}) \quad p@\{2..38\} \Rightarrow \text{lev}(p.\text{node}) < \text{PC}[p, 1] \quad (\text{I62})$$

Proofs of Invariants

We now prove that each of (I1)–(I62) 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 holds initially, so we leave this part of the proof to the reader.) The following lemma is used in several of the proofs.

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

- u is reached from t via the execution of statement $3.p$.
- $p@{4}$ 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:

- $1.p$ and $11.p$, which establish $p.node = i$.
- $3.p$ and $10.p$, which establish $p@{4..9, 11..38}$.
- $2.p$, which establishes $X[i] = p$.
- $6.p$, which falsifies $p@{4..6}$.
- $3.p, 22.q$, and $30.q$, where q is any arbitrary process, which could establish $Reset[i] = p.y$.
- $6.p, 7.p$, and $9.p$, which falsify $p@{4..9}$.
- $3.p, 6.p, 7.p$, and $9.p$, which could establish $p@{11} \wedge p.dir = stop$.

We now show that none of these statements other than $3.p$ can establish $A(p, i)$, and $3.p$ can do so only if the conditions specified in the lemma are met.

Statements $1.p$ and $2.p$ establish $p@{2, 3}$. Therefore, they cannot establish $A(p, i)$.

Statement $3.p$ can establish $A(p, i)$ by establishing $p@{4}$ only if $p.node = i \wedge Y[i].free = true$ holds at t . But then this condition also holds at u .

Statement $6.p$ establishes either $p@{11}$ or $p@{7}$. If it establishes $p@{11}$, then it also establishes $p.dir = left$, in which case $A(p, i)$ is false. If $6.p$ establishes $p@{7}$, then $A(p, i)$ holds at u only if $p.node = i \wedge X[i] = p \wedge Reset[i] = p.y$ holds at t . But this contradicts our assumption that $A(p, i)$ is false at t .

Statements $7.p$ and $10.p$ can establish $A(p, i)$ only if they establish $p@{11}$. In this case, they also establish $p.dir = left$, which implies that $A(p, i)$ is false.

Statement $9.p$ can establish $A(p, i)$ only by establishing $p@{11}$. However, $9.p$ establishes $p@{11}$ only if $p.node = i \wedge Reset[i] = p.y$ holds at t , which implies that $A(p, i)$ also holds at t , a contradiction.

Statement $11.p$ establishes $p@{2, 12}$. If it establishes $p@{2}$, then $A(p, i)$ is false at u . If it establishes $p@{12}$, then either $p.dir = stop$ or $p.level = L$ holds at t . In the former case, $11.p$ 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. If $p.level = L$, then by (I36), $p.node > T$ holds at u . Thus, $p.node \neq i$ at u , which implies that $A(p, i)$ is false at u .

Statements $22.p$, and $30.p$ 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 $22.q$, where $q \neq p$, can establish $A(p, i)$ only by establishing $Reset[i] = p.y$ when $p@{4..9} \wedge q.n = i$ holds. In this case, $22.q$ establishes $Reset[i].free = false$. By (I29), if $p@{4..9}$ holds at t , then $p.y.free = true$ holds at t , and hence also at u . Therefore, 22.1 cannot establish $Reset[i] = p.y$.

Statement $30.q$ can establish $A(p, i)$ only by establishing $Reset[i] = p.y$ when $p@{4..9} \wedge q.n = i$ holds. We consider the two cases $p@{4..6}$ and $p@{7..9}$ separately.

If $p@{4..6}$ holds at t , then $30.q$ can establish $A(p, i)$ only if $X[i] = p$ holds at t . By (I5), this implies that $Reset[i].rnd = p.y.rnd$ holds at t . Therefore, if statement $30.q$ changes the value of $Reset[i].rnd$, it must establish $Reset[i].rnd \neq p.y.rnd$.

If $p@{7..9}$ holds at t , then by (I9), $p.y.rnd \neq q.nextrnd$ also holds. Therefore, $30.q$ falsifies $Reset[i].rnd = p.y.rnd$. \square

We now prove each invariant listed above.

$$\text{invariant } A(p, i) \wedge (p@{5..9, 11..17} \vee (p@{18..31} \wedge p.j = p.level)) \Rightarrow Y[i] = (false, 0) \quad (\text{I1})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. However, if statement $3.p$ establishes $A(p, i)$, then it also establishes $p@{4}$. Hence, it cannot falsify (I1).

The only statements that can establish $p@{5..9, 11..17}$ are $3.p$, $4.p$, and $10.p$. Statement $4.p$ establishes the consequent of (I1). If statements $3.p$ and $10.p$ can establish $p@{5..9, 11..17}$ only by establishing $p@{11}$, in which case they also establish $p.dir \neq stop$. Thus, if these statements establish $p@{5..9, 11..17}$, then they also falsify $A(p, i)$.

The only statement that can establish $p@{18..31} \wedge p.j = p.level$ is $17.p$, which may establish the antecedent only if executed when $A(p, i)$ holds. But this implies that the antecedent holds before the execution of $17.p$. It follows that, although statement $17.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statement $31.q$, where q is any arbitrary process. If $q = p$, then statement $31.q$ establishes $(p@{18} \wedge p.j < p.level) \vee p@{32, 34, 37}$, which implies that the antecedent is false.

Suppose that $q \neq p$. Statement $31.q$ can falsify the consequent only if executed when $q.n = i$ holds. However, by (I19), (I20), and (I21), the antecedent and $q@{31} \wedge q.n = i$ cannot hold simultaneously. \square

$$\text{invariant } A(p, i) \wedge (p@{9, 11..17} \vee (p@{18..31} \wedge p.j = p.level)) \Rightarrow Round[p.y.rnd] = true \quad (\text{I2})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. However, if statement $3.p$ establishes $A(p, i)$, then it also establishes $p@{4}$. Hence, it cannot falsify (I2).

The condition $p@{9, 11..17}$ may be established only by statements $3.p$, $6.p$, $7.p$, $8.p$, and $10.p$. Statement $8.p$ establishes the consequent. If statements $3.p$, $6.p$, $7.p$, and $10.p$ establish $p@{9, 11..17}$, then they also establish $p@{11} \wedge p.dir \neq stop$, which implies that $A(p, i)$ is false.

The only statement that can establish $p@{18..31} \wedge p.j = p.level$ is $17.p$, which may establish the antecedent only if executed when $A(p, i)$ holds. But this implies that the antecedent holds before the execution of $17.p$. It follows that, although statement $17.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements $3.p$ and $21.p$ (which may change the value of $p.y.rnd$) and $10.q$ and $32.q$ (which assign the value *false* to an element of the *Round* array), where q is any arbitrary process. Statement $3.p$ establishes $p@{4} \vee (p@{11} \wedge p.dir = right)$, which implies that the antecedent is false. If both the antecedent and $p@{21}$ hold, then by (I35) and the definition of $A(p, i)$, $p.n = p.node = i$ holds, and by (I3), $Reset[i].rnd = p.y.rnd$ holds. It follows that statement $21.p$ cannot change the value of $p.y.rnd$ while the antecedent holds, and hence cannot falsify (I2).

If $q = p$, then statement $10.q$ establishes $p@{11} \wedge p.dir = left$ and statement $32.q$ establishes $p@{33}$, both of which imply that the antecedent is false.

Suppose that $q \neq p$. In this case, statements $10.q$ and $32.q$ may falsify the consequent only if $p.y.rnd = q.y.rnd$ holds. By (I22), $p.y.rnd = q.y.rnd \wedge q@{10}$ implies that the antecedent of (I2) is false. Thus, statement $10.q$ cannot falsify (I2).

By (I61), if the antecedent and $q@{32}$ hold, then $A(p, i) \wedge p@{9, 11..16}$ holds. By the definition of $A(p, i)$, $A(p, i) \wedge p@{9}$ implies that $Reset[i].rnd = p.y.rnd$ holds. By (I3), $p@{11..16}$ also implies that $Reset[i].rnd = p.y.rnd$ holds. By (I13), $Reset[i].rnd = p.y.rnd \wedge q@{32}$ implies that $p.y.rnd \neq q.y.rnd$ holds. Thus, statement $32.q$ cannot falsify (I2). \square

$$\text{invariant } A(p, i) \wedge (p@{11..17} \vee (p@{18..30} \wedge p.j = p.level)) \Rightarrow Reset[i].rnd = p.y.rnd \quad (\text{I3})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. However, if statement $3.p$ establishes $A(p, i)$, then it also establishes $p@{4}$. Hence, it cannot falsify (I2).

The condition $p@\{11..17\}$ may be established only by $3.p$, $6.p$, $7.p$, $9.p$, and $10.p$. If statements $3.p$, $6.p$, $7.p$, and $10.p$ establish $p@\{11..19\}$, then they also establish $p@\{11\} \wedge p.dir \neq stop$, which implies that $A(p, i)$ is false. Statement $9.p$ can establish the antecedent only if $Reset[i] = p.y$ holds. Hence, it preserves (I3).

The only statement that can establish $p@\{18..30\} \wedge p.j = p.level$ is $17.p$, which may establish the antecedent only if executed when $A(p, i)$ holds. But this implies that the antecedent holds before the execution of $17.p$. It follows that, although statement $17.p$ may preserve the antecedent, it cannot establish it.

The consequent may be falsified only by statements $3.p$ and $21.p$ (which may change the value of $p.y.rnd$) and $22.q$ and $30.q$ (which may update $Reset[i].rnd$), where q is any arbitrary process. The antecedent is false after the execution of $3.p$ (as explained above) and and $30.p$ (which establishes $p@\{31\}$). If $A(p, i) \wedge p@\{21, 22\}$ holds, then by (I35), $p.n = i$. Thus, $Reset[i].rnd = p.y.rnd$ holds after the execution of $21.p$ or $22.p$.

This leaves only statements $22.q$ and $30.q$, where $q \neq p$. By (I24), statement $22.q$ cannot change $Reset[i].rnd$, and hence, cannot falsify (I3). Statement $30.q$ may falsify the consequent only if $q.n = i$ holds. By (I21), $q.n = i \wedge q@\{30\}$ implies that the antecedent of (I3) is false. Thus, statement $30.q$ cannot falsify (I3). \square

$$\text{invariant } (\exists p :: A(p, i) \wedge p@\{13, 14, 17..36\}) = (Acquired[i] = true) \quad (\text{I4})$$

Proof: By the definition of $A(p, i)$, the left-hand side of (I4) is equivalent to $(\exists p :: p.node = i \wedge p@\{13, 14, 17..36\})$. Note that $p.node$ is updated only by $1.p$ and $11.p$. Thus, $p.node = i \wedge p@\{13, 14, 17..36\}$ can be established or falsified only by statements that either establish or falsify $p@\{13, 14, 17..36\}$. These statements include $12.p$ and $16.p$ (which establish $p@\{13, 14, 17..36\}$) and $17.p$, $23.p$, $31.p$, $33.p$, and $36.p$ (which may falsify $p@\{13, 14, 17..36\}$). $12.p$ and $36.p$ are also the only statements that may establish or falsify the right-hand side of (I4) (p can be any process here).

Statement $12.p$ establishes the left-hand side if and only if it also establishes the right-hand side. Statement $36.p$ may falsify the right-hand side only if executed when $p.node = i$ holds, in which case $A(p, i)$ holds. By (I18), this implies that the left-hand side of (I4) is falsified.

Statement $16.p$ may be executed only when $p.level > L$. Similarly, due to the **if** test prior to statement 34 , statements $17.p$, $23.p$, $31.p$, and $33.p$ may falsify $p@\{13, 14, 17..36\}$ by establishing $p@\{37\}$ only if executed when $p.level > L$. By (I36), $p.level > L$ 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). \square

$$\text{invariant } p@\{4..6\} \wedge X[p.node] = p \Rightarrow \\ Reset[p.node] = p.y \wedge Dist[p.y.rnd] = \perp \wedge (\forall q :: q@\{30, 31\} \Rightarrow p.y.rnd \neq q.nextrnd) \quad (\text{I5})$$

Proof: The antecedent may be established only by statements $3.p$ (which establishes $p@\{4..6\}$), $1.p$ and $11.p$ (which update $p.node$), and $2.p$ and $20.p$ (which may establish $X[p.node] = p$). However, statements $1.p$, $2.p$, $11.p$, and $20.p$ establish $p@\{2, 3, 12, 21\}$ and hence cannot establish the antecedent. Also, by (I27) and (I11), if $3.p$ establishes the antecedent, then it also establishes the consequent.

The consequent may be falsified only by statements $3.p$ and $21.p$ (which may change the value of $p.y$), $1.p$ and $11.p$ (which may update $p.node$), $29.r$ (which may establish $r@\{30, 31\} \wedge p.y.rnd = r.nextrnd$), $22.r$ and $30.r$ (which may update $Reset[p.node]$), and $28.r$ (which may falsify $Dist[p.y.rnd] = \perp$), where r is any arbitrary process. However, statement $3.p$ preserves (I5) as shown above. Furthermore, the antecedent is false after the execution of $1.p$, $11.p$, and $21.p$ and also after the execution of each of $22.r$, $28.r$, $29.r$, and $30.r$ if $r = p$.

Consider statements $22.r$, $28.r$, and $30.r$, where $r \neq p$. If the antecedent and consequent of (I5) both hold, then by (I33), $p.node \leq T$ holds, and hence $A(p, p.node)$ holds. Statements $22.r$ and $30.r$ may falsify

the consequent only if $r.n = p.node$ holds. Similarly, statement $28.r$ may falsify the consequent only if $r.y.rnd = p.y.rnd$. If $r.y.rnd = p.y.rnd$ and the consequent both hold, then by (I25), $Reset[p.node].rnd = Reset[r.n].rnd$. By (I5), 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), $r.n = p.node \wedge r@\{22, 28, 30\} \wedge A(p, p.node)$ implies that $p@\{4..6\}$ is false. Thus, these statements cannot falsify (I5).

Finally, statement $29.r$ may establish $p.y.rnd = r.nextrnd$ only if $p.y.rnd$ is at the head of *Free* queue, i.e., $Dist[p.y.rnd] = 0$. But this implies that $Dist[p.y.rnd] = \perp$ is false. Because (I5) is assumed to hold prior to the execution of $29.r$, this implies that the antecedent of (I5) is false before $29.r$ is executed. Thus, the antecedent is also false after the execution of $29.r$. \square

$$\begin{aligned} \text{invariant } p@\{7..10\} \wedge (\exists q :: q@\{27\}) \Rightarrow \\ Reset[p.node].rnd = p.y.rnd \vee \\ Dist[p.y.rnd] > 2N - 2 \cdot ((Check - p) \bmod N) - 2 \end{aligned} \quad (I6)$$

Proof: The antecedent may be established only by statements $6.p$ and $26.q$, where q is any process. Statement $6.p$ 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 $26.q$ may establish the consequent only if executed when $p@\{7..10\}$ holds. By (I61), $q@\{26\} \wedge p@\{7..10\}$ implies that the antecedent of (I8) holds. By the consequent of (I8),

$$(Reset[p.node].rnd = p.y.rnd) \vee (Dist[p.y.rnd] > 2N - 2 \cdot ((Check - p - 1) \bmod N) - 3) \quad (1)$$

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

- $Reset[p.node].rnd = p.y.rnd$ before $26.q$ is executed: In this case, $Reset[p.node].rnd = p.y.rnd$ holds after $26.q$, so (I6) is preserved.
- $Check = p \wedge Reset[p.node].rnd \neq p.y.rnd$ before $26.q$ is executed: In this case, by (I30), $p.y.rnd \neq 0$ holds, and by (I16), $q.usedrnd = p.y.rnd$ holds. Therefore, procedure *MoveToTail* is called. By (I61) and (I43), $q@\{26\}$ implies that $\|Free\| = 2N$. Thus, statement $26.q$ establishes $Dist[p.y.rnd] = 2N - 1$, which implies the consequent.
- $Check \neq p \wedge Reset[p.node].rnd \neq p.y.rnd$ before $26.q$ is executed: By (1), this implies $Dist[p.y.rnd] > 2N - 2 \cdot ((Check - p - 1) \bmod N) - 3$. Note that the value of $Dist[p.y.rnd]$ can decrease by at most one by a call to *MoveToTail*. Note also that if $Check \neq p$, then $(Check - p) \bmod N = ((Check - p - 1) \bmod N) + 1$. Therefore, after the execution of $26.q$,

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

The consequent may be falsified only by statements $1.p$ and $11.p$ (which may update $p.node$), $3.p$ and $21.p$ (which may change the value of $p.y$), $22.r$ and $30.r$ (which may update $Reset[p.node].rnd$), $26.r$, $28.r$, and $29.r$ (which may update $Dist[p.y.rnd]$), and $27.r$ (which may change the value of $Check$), where r is any arbitrary process. However, statements $1.p$, $3.p$, $11.p$, and $21.p$ falsify the antecedent. By (I61), statements $22.r$, $26.r$, $28.r$, $29.r$, and $30.r$ cannot be executed while the antecedent holds. Finally, by (I61), statement $27.r$ falsifies the antecedent. \square

$$\begin{aligned} \text{invariant } p@\{7..10\} \wedge (\exists q :: q@\{28, 29\}) \Rightarrow \\ Reset[p.node].rnd = p.y.rnd \vee \\ Dist[p.y.rnd] > 2N - 2 \cdot ((Check - p - 1) \bmod N) - 2 \end{aligned} \quad (I7)$$

Proof: The antecedent may be established only by statements $6.p$ and $27.q$, where q is any process. Statement $6.p$ 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 $27.q$ can establish the antecedent only if $q@{27} \wedge p@{7..10}$ holds. This implies that the consequent of (I6) holds before statement $27.q$ is executed. By (I45), statement $27.q$ 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 $11.p$ (which may update $p.node$), $3.p$ and $21.p$ (which may change the value of $p.y$), $22.r$ and $30.r$ (which may update $Reset[p.node].rnd$), $26.r$, $28.r$, and $29.r$ (which may update $Dist[p.y.rnd]$), and $27.r$ (which may change the value of $Check$), where r is any arbitrary process. However, statements $1.p$, $3.p$, $11.p$, and $21.p$ falsify the antecedent. By (I61), statements $22.r$, $26.r$, $27.r$, and $30.r$ cannot be executed while the antecedent holds. Statement $29.r$ falsifies the antecedent by (I61).

Statement $28.r$ may update $Dist[p.y.rnd]$ only if executed when $Dist[p.y.rnd] = \perp$, in which case it establishes $Dist[p.y.rnd] = 2N$, by (I61) and (I43). This implies that the consequent holds. \square

$$\begin{aligned} \text{invariant } p@{7..10} \wedge \neg(\exists q :: q@{27..29}) \Rightarrow \\ Reset[p.node].rnd = p.y.rnd \vee \\ Dist[p.y.rnd] > 2N - 2 \cdot ((Check - p - 1) \bmod N) - 3 \end{aligned} \quad (I8)$$

Proof: The antecedent may be established only by statements $6.p$ and $29.q$, where q is any process. Statement $6.p$ 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 $29.q$ may establish the antecedent only if $q@{29} \wedge p@{7..10}$ holds. If $Reset[p.node].rnd = p.y.rnd$ holds before the execution of $29.q$, then it holds afterward as well, and thus (I8) is not falsified. So, assume that $q@{29} \wedge p@{7..10} \wedge Reset[p.node].rnd = p.y.rnd$ holds before the execution of $29.q$. In this case, by (I7), $Dist[p.y.rnd] > 0$ holds. Therefore, the function *Dequeue* decrements $Dist[p.y.rnd]$ by 1. Moreover, by (I7), $Dist[p.y.rnd] > 2N - 2 \cdot (Check - p - 1) \bmod N - 2$ holds before $29.q$ is executed, which implies that $Dist[p.y.rnd] > 2N - 2 \cdot (Check - p - 1) \bmod N - 3$ holds afterward.

The consequent may be falsified only by statements $1.p$ and $11.p$ (which may update $p.node$), $3.p$ and $21.p$ (which may change the value of $p.y$), $22.r$ and $30.r$ (which may update $Reset[p.node].rnd$), $26.r$, $28.r$, and $29.r$ (which may update $Dist[p.y.rnd]$), and $27.r$ (which may change the value of $Check$), where r is any arbitrary process. However, statements $1.p$, $3.p$, $11.p$, and $21.p$ falsify the antecedent. Statement $26.r$ establishes $r@{27}$, and hence falsifies the antecedent. By (I24), statement $22.r$ does not change the value of $Reset[p.node].rnd$. Statements $27.r$ and $28.r$ cannot be executed while the antecedent holds.

Finally, statement $30.r$ may falsify the consequent only if executed when $Reset[p.node].rnd = p.y.rnd \wedge p.node = r.n$ holds. By (I25), this implies that $Reset[r.n].rnd = r.y.rnd$ holds. Also, by (I48), $Dist[r.y.rnd] = 2N - 1$ holds. Combining these assertions, we have $p.y.rnd = r.y.rnd$, and hence $Dist[p.y.rnd] = 2N - 1$. This implies that the consequent of (I8) holds after $30.r$ is executed. \square

$$\text{invariant } p@{7..10} \wedge q@{30,31} \Rightarrow p.y.rnd \neq q.nextrnd \quad (I9)$$

Proof: The antecedent may be established only by statements $6.p$ and $29.q$. Statement $6.p$ may establish the antecedent only if executed when $p@{6} \wedge X[p.node] = p \wedge q@{30,31}$ holds. By (I5), this implies that $p.y.rnd \neq q.nextrnd$ holds. Thus, the consequent of (I9) is true after the execution of $6.p$.

Statement $29.q$ may establish the antecedent of (I9) only if executed when the antecedent of (I7) holds. By (I61), the third disjunct of (I14) does not hold before the execution of $29.q$. Thus, if $Reset[p.node].rnd = p.y.rnd$ holds before $29.q$ is executed, then $Dist[p.y.rnd] = \perp \vee Dist[p.y.rnd] = 2N$ holds. On the other hand, if $Reset[p.node].rnd \neq p.y.rnd$ holds before $29.q$ is executed, then by (I7), $Dist[p.y.rnd] > 0$ 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 $3.p$ and $21.p$ (which may change the value of $p.y$) and $29.q$ (which may update $q.nextrnd$). However, statements $3.p$ and $21.p$ falsify the antecedent, and statement $29.q$ preserves (I9) as shown above. \square

$$p@{10} \wedge \text{Reset}[p.\text{node}].\text{rnd} = p.y.\text{rnd} \Rightarrow \text{Reset}[p.\text{node}].\text{free} = \text{false} \quad (\text{I10})$$

Proof: (I10) may be falsified only by statements 9.*p* (which may establish $p@{10}$), 3.*p* and 21.*p* (which may change the value of $p.y.\text{rnd}$), 1.*p* and 11.*p* (which may change the value of $p.\text{node}$), and 22.*q* and 30.*q* (which may update $\text{Reset}[p.\text{node}]$), where q is any arbitrary process. Statements 1.*p*, 3.*p*, 11.*p*, and 21.*p* establish $p@{2, 4, 11, 12, 22}$, which implies that the antecedent is false. Statement 9.*p* establishes the antecedent only if executed when $\text{Reset}[p.\text{node}] \neq p.y \wedge \text{Reset}[p.\text{node}].\text{rnd} = p.y.\text{rnd}$ holds, which implies that $\text{Reset}[p.\text{node}].\text{free} \neq p.y.\text{free}$. However, by (I29), $p.y.\text{free} = \text{true}$. Thus, $\text{Reset}[p.\text{node}].\text{free} = \text{false}$.

If $q = p$, then each of 22.*q* and 30.*q* establishes $p@{23, 31}$, which implies that the antecedent is false.

Consider statements 22.*q* and 30.*q*, where $q \neq p$. Statement 22.*q* trivially establishes or preserves the consequent. If statement 30.*p* changes the value of $\text{Reset}[p.\text{node}]$, then by (I9), it establishes $p.y.\text{rnd} \neq \text{Reset}[p.\text{node}].\text{rnd}$. Thus, statements 22.*q* and 30.*q* cannot falsify (I10). \square

$$\text{invariant } Y[i].\text{free} = \text{true} \Rightarrow \text{Dist}[Y[i].\text{rnd}] = \perp \wedge (\forall p :: p@{30, 31} \Rightarrow Y[i].\text{rnd} \neq p.\text{next}\text{rnd}) \quad (\text{I11})$$

Proof: The antecedent may be established only by statement 31.*q*, where q is any arbitrary process. But by (I61) and (I46), it establishes $\text{Dist}[Y[i].\text{rnd}] = \perp \wedge \neg(\exists p :: p@{30, 31})$.

The consequent may be falsified only by statements 4.*q*, 19.*q*, and 31.*q* (which may update $Y[i].\text{rnd}$), 28.*q* (which may falsify $\text{Dist}[Y[i].\text{rnd}] = \perp$), and 29.*q* (which may update $q.\text{next}\text{rnd}$, and may also establish $q@{30, 31}$), where q is any arbitrary process. Statements 4.*q* and 19.*q* falsify the antecedent. Statement 31.*q* preserves (I11) as shown above.

Statement 28.*q* may falsify $\text{Dist}[Y[i].\text{rnd}] = \perp$ only if executed when $Y[i].\text{free} = \text{true} \wedge q.y.\text{rnd} = Y[i].\text{rnd}$ holds, $Y[i].\text{rnd} = \text{Reset}[i].\text{rnd}$ holds, by (I27). Furthermore, $\text{Reset}[q.n].\text{rnd} = q.y.\text{rnd}$ also holds, by (I25). It follows that $\text{Reset}[q.n].\text{rnd} = \text{Reset}[i].\text{rnd}$, and hence $q.n = i$, by (I15). By (I26), this in turn implies $Y[i].\text{free} = \text{false}$, a contradiction. It follows that statement 28.*q* cannot falsify the consequent while the antecedent holds.

If $Y[i].\text{free} = \text{false}$ holds before statement 29.*q* is executed, then it holds afterward, and hence (I11) is not falsified. If $Y[i] = \text{true}$ holds before 29.*q* is executed, then $\text{Dist}[Y[i].\text{rnd}] = \perp$ holds as well, since (I11) is presumed to hold before the execution of 29.*q*. Thus, $Y[i].\text{rnd}$ is not in the *Free* queue. This implies that a value different from $Y[i].\text{rnd}$ is dequeued, i.e., $Y[i].\text{rnd} \neq q.\text{next}\text{rnd}$ holds after the execution of 29.*q*. \square

$$\text{invariant } p@{30} \Rightarrow (\forall i :: \text{Reset}[i].\text{rnd} \neq p.\text{next}\text{rnd}) \quad (\text{I12})$$

Proof: The antecedent may be established only by statement 29.*p*, which may establish $\text{Reset}[i].\text{rnd} = p.\text{next}\text{rnd}$ only if executed when $\text{Dist}[\text{Reset}[i].\text{rnd}] = 0$, i.e., when $\text{Reset}[i].\text{rnd}$ is at the head of the *Free* queue. However, this is precluded by (I14).

The consequent may be falsified only by statements 29.*p* (which may update $p.\text{next}\text{rnd}$), and 22.*q* and 30.*q* (which may update $\text{Reset}[i].\text{rnd}$), where q is any arbitrary process. However, statement 29.*p* preserves (I12) as shown above. By (I24), statement 22.*q* does not change the value of $\text{Reset}[i].\text{rnd}$. By (I61), the antecedent is false after the execution of statement 30.*q*. \square

$$\text{invariant } p@{31, 32} \Rightarrow (\forall i :: \text{Reset}[i].\text{rnd} \neq p.y.\text{rnd}) \quad (\text{I13})$$

Proof: The antecedent may be established only by statement 30.*p*. Before its execution, $\text{Reset}[i].\text{rnd} \neq p.\text{next}\text{rnd}$ holds, by (I12), and $\text{Reset}[p.n].\text{rnd} = p.y.\text{rnd}$ holds, by (I25). We consider two cases. First, if $i = p.n$, then before the execution of 30.*p*, $p.y.\text{rnd} \neq p.\text{next}\text{rnd}$ holds. Thus, statement 30.*p* establishes $\text{Reset}[i].\text{rnd} \neq p.y.\text{rnd}$.

Second, suppose that $i \neq p.n$. Then, by (I15), $\text{Reset}[i].\text{rnd} \neq \text{Reset}[p.n].\text{rnd}$ holds. Because $\text{Reset}[p.n].\text{rnd} = p.y.\text{rnd}$ holds before 30.*p* is executed, this implies that $\text{Reset}[i].\text{rnd} \neq p.y.\text{rnd}$ holds as well. Thus, $\text{Reset}[i].\text{rnd} \neq p.y.\text{rnd}$ holds after the execution of 30.*p*.

The consequent may be falsified only by statements $3.p$ and $21.p$ (which may update $p.y.rnd$), and $22.q$ and $30.q$ (which may update $Reset[i].rnd$), where q is any arbitrary process. However, the antecedent is false after the execution of $3.p$ and $21.p$, and by (I61), $22.q$ and $30.q$ are not enabled while the antecedent holds. \square

$$\begin{aligned} \text{invariant } \text{Dist}[Reset[i].rnd] = \perp \vee \\ (\exists p :: p@{29} \wedge p.n = i \wedge \text{Dist}[Reset[i].rnd] = 2N) \vee \\ (\exists p :: p@{30} \wedge p.n = i \wedge \text{Dist}[Reset[i].rnd] = 2N - 1) \end{aligned} \quad (\text{I14})$$

Proof: The only statements that may falsify (I14) are $22.q$ (which may update $Reset[i].rnd$), $30.q$ (which may falsify $(\exists p :: p@{29,30})$ and may update $Reset[i].rnd$), and $26.q$, $28.q$, and $29.q$ (which may update $\text{Dist}[Reset[i].rnd]$), where q is any arbitrary process. By (I24), statement $22.q$ does not change the value of $Reset[i].rnd$.

Statement $30.q$ may falsify (I14) only if executed when $q.n = i$ holds, in which case it establishes $\text{Dist}[Reset[i].rnd] = \perp$, by (I46).

If $26.q$ is enabled, then by (I61), $\neg(\exists p :: p@{29,30})$ holds. Since (I14) is presumed to hold before the execution of $26.q$, this implies that $\text{Dist}[Reset[i].rnd] = \perp$ holds both before and after $26.q$ is executed.

Statement $28.q$ may falsify $\text{Dist}[Reset[i].rnd] = \perp$ only if executed when $q.y.rnd = Reset[i].rnd$ holds. If $q@{28}$ holds, then by (I25), $Reset[q.n].rnd = q.y.rnd$ holds as well. It follows that $Reset[q.n].rnd = Reset[i].rnd$ holds, and hence $q.n = i$, by (I15). Because the *Enqueue* procedure puts $q.y.rnd$ at the tail of the *Free* queue, by (I43) and (I61), it establishes $\text{Dist}[q.y.rnd] = 2N$. Therefore, if statement $28.q$ falsifies $\text{Dist}[Reset[i].rnd] = \perp$, it establishes the second disjunct of the consequent.

Statement $29.q$ may falsify the consequent only if executed when $q@{29} \wedge q.n = i \wedge \text{Dist}[Reset[i].rnd] = 2N$ holds. In this case, the *Dequeue* decrements $\text{Dist}[Reset[i].rnd]$ by one, establishing the third disjunct of the consequent. \square

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

Proof: The only statements that may falsify (I15) are $22.p$ and $30.p$, where p is any arbitrary process. By (I24), statement $22.p$ does not change the value of $Reset[i].rnd$, and hence cannot falsify (I15). Statement $30.p$ may falsify (I15) only if $p.n = h \wedge Reset[i].rnd = p.nextrnd$ holds prior to its execution. However, this is precluded by (I12). \square

$$\begin{aligned} \text{invariant } p@{7..10} \wedge q@{26} \wedge (Check = p) \Rightarrow \\ Reset[p.node].rnd = p.y.rnd \vee q.usedrnd = p.y.rnd \end{aligned} \quad (\text{I16})$$

Proof: The antecedent may be falsified only by statements $6.p$, $25.q$, and $27.r$, where r is any arbitrary process. Statement $6.p$ may establish $p@{7}$ only if executed when $X[p.node] = p$ holds. By (I5), this implies that $Reset[p.node].rnd = p.y.rnd$ holds both before and after $6.p$ is executed.

Statement $25.q$ may establish the antecedent only if executed when $p@{7..10} \wedge q@{25} \wedge Check = p$ holds. By (I31), this implies that $Inuse[p] = p.y.rnd$, and by (I45), it implies that $q.ptr = p$. Thus, statement $25.q$ establishes $q.usedrnd = p.y.rnd$, and hence does not falsify (I16). By (I61), statement $27.r$ cannot be executed while $q@{26}$ holds.

The consequent may be falsified only by statements $1.p$, $3.p$, $11.p$, and $21.p$ (which may update either $p.node$ or $p.y.rnd$), $25.q$ (which may change the value of $q.usedrnd$), and $22.r$ and $30.r$ (which may update $Reset[p.node].rnd$), where r is any arbitrary process. However, $p@{7..10}$ is false after the execution of statements $1.p$, $3.p$, $11.p$, and $21.p$. Statement $25.q$ preserves (I16), as shown above. By (I61), Statements $22.r$ and $30.r$ cannot be executed while $q@{26}$ holds. \square

$$\text{invariant } A(p, i) \wedge A(q, i) \wedge p \neq q \Rightarrow \neg[p@{4..7} \wedge (q@{4..9, 11..17} \vee (q@{18..31} \wedge q.j = q.level))] \quad (\text{I17})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. The only statements that may falsify the consequent are $3.p$ (which may establish $p@{4..7}$), and $3.q$ and $10.q$ (which may establish $q@{4..9, 11..17} \vee (q@{18..31} \wedge q.j = q.level)$). Note that if statements $3.q$ and $10.q$ establish $q@{11}$, 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 $3.p$ or $3.q$ is executed, establishing $p@{4}$ or $q@{4}$, respectively. Without loss of generality, it suffices to consider only statement $3.p$.

If $A(q, i) \wedge q@{4..6}$ holds before $3.p$ 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 $3.p$ is executed. Hence, $A(p, i)$ is false after the execution of $3.p$.

Next, suppose that $A(q, i) \wedge (q@{7..9, 11..17} \vee (q@{18..31} \wedge q.j = q.level))$ holds before $3.p$ is executed, where $q \neq p$. In this case, by (I1), $Y[i].free = false$. This implies that $3.p$ does not establish $p@{4}$. \square

$$\text{invariant } A(p, i) \wedge A(q, i) \wedge p \neq q \Rightarrow \neg(p@{8, 9, 11..36} \wedge q@{8, 9, 11..36}) \quad (\text{I18})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. However, if $3.p$ establishes $A(p, i)$, it also establishes $p@{4}$, and hence it cannot falsify (I18). Similar argument applies to $3.q$.

By symmetry, when considering statements that might falsify the consequent, it suffices to consider only $3.p$, $6.p$, $7.p$, and $10.p$ (which may establish $p@{8, 9, 11..36}$). Statements $3.p$, $6.p$, and $10.p$ can establish $p@{8, 9, 11..36}$ only by establishing $p@{11}$, 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 $7.p$ establishes $p@{11}$.

Statement $7.p$ could also establish $p@{8, 9, 11..36}$ by establishing $p@{8}$. In this case, $Acquired[i] = false$ holds before its execution. If $A(p, i)$ is false before $7.p$ 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 $7.p$ is executed. By (I17), this implies that $A(q, i) \wedge q@{8, 9, 11..17}$ is false. Thus, $7.p$ could potentially falsify (I18) only if executed when $A(q, i) \wedge q@{18..36}$ holds. However, in this case, by (I4), $Acquired[i] = true$. Thus, statement $7.p$ cannot falsify (I18). \square

$$\text{invariant } A(p, i) \wedge q.n = i \Rightarrow \neg(p@{4..6} \wedge q@{21..31}) \quad (\text{I19})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$, which may do so only if $Y[i] = true$. By (I26), this implies that $q@{21..31} \wedge q.n = i$ is false. Therefore, statement $3.p$ cannot falsify (I19).

The only other statements that may falsify (I19) are $18.q$ (which may change the value of $q.n$) and $20.q$ (which establishes $q@{21..31}$). However, statement $18.q$ establishes $q@{19}$, and hence cannot falsify (I19). Statement $20.q$ may falsify (I19) only if executed when $q.n = i$ holds. In this case, it falsifies $X[i] = p$, which implies that $A(p, i) \wedge p@{4..6}$ is false as well. \square

$$\text{invariant } A(p, i) \wedge q.n = i \Rightarrow \neg(p@{7, 8} \wedge q@{23..31}) \quad (\text{I20})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$, which establishes $p@{4, 11}$. Hence, it cannot falsify (I20).

The only other statements that may falsify (I20) are $18.q$ (which may change the value of $q.n$), $6.p$ (which may establish $p@{7, 8}$), and $22.q$ (which establishes $q@{23..31}$). However, statement $18.q$ establishes $q@{19}$, and hence cannot falsify (I20).

Statement $6.p$ may falsify (I20) only by establishing $p@{7}$, which it does only if $X[p.node] = p$. By (I5), this implies that $p@{6} \wedge X[p.node] = p \wedge Reset[p.node] = p.y$ holds. Thus, $6.p$ 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 $6.p$. If $i = p$, then $A(p, i)$ holds before $6.p$ is executed, which implies that $q@{23..31} \wedge q.n = i$ is false, by (I19). Thus, statement $6.p$ cannot falsify (I20).

Statement $22.q$ may falsify (I20) only if executed when $q.n = i \wedge p@{7, 8}$ holds. By (I29), this implies that $p.y.free = true$ holds. Because statement $22.q$ establishes $Reset[i].free = false$, this implies that $p@{7, 8} \wedge Reset[i] \neq p.y$ holds, i.e., $A(p, i)$ is false. \square

$$\text{invariant } A(p, i) \wedge q.n = i \Rightarrow \neg(p@{9, 11..34} \wedge q@{24..31}) \quad (\text{I21})$$

Proof: By Lemma A1, the only statement that can establish $A(p, i)$ is $3.p$. However, if $3.p$ establishes $A(p, i)$, it also establishes $p@{4}$, and hence it cannot falsify (I21).

The only other statements that may falsify (I21) are $18.q$ (which may change the value of $q.n$), $3.p$, $6.p$, $7.p$, $8.p$, and $10.p$ (which may establish $p@{9, 11..34}$), and $23.q$ (which may establish $q@{24..31}$). However, statement $18.q$ establishes $q@{19}$, and hence it cannot falsify (I21).

Statements $3.p$, $6.p$, $7.p$, and $10.p$ can establish $p@{9, 11..34}$ only by establishing $p@{11}$, in which case they also establish $p.dir \neq stop$. This implies that $A(p, i)$ is false. Statement $8.p$ can neither establish nor falsify $A(p, i)$. Thus, it may falsify (I21) only if executed when $A(p, i) \wedge q.n = i \wedge q@{24..31}$ holds, but this is precluded by (I20).

Statement $23.q$ may falsify (I21) only if it is executed when $q@{23} \wedge A(p, i) \wedge p@{9, 11..34} \wedge q.n = i \wedge (q.j = q.level \vee Round[q.y.rnd] = false)$ holds. By (I61), $p@{17..34}$ and $q@{23}$ cannot hold simultaneously. So, assume that the following assertion holds prior to the execution of $23.q$.

$$q@{23} \wedge A(p, i) \wedge p@{9, 11..16} \wedge q.n = i \wedge (q.j = q.level \vee Round[q.y.rnd] = false) \quad (2)$$

If $q.j = q.level$ holds, then by (I35), $q.n = q.node$. Because $q.n = i$, this implies that $A(q, i)$ holds. By (2), this implies that $A(p, i) \wedge A(q, i) \wedge p@{9, 11..16} \wedge q@{23}$ holds. However, this is precluded by (I18).

The only other possibility is that

$$q@{23} \wedge A(p, i) \wedge p@{9, 11..16} \wedge q.n = i \wedge Round[q.y.rnd] = false$$

holds before $23.q$ is executed. In this case, $Reset[q.n].rnd = q.y.rnd$ holds, by (I25), and does $Reset[i].rnd = p.y.rnd$, by the definition of $A(p, i)$ and (I3). Thus, $q.y.rnd = p.y.rnd$. In addition, by (I2), $Round[p.y.rnd] = true$. Thus, $Round[q.y.rnd] = true$, which is a contradiction. \square

$$\text{invariant } A(p, i) \wedge (p@{7..9, 11..17} \vee (p@{18..31} \wedge p.j = p.level)) \wedge q@{7..10} \wedge p \neq q \Rightarrow p.y.rnd \neq q.y.rnd \quad (\text{I22})$$

Proof: The only statements that may falsify the consequent are $3.p$ and $21.p$ (which may change the value of $p.y.rnd$) and $3.q$ and $21.q$ (which may change the value of $q.y.rnd$). However, the antecedent is false after the execution of $3.q$ and $21.q$. Also, $3.p$ either establishes $p@{4}$ or $p@{10} \wedge p.dir = right$. The latter implies that $A(p, i)$ is false. Thus, statement $3.p$ cannot falsify (I22).

Statement $21.p$ may falsify (I22) only if executed when $A(p, i) \wedge p.j = p.level$ holds. In this case, $Reset[i].rnd = p.y.rnd$ holds, by (I3), and $p.n = i$ holds, by (I35). It follows that statement $21.p$ does not change the value of $p.y.rnd$ if executed when the antecedent holds.

The antecedent may be established only by statements $3.p$ (which, by Lemma A1, may establish $A(p, i)$ and may also establish $p@{7..9, 11..17}$), $6.p$ and $10.p$ (which may establish $p@{7..9, 11..17}$), and $6.q$ (which may establish $q@{7..10}$). However, as shown above, statement $3.p$ cannot falsify (I22).

Statement $6.q$ can establish $q@{7}$ only if $X[q.node] = q$ holds. If $i = q.node$, then this implies that $A(q, i)$ holds. However, by (I17) (with p and q exchanged), this implies that the antecedent of (I22) is false. Thus, $6.q$ cannot falsify (I22) in this case. If $i \neq q.node$, then we have $q.y = Reset[q.node]$, by the

definition of $A(q, q.node)$, and $Reset[q.node].rnd \neq Reset[i].rnd$, by (I15). If the antecedent of (I22) holds, then either $A(p, i) \wedge p@{7..9, 11..17} \vee (p@{18..30} \wedge p.j = p.level)$ holds or $p@{31}$ holds. In the former case, by (I3) and the definition of $A(p, i)$, $p.y.rnd = Reset[i].rnd$ holds. In the latter case, by (I13), $p.y.rnd \neq Reset[q.node].rnd$. Thus, in either case, $p.y.rnd \neq q.y.rnd$. Hence, statement $6.q$ cannot falsify (I22).

Statements $6.p$ and $10.p$ are the remaining statements to consider. Statement $10.p$ establishes $p@{11} \wedge p.dir = left$, which implies that $A(p, i)$ is false. Statement $6.p$ establishes either $p@{11} \wedge p.dir = left$ or $p@{7}$. In the former case, $A(p, i)$ is false. In the latter case, note that statement $6.p$ may establish $A(p, i) \wedge p@{7}$ only if executed when $X[i] = p$ holds. By (I5) and (I29), this implies that

$$p.y = Reset[i] \wedge Reset[i].free = true. \quad (3)$$

In addition, if $6.p$ is executed when the antecedent of (I22) holds, then by (I6), (I7), and (I8), either $q.y.rnd = Reset[q.node].rnd$ holds or $Dist[q.y.rnd] \neq \perp$ holds. This gives us two cases to analyze.

- $q.y.rnd = Reset[q.node].rnd$. Note that statement $6.p$ can falsify (I22) only if executed when $p.y.rnd = q.y.rnd \wedge q@{7..10}$ holds. By (3), this implies that $Reset[i].rnd = Reset[q.node].rnd$. By (I15), this in turn implies that $i = q.node$.

If $q@{7..9}$ holds before the execution of $6.p$, then by (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 (I29), we have $q.y.free = true$. By (3), this implies that $p.y.rnd \neq q.y.rnd$, which is a contradiction.

On the other hand, if $q@{10}$ holds before the execution of $6.p$, then we have $Reset[q.node].free = false$ by (I10). Because $i = q.node$, this contradicts the second conjunct of (3).

- $Dist[q.y.rnd] \neq \perp$. By Lemma A1, statement $6.p$ may establish the antecedent only if $A(p, i)$ holds before its execution. By (I19), this implies that $\neg(\exists r :: r@{29, 30} \wedge r.n = i)$ holds. Hence, by (I14), $Dist[Reset[i].rnd] = \perp$ holds as well. By (3), this implies that $Dist[p.y.rnd] = \perp$ holds. Because $Dist[p.y.rnd] = \perp$ and $Dist[q.y.rnd] \neq \perp$ both hold, $p.y.rnd \neq q.y.rnd$. It follows that statement $6.p$ cannot falsify (I22). \square

$$\text{invariant } p@{3..10} \Rightarrow y.dir = stop \quad (I23)$$

$$\text{invariant } p@{22} \Rightarrow Reset[p.n] = p.y \quad (I24)$$

$$\text{invariant } p@{23..30} \Rightarrow Reset[p.n] = (false, p.y.rnd) \quad (I25)$$

$$\text{invariant } p@{20..31} \Rightarrow Y[p.n] = (false, 0) \quad (I26)$$

$$\text{invariant } Y[i].free = true \Rightarrow Y[i] = Reset[i] \quad (I27)$$

$$\text{invariant } Reset[i].rnd \neq 0 \quad (I28)$$

$$\text{invariant } p@{4..10} \Rightarrow p.y.free = true \quad (I29)$$

$$\text{invariant } p@{4..10, 22..33} \Rightarrow p.y.rnd \neq 0 \quad (I30)$$

Proof Sketch: By (I61), all writes to *Reset* (statements 22 and 30) and to *Y*, except for statement 4 (statements 19, 31), and all operations involving the *Free* queue (statements 26, 28, and 29) occur within mutually exclusive regions of code. Given this and the initial condition $(\forall i, p :: Y[i] = (true, i) \wedge Reset[i] = (true, i)) \wedge Free = (T + 1) \rightarrow \dots \rightarrow S$, each of these invariants easily follows. Note that statement 4 establishes $Y[i] = (false, 0)$, and hence cannot falsify either (I26) or (I27). Note also that (I26) implies that statements 22 and 30 cannot falsify (I27). \square

$$\text{invariant } (p@{6..17} \vee (p@{18..33} \wedge p.j = p.level)) \wedge (p@{11} \Rightarrow p.dir \neq right) \Rightarrow Inuse[p] = p.y.rnd \wedge p.y.rnd \neq 0 \quad (I31)$$

Proof: The first conjunct of the antecedent may be established only by statements $3.p$ and $5.p$. Statement $3.p$ can establish this conjunct only by establishing $p@{11}$, in which case it also establishes $p.dir = right$. This implies that the second conjunct of the antecedent is false. By (I30), statement $5.p$ establishes the consequent. The second conjunct of the antecedent may be established only by statements $2.p$, $6.p$, $7.p$,

and $10.p$ (which establish $p.dir \neq right$), and statement $11.p$ (which falsifies $p@\{11\}$). However, statement $2.p$ establishes $p@\{3\}$, which implies that the antecedent is false. Because (I31) is assumed to hold before the execution of $6.p$, $7.p$, and $10.p$, $Inuse[p] = p.y.rnd \wedge p.y.rnd \neq 0$ holds both before and after each of these statements is executed. Likewise, if $p.dir \neq right$ holds before $11.p$ is executed, then $Inuse[p] = p.y.rnd \wedge p.y.rnd \neq 0$ holds afterward. If $p.dir = right$ holds before $11.p$ is executed, then $11.p$ establishes $p@\{2\} \vee (p@\{15\} \wedge p.level > L)$. Because $p.j$ ranges over $\{0, \dots, L\}$, this implies that the antecedent of (I31) is false.

The only statements that may falsify the consequent are $3.p$ and $21.p$ (which update $p.y$) and $5.p$ and $33.p$ (which update $Inuse[p]$). As shown above, statement $3.p$ cannot falsify (I31). If the antecedent of (I31) holds when statement $21.p$ is executed, then by (I35), $p.n = p.node$ holds, and hence by (I3), $21.p$ does not change the value of $p.y.rnd$. By (I30), the consequent of (I31) holds after the execution of statement $5.p$. Statement $33.p$ establishes either $p@\{18\}$ or $p@\{34, 37\}$. If it establishes $p@\{18\}$, then $p.j < p.level$ holds after its execution. Thus, $33.p$ falsifies the antecedent of (I31). \square

$$\text{invariant } Round[i] = true \Rightarrow (\exists p :: p.y.rnd = i \wedge (p@\{9..17\} \vee (p@\{18..32\} \wedge p.j = p.level))) \wedge (p@\{11\} \Rightarrow (p.dir = stop \wedge p.level \leq L)) \quad (\text{I32})$$

Proof: The antecedent may be established only by statement $8.p$, which also establishes the consequent.

Suppose that

$$p.y.rnd = i \wedge (p@\{9..17\} \vee (p@\{18..32\} \wedge p.j = p.level)) \wedge (p@\{11..17\} \Rightarrow (p.dir = stop \wedge p.level \leq L)) \quad (4)$$

holds. Because this assertion implies that $p@\{9..32\}$ holds, it can be falsified only by the following statements: $9.p$ (which might establish $p@\{11\} \wedge p.dir \neq stop$), $10.p$ (which establishes $p@\{11\} \wedge p.dir = left$), $11.p$ (which may falsify $p@\{9..32\}$ and $p.level \leq L$), and $17.p$, $23.p$, $31.p$, and $32.p$ (which may falsify $p@\{18..32\} \wedge p.j = p.level$), and $21.p$ (which updates $p.y$). Of these statements, only $21.p$ could possibly falsify $p.y.rnd = i$.

If $p.y.rnd = i$, then statements $10.p$ and $32.p$ falsify the antecedent of (I32). If executed when (4) holds, then $11.p$ establishes $p@\{12\} \wedge p.dir = stop \wedge p.level \leq L$, by (I33); $17.p$ establishes $p@\{18\} \wedge p.j = p.level$; $23.p$ establishes $p@\{24\} \wedge p.j = p.level$; and $31.p$ establishes $p@\{32\} \wedge p.j = p.level$. Moreover, if $21.p$ is executed when (4) holds, then by (I3) and (I35), it does not change the value of $p.y.rnd$. \square

$$\text{invariant } p@\{2..11\} \Rightarrow (0 \leq p.level \leq L) \wedge (1 \leq p.node \leq T) \quad (\text{I33})$$

$$\text{invariant } p@\{19..33\} \Rightarrow 1 \leq p.n \leq T \wedge p.n = p.path[p.j].node \quad (\text{I34})$$

$$\text{invariant } p@\{19..33\} \wedge p.j = p.level \Rightarrow p.n = p.node \quad (\text{I35})$$

$$\text{invariant } p@\{2..38\} \Rightarrow (0 \leq p.level \leq L + 1) \wedge (lev(p.node) = p.level) \quad (\text{I36})$$

$$\text{invariant } p@\{32, 33\} \Rightarrow p.j = p.level \quad (\text{I37})$$

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

$$\text{invariant } p@\{12..38\} \wedge (p.node \leq T) \Rightarrow p.path[p.level] = (p.node, stop) \wedge p.dir = stop \quad (\text{I38})$$

Proof: The antecedent of (I38) can only be established by statement $11.p$, which does so only if $p.level \leq L \vee p.dir = stop$ holds prior to its execution. If $p.dir = stop$ holds, then $11.p$ establishes the consequent of (I38). If $p.level \leq L$ holds before $11.p$ is executed, then by (I36) $p.node > T$ holds afterward. No statement can falsify the consequent of (I38) while the antecedent holds. \square

$$\text{invariant } p@\{2..38\} \wedge (2i \xrightarrow{*} p.node) \Rightarrow p.path[lev(i)] = (i, left) \quad (\text{I39})$$

Proof: The only statements that may falsify (I39) are $1.p$ and $11.p$. Statement $1.p$ establishes $p.node = 1$, which implies that $2i \xrightarrow{*} p.node$ is false. Statement $11.p$ 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 (I36), statement $11.p$ establishes the consequent. \square

$$\text{invariant } (p@\{11\} \wedge p.dir = stop) \vee p@\{12..38\} \Rightarrow 1 = p.path[0] \wedge p.path[p.level - 1] \xrightarrow{*} p.node \wedge (\forall l : 0 \leq l < p.level - 1 :: p.path[l] \xrightarrow{*} p.path[l + 1]) \quad (\text{I40})$$

$$\text{invariant } p@\{11\} \wedge p.dir \neq stop \wedge p.level \geq L \Rightarrow 1 = p.path[0] \wedge p.path[p.level] \xrightarrow{*} 2 \cdot p.node \wedge (\forall l : 0 \leq l < p.level :: p.path[l] \xrightarrow{*} p.path[l + 1]) \quad (\text{I41})$$

$$\text{invariant } p@\{19..31\} \Rightarrow (p.n = p.node) \vee (2 \cdot p.n \xrightarrow{*} p.node) \quad (\text{I42})$$

Proof Sketch: Each iteration of the **repeat** loop of statements 2-11 descends one level in the renaming tree, starting with splitter 1 (the root). When descending from level l , $p.path[l]$ is updated (statement 11) to indicate the splitter visited at level l . From this, invariant (I40) easily follows.

The **repeat** loop of statements 2-11 terminates only if $p.level \geq L \vee p.dir = stop$ holds prior to the execution of statement $11.p$. If $p.dir = stop$ holds when $11.p$ is executed, then when p executes within statements $12.p$ through $38.p$, $p.node$ equals the node at which it stopped. If $p.level \geq L$ holds when $11.p$ is executed, then when p executes within statements $12.p$ through $38.p$, $p.node$ equals a node at level $L + 1$ (in which case there is no actual splitter corresponding to $p.node$). In either case, the **for** loop at statements 18-33 will ascend the renaming tree, visiting only ancestors of $p.node$ for which p either stopped or moved left while descending the renaming tree. The corresponding node is indicated by the variable $p.n$. If p stopped at that node, then $p.n = p.node$ holds. If p moved left from that node, then $2 \cdot p.n \xrightarrow{*} p.node$ holds. From these observations, it should be clear that (I42) is an invariant. \square

$$\text{invariant } \neg(\exists p :: p@\{29\}) \Rightarrow \|Free\| = 2N \quad (\text{I43})$$

$$\text{invariant } (\exists p :: p@\{29\}) \Rightarrow \|Free\| = 2N + 1 \quad (\text{I44})$$

$$\text{invariant } p@\{25..27\} \Rightarrow p.ptr = Check \quad (\text{I45})$$

$$\text{invariant } p@\{30, 31\} \Rightarrow \text{Dist}[p.nextrnd] = \perp \quad (\text{I46})$$

$$\text{invariant } p@\{29\} \Rightarrow \text{Dist}[p.y.rnd] = 2N \quad (\text{I47})$$

$$\text{invariant } p@\{30\} \Rightarrow \text{Dist}[p.y.rnd] = 2N - 1 \quad (\text{I48})$$

Proof Sketch: Note that every statement that accesses the *Free* queue (statements 26, 28, and 29) executes within a mutually exclusive region of code. From this and the initial condition, $\|Free\| = 2N$, these invariants easily follow. Note also that by (I25), if $p@\{28\}$ holds, then $p.y.rnd = Reset[p.n]$ holds. By (I14) and (I61), this in turn implies that $\text{Dist}[p.y.rnd] = \perp$ holds. Hence, statement $28.p$ does not enqueue a duplicate entry onto the *Free* queue. \square

$$\text{invariant } p@\{0, 1\} \Rightarrow \text{Loc}[p] = 1 \wedge \text{Loc}[p + N] = 1 \quad (\text{I49})$$

$$\text{invariant } p@\{2\} \Rightarrow \text{Loc}[p] = p.node \wedge \text{Loc}[p + N] = p.node \quad (\text{I50})$$

$$\text{invariant } p@\{3\} \Rightarrow (X[p.node] = p \Rightarrow \text{Loc}[p] = p.node \wedge \text{Loc}[p + N] = p.node) \wedge (X[p.node] \neq p \Rightarrow \text{Loc}[p] = 2 \cdot p.node \wedge \text{Loc}[p + N] = 2 \cdot p.node) \quad (\text{I51})$$

$$\text{invariant } p@\{4..10\} \Rightarrow (A(p, p.node) \Rightarrow \text{Loc}[p] = p.node \wedge \text{Loc}[p + N] = p.node) \wedge (\neg A(p, p.node) \Rightarrow \text{Loc}[p] = 2 \cdot p.node \wedge \text{Loc}[p + N] = 2 \cdot p.node) \quad (\text{I52})$$

$$\text{invariant } p@\{11\} \Rightarrow (p.dir = stop \Rightarrow \text{Loc}[p] = p.node \wedge \text{Loc}[p + N] = p.node) \wedge (p.dir = left \Rightarrow \text{Loc}[p] = 2 \cdot p.node \wedge \text{Loc}[p + N] = 2 \cdot p.node) \wedge (p.dir = right \Rightarrow \text{Loc}[p] = 2 \cdot p.node + 1 \wedge \text{Loc}[p + N] = 2 \cdot p.node + 1) \quad (\text{I53})$$

$$\text{invariant } p@\{12..38\} \Rightarrow \text{Loc}[p + N] = p.node \quad (\text{I54})$$

$$\text{invariant } p@\{12..17\} \Rightarrow \text{Loc}[p] = p.node \quad (\text{I55})$$

$$\text{invariant } p@\{18\} \Rightarrow p.path[p.j].node \xrightarrow{*} \text{Loc}[p] \quad (\text{I56})$$

$$\text{invariant } p@\{19..33\} \Rightarrow \text{Loc}[p] = p.n \quad (\text{I57})$$

Proof Sketch: These invariants easily follow from the program text and the structure of the renaming tree. Note that $p@\{3\}$ is established only if $X[p.node] = p$ holds. Also note that whenever $X[p.node] = p$ is falsified

by either 2. q or 23. q , where q is any arbitrary process, q also establishes $\text{Loc}[p] = 2 \cdot p.\text{node} \wedge \text{Loc}[p + N] = 2 \cdot p.\text{node}$.

Statement 3. p may establish $p@4$ only if executed when $Y[p.\text{node}].\text{free} = \text{true}$ holds. By (I27), this implies that $Y[p.\text{node}] = \text{Reset}[p.\text{node}]$ holds. From this condition, the consequent of (I51), and the definition of $A(p, i)$, it follows that if statement 3. p establishes $p@4$, then the consequent of (I52) is true.

$A(p, i)$ potentially could be falsified some process $q \neq p$ only by executing one of the statements 2. q , 22. q , or 30. q . However, by (I25), $q@30$ implies that $\text{Reset}[q.n].\text{free} = \text{false}$ holds. Moreover, by (I29), $p@4.9 \Rightarrow p.y.\text{free} = \text{true}$. It follows that statement 30. q cannot change the value of $A(p, i)$ from true to false. Each of the statements 2. q and 22. p assigns the value $2 \cdot p.\text{node}$ to each of $\text{Loc}[p]$ and $\text{Loc}[p + N]$. \square

$$\begin{aligned}
\text{invariant } Y[i].\text{free} = \text{false} \Rightarrow \mathcal{A}: & (\exists p :: (p@5..17 \vee (p@18..31 \wedge p.j = p.\text{level})) \wedge \\
& (p.\text{node} = i) \wedge (p@11 \Rightarrow p.\text{dir} \neq \text{right})) \vee \\
\mathcal{B}: & (\exists p :: p@2..33 \wedge (2i \xrightarrow{*} p.\text{node}) \wedge \\
& (p@18..33 \Rightarrow 2i \xrightarrow{*} p.\text{path}[p.j].\text{node})) \vee \\
\mathcal{C}: & (\exists p :: p@18..31 \wedge p.\text{path}[p.j].\text{node} = i \wedge \\
& (p@18 \Rightarrow 2i \xrightarrow{*} \text{Loc}[p]))
\end{aligned} \tag{I58}$$

Proof: The only statements that can establish the antecedent are 4. q and 19. q , where q is any arbitrary process. Statement 4. q establishes disjunct \mathcal{A} . If statement 19. q establishes the antecedent, then by (I34), 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.

$$(p@5..17 \vee (p@18..31 \wedge p.j = p.\text{level})) \wedge (p.\text{node} = i) \wedge (p@11 \Rightarrow p.\text{dir} \neq \text{right}) \tag{5}$$

This assertion implies that $p@5..31$ holds. Thus, it may be falsified only by the following statements: 6. p , 7. p , and 9. p (which may establish $p@11 \wedge p.\text{dir} = \text{right}$), 11. p (which updates $p.\text{node}$ and $p.\text{level}$ and may falsify $p@5..17$), 17. p (which falsifies $p@5..17$ and may falsify $p@18..31 \wedge p.j = p.\text{level}$), and 23. p and 31. p (which may falsify $p@18..31 \wedge p.j = p.\text{level}$).

By (I23), 6. p , 7. p , and 9. p cannot establish $p@11 \wedge p.\text{dir} = \text{right}$. If executed when (5) holds, statement 11. p establishes either $p@2, 12 \wedge p.\text{node} = 2i$ or $p@12 \wedge p.\text{node} = i$ (depending on the values of $p.\text{dir}$ and $p.\text{level}$). The former establishes disjunct \mathcal{B} , while the latter preserves (5).

Because $Y[i]$ is defined only for $1 \leq i \leq T$, if (5) holds, then $p.\text{node} \leq T$. Thus, if $p@17$ holds, then by (I36), $p.\text{level} < L$. Moreover, by (I38), $p.\text{dir} = \text{stop}$. Thus, if (5) holds when 17. p is executed, it establishes $p@18 \wedge p.j = p.\text{level}$.

If executed when (5) holds, statement 23. p establishes $p@24 \wedge p.j = p.\text{level}$. By (I35), statement 31. p falsifies the antecedent.

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

$$p@2..33 \wedge (2i \xrightarrow{*} p.\text{node}) \wedge (p@18..33 \Rightarrow 2i \xrightarrow{*} p.\text{path}[p.j].\text{node}) \tag{6}$$

Because this assertion implies that $p@2..33$ holds, it can be falsified only by statements 11. p (which may update $p.\text{node}$ and $p.\text{path}$), and 17. p , 23. p , 31. p , and 33. p (which may establish $p@18..33$, falsify $p@2..33$, or modify $p.j$). However, if statement 11. p is executed when (6) holds, then it cannot falsify $2i \xrightarrow{*} p.\text{node}$.

By (I39), (6) implies that $p.\text{path}[\text{lev}(i)] = (i, \text{left})$ holds — informally, p moved left from splitter i when descending the renaming tree. Moreover, because $i \leq T$, we have $\text{lev}(i) \leq L$. It follows that statement 17. p establishes $p@18 \wedge p.j = l$, where $\text{lev}(l) \geq \text{lev}(i)$. If 17. p establishes $p@18 \wedge p.j = l$, where $\text{lev}(l) > \text{lev}(i)$, then because $p.\text{path}[\text{lev}(i)] = (i, \text{left})$ holds, splitter l must be a descendent of splitter $2i$ (note that l could equal $2i$). Thus, in this case, 17. p establishes $p@18 \wedge 2i \xrightarrow{*} p.\text{path}[p.j].\text{node}$, i.e., (6) is not falsified. On

the other hand, if 17. p establishes $p@18 \wedge p.j = l$, where $l = \text{lev}(i)$, then $p@18 \wedge p.\text{path}[p.j].\text{node} = i$ holds. Moreover, because $2i \xrightarrow{*} p.\text{node}$ (from (6)), by (I55), $2i \xrightarrow{*} \text{Loc}[p]$. This implies that disjunct \mathcal{C} holds.

Similar reasoning can be applied to show that if 23. p , 31. p , or 33. p is executed when (6) holds, then the **for** loop must iterate again, and either $p@18 \wedge 2i \xrightarrow{*} p.\text{path}[p.j].\text{node}$ or $p@18 \wedge p.\text{path}[p.j].\text{node} = i \wedge 2i \xrightarrow{*} \text{Loc}[p]$ is established. (Informally, the **for** loop iterates again because, by (6) and (I34), the splitter indicated by $p.n$ is a descendent of the left child of i . Because process p moved left at splitter i , the loop cannot terminate.)

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

$$p@18..31 \wedge p.\text{path}[p.j].\text{node} = i \wedge (p@18 \Rightarrow 2i \xrightarrow{*} \text{Loc}[p]) \quad (7)$$

This assertion implies that $p@18..31$ holds. Thus, it may be falsified only by statements 23. p and 31. p (which may falsify $p@18..31$, establish $p@18$, or modify $p.j$). By (I34), statement 31. p falsifies the antecedent of (I58).

Statement 23. p may falsify (7) only if executed when $\text{Round}[p.y.\text{rnd}] = \text{true}$. So, assume that $p@23 \wedge \text{Round}[p.y.\text{rnd}] = \text{true}$ holds. Then, by (I32), there exists a process q such that

$$(q@9..17 \vee (q@18..32 \wedge q.j = q.\text{level})) \wedge (q@11 \Rightarrow q.\text{dir} = \text{stop}) \wedge q.y.\text{rnd} = p.y.\text{rnd}. \quad (8)$$

If $q = p$, then this assertion implies that $p.j = p.\text{level}$ holds, in which case statement 23. p establishes $p@24$, preserving (7).

So, suppose that $q \neq p$. By (I25), (I34), and (7), we have $\text{Reset}[i].\text{rnd} = p.y.\text{rnd}$, which by (8) implies that

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

In addition, by (I61) and our assumption that $p@23$ holds, we have $q@9..16$. We now prove that $\text{Reset}[q.\text{node}].\text{rnd} = q.y.\text{rnd}$ holds by considering the two cases $q@9,10$ and $p@11..16$ separately. First, suppose that $p@9,10$ holds. Then, by (I8) and (I61), either $\text{Reset}[q.\text{node}].\text{rnd} = q.y.\text{rnd}$ or $\text{Dist}[q.y.\text{rnd}] \neq \perp$ holds. However, because $\text{Reset}[i].\text{rnd} = q.y.\text{rnd}$, by (I14) and (I61), this implies that $\text{Dist}[q.y.\text{rnd}] = \perp$ holds. Thus, in this case, we have $\text{Reset}[q.\text{node}].\text{rnd} = q.y.\text{rnd}$.

On the other hand, if $q@11..16$ holds, then by (8), $A(q, q.\text{node})$ holds as well. By (I3), this implies that $\text{Reset}[q.\text{node}].\text{rnd} = q.y.\text{rnd}$ holds.

Putting these assertions together, we have $\text{Reset}[i].\text{rnd} = \text{Reset}[q.\text{node}].\text{rnd}$. By (I15), this implies that $q.\text{node} = i$. Therefore, if $q \neq p$, we have $q@9..16 \wedge q.\text{node} = i$, which implies that disjunct \mathcal{A} holds both before and after the execution of 23. p . \square

The following lemma is used in proving invariants (I59) and (I60).

Lemma A2: If t and u are consecutive states such that $p@2..11$ holds at t , the condition $i \xrightarrow{*} \text{Loc}[p]$ 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 $\text{Loc}[p]$ is changed by a call to $\text{UpdateLoc}(P, i)$ such that $p \in P$.
- $i = \text{Loc}[p] \wedge C(\lfloor i/2 \rfloor) > C(i)$ holds at u .

Proof: The only statements that may establish $i \xrightarrow{*} \text{Loc}[p]$ while $p@2..11$ holds are 3. p , 6. p , 7. p , 2. q , 20. q , and 22. q , where q is any arbitrary process. It should be obvious that these statements can update $\text{Loc}[p]$ only by calling $\text{UpdateLoc}(P, i)$, where $p \in P$. Now we show that if state u is reached via the execution of any of these statements, then $i = \text{Loc}[p]$ is established.

By using (I51) and (I52), we can tabulate all the possible ways in which $\text{Loc}[p]$ can be changed by one of these statements. Such a tabulation is given below. For example, by (I51), if $p@3$ holds, then $\text{Loc}[p]$ is

either $p.node$ or $2 \cdot p.node$. In both cases, statement $3.p$ establishes $\text{Loc}[p] = 2 \cdot p.node + 1$. Note that the table only shows ways in which $\text{Loc}[p]$ may *change* value. For instance, by (I52), if $p@\{6\}$ holds, then $\text{Loc}[p]$ is either $p.node$ or $2 \cdot p.node$. Because $6.p$ can change the value of $\text{Loc}[p]$ only by establishing $\text{Loc}[p] = 2 \cdot p.node$, we do not include an entry for $\text{Loc}[p] = 2 \cdot p.node$ in the column for state t .

| statement | at state t | at state u |
|-------------|--|--|
| $3.p$ | $\text{Loc}[p] = p.node$ $\text{Loc}[p] = 2 \cdot p.node$ | $\text{Loc}[p] = 2 \cdot p.node + 1$ $\text{Loc}[p] = 2 \cdot p.node + 1$ |
| $6.p, 7.p$ | $\text{Loc}[p] = p.node$ | $\text{Loc}[p] = 2 \cdot p.node$ |
| $2.q, 20.q$ | $\text{Loc}[p] = p.node$ | $\text{Loc}[p] = 2 \cdot p.node$ |
| $22.q$ | $\text{Loc}[p] = p.node$ | $\text{Loc}[p] = 2 \cdot p.node$ |

From this table, we see that there are only three ways in which the value of $\text{Loc}[p]$ 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 $i \xrightarrow{*} \text{Loc}[p]$ holds at u but not at t if and only if $i = \text{Loc}[p]$ 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 $i \xrightarrow{*} \text{Loc}[p]$ does not hold at t , splitter i is not the root, i.e., its parent does exist.) Note that in each of cases **(i)** through **(iii)**

$$h = p.node \wedge (i = 2 \cdot p.node \vee i = 2 \cdot p.node + 1). \quad (9)$$

We now show that $C(\lfloor i/2 \rfloor) > C(i)$ holds at u by considering each of statements $3.p, 6.p, 7.p, 2.q, 20.q,$ and $22.q$ separately. Statement $3.p$ may establish $\text{Loc}[p] = i$, where $i = 2 \cdot p.node + 1$ only if executed when $Y[h].free = false$, in which case the antecedent of (I58) holds. Thus, one of the three disjuncts of the consequent of (I58) holds.

- If disjunct \mathcal{A} holds, then by (I52)–(I54), there exists q such that $q@\{5..31\} \wedge (\text{Loc}[q + N] = h \vee \text{Loc}[q + N] = 2h)$.
- If disjunct \mathcal{B} holds, then by (I50)–(I54), there exists q such that $q@\{2..33\} \wedge 2h \xrightarrow{*} \text{Loc}[q + N]$.
- If disjunct \mathcal{C} holds, then by (I34) and (I57), there exists q such that $q@\{18..31\} \wedge (2h \xrightarrow{*} \text{Loc}[q] \vee \text{Loc}[q] = q.n = h)$.

If $q = p$, then the condition $p@\{3\}$ implies that disjunct \mathcal{B} must hold. By (9) and (I51), this implies that $2h = 2 \cdot p.node \xrightarrow{*} q.node$ holds, which is a contradiction. Hence, $q \neq p$, and thus statement $3.p$ does not change the value of $\text{Loc}[q]$. Because one of \mathcal{A} through \mathcal{C} holds before $3.p$ is executed, the following assertion holds both before and after the execution of $3.p$.

$$q \neq p \wedge \text{Loc}[q] = h \vee \text{Loc}[q + N] = h \vee 2h \xrightarrow{*} \text{Loc}[q] \vee 2h \xrightarrow{*} \text{Loc}[q + N]$$

(Informally, this assertion states that either $\text{Loc}[q]$ or $\text{Loc}[q + N]$ remains in h , or the left subtree of h . Note that, because $i = 2 \cdot p.node + 1$, by (9), i is the right child of h .) This assertion implies that $C(h) > C(i)$.

Statement $6.p$ may establish $\text{Loc}[p] = i$, where $i = 2h$, only if executed when $X[h] \neq p$ holds. However, this implies that $A(p, h)$ is false, and hence $\text{Loc}[p] = 2h$, by (I52). Therefore, statement $6.p$ cannot change the value of $\text{Loc}[p]$.

Statement $7.p$ may establish $\text{Loc}[p] = i$, where $i = 2h$, only if executed when $Acquired[h] = true$. By (I4), this implies that there exists a process q such that $q@\{13, 14, 17..36\} \wedge A(q, h)$ holds. Thus, by (I54), $\text{Loc}[q + N] = h$ holds both before and after the execution of $7.p$. This implies that $C(h) > C(i)$.

Statement $2.q$ may establish $\text{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{3..6\} \wedge p.node = q.node = h$ holds. In this case, by (I50), $\text{Loc}[q] = h$ holds at t . Because statement $2.q$ does not update $\text{Loc}[q]$, this condition also holds at u , which implies that $C(h) > C(i)$.

Similarly, statements $20.q$ and $22.q$ may establish $\text{Loc}[p] = i$, where $i = 2h$, only if executed when $p@\{3..9\} \wedge p.\text{node} = q.n = h$ holds. In this case, by (I57), $\text{Loc}[q] = h$ holds at t , and hence at u . This implies that $C(h) > C(i)$. \square

$$\text{invariant } p@\{2..11\} \wedge (i \xrightarrow{*} \text{Loc}[p]) \Rightarrow \text{PC}[p, i] \geq C(i) \quad (\text{I59})$$

Proof: The only statement that can establish $p@\{2..11\}$ is $1.p$. But this establishes $\text{Loc}[p] = 1$ and $\text{PC}[p, 1] = C(1)$. Hence the consequent is established (or preserved) for $i = 1$, and the antecedent is falsified for $i \neq 1$.

If $p@\{2..11\}$ holds, then the only statements that can establish $i \xrightarrow{*} \text{Loc}[p]$ are $3.p$, $6.p$, $7.p$, $2.q$, $20.q$, and $22.q$, where q is any arbitrary process. By Lemma A2, if one of these statements establishes $i \xrightarrow{*} \text{Loc}[p]$, it establishes it by calling $\text{UpdateLoc}(P, i)$ such that $p \in P$. In this case, line u2 of UpdateLoc establishes $\text{PC}[p, i] = C(i)$.

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

The value of $C(i)$ may be changed only by statements $18.q$, $36.q$, and $38.q$ (by updating $\text{Loc}[q]$ or $\text{Loc}[q+N]$ directly), or a call to procedure UpdateLoc . However, by (I56), statement $18.q$ always changes $\text{Loc}[q]$ from a splitter to its ancestor. It follows that statement $18.q$ cannot cause $C(i)$ to increase for any i . Similarly, statements $36.q$ and $38.q$ establishes $\text{Loc}[q] = 0 \wedge \text{Loc}[q+N] = 0$, 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 (I59) is always preserved in this case. \square

$$\text{invariant } p@\{2..11\} \wedge (h \xrightarrow{*} i \xrightarrow{*} \text{Loc}[p]) \wedge i \neq h \Rightarrow \text{PC}[p, i] < \text{PC}[p, h] \quad (\text{I60})$$

Proof: It is enough to consider the case when h is immediate parent of i , i.e., when $h = \lfloor i/2 \rfloor$. The general case follows by inducting over $\text{lev}(i) - \text{lev}(h)$.

The only statement that can establish $p@\{2..11\}$ is $1.p$, but this establishes $\text{Loc}[p] = 1$, and hence falsifies the antecedent.

The only other statements that may establish the antecedent (by changing the value of $\text{Loc}[p]$) are $3.p$, $6.p$, $7.p$, $18.p$, $36.p$, $38.p$, $2.q$, $20.q$, and $22.q$, where q is any arbitrary process. Statements $18.p$, $36.p$, $38.p$ falsify the antecedent. The other statements may establish the antecedent only by calling UpdateLoc when $p@\{2..11\}$ holds.

Similarly, the only statements that may falsify the consequent (by changing the value of $\text{PC}[p, i]$ or $\text{PC}[p, h]$) when $p@\{2..11\}$ holds are $1.q$, $2.q$, $3.q$, $6.q$, $7.q$, $20.q$, and $22.q$, where q is any arbitrary process. These statements may falsify (I60) only by calling UpdateLoc when $p@\{2..11\}$ holds.

Therefore, it is enough to show that each call to $\text{UpdateLoc}(P, f)$ always preserves (I60). We consider two cases.

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

By Lemma A2, $i \xrightarrow{*} \text{Loc}[p]$ can be established only if $p \in P \wedge f = i$, and in this case $i = \text{Loc}[p] \wedge C(\lfloor i/2 \rfloor) > C(i)$ is also established. In this case, line u2 of UpdateLoc establishes $\text{PC}[p, i] = C(i)$, and line u3 ensures that $\text{PC}[p, h] \geq C(h)$ holds after the call to UpdateLoc . Because $h = \lfloor i/2 \rfloor$, this implies that $\text{PC}[p, h] = \text{PC}[p, i]$ holds after the call to UpdateLoc .

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

In this case, the antecedent of (I60) holds before the call to UpdateLoc , and hence $\text{PC}[p, h] > \text{PC}[p, i]$ holds as well. If the value of $\text{PC}[p, h]$ is changed by line u2 of UpdateLoc , then $p \in P$ and $f = h$. But this

implies that line u1 establishes $\text{Loc}[p] = h$, which falsifies the antecedent of (I60). (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.)

Therefore, we can assume that the value of $\text{PC}[p, h]$ is not changed by line u2. Because line u3 never causes a PC entry to decrease, and $\text{PC}[p, h] > \text{PC}[p, i]$ holds before `UpdateLoc` is called, this condition can be falsified only if the value of $\text{PC}[p, i]$ is increased.

But whenever $\text{PC}[p, i]$ is changed, either by line u2 or line u3, $\text{PC}[p, i] = C(i)$ is established. From (I59), it follows that the value of $\text{PC}[p, i]$ can increase only if $C(i)$ is also increases. By the definition of $C(i)$, this can happen only if `UpdateLoc`(P, f) establishes either $i \xrightarrow{*} \text{Loc}[r]$ or $i \xrightarrow{*} \text{Loc}[r + N]$ for some $r \in P$. In this case, either `UpdateLoc`(P, f) is called by statement 1. r , or $r \in \{2..11\}$ holds before the execution of the statement calling `UpdateLoc`(P, f). But statement 1. r calls `UpdateLoc`($\{r\}, 1$), and hence cannot establish $h \xrightarrow{*} i \xrightarrow{*} \text{Loc}[r] \wedge h \neq i$ for any pair of h and i . Therefore, $r \in \{2..11\}$ holds before `UpdateLoc` is called. By (I50)–(I53), this implies that $\text{Loc}[r] = \text{Loc}[r + N]$ holds. Therefore, it is enough to consider the case in which $i \xrightarrow{*} \text{Loc}[r]$ is established.

By Lemma A2, if `UpdateLoc`(P, f) establishes $i \xrightarrow{*} \text{Loc}[r]$, then it also establishes $C(\lfloor i/2 \rfloor) > C(i)$, i.e., $C(h) > C(i)$. Moreover, line u3 ensures that $\text{PC}[p, h] \geq C(h)$ holds after the call to `UpdateLoc`. Thus, arguing as in Case 1, we again have $\text{PC}[p, h] > \text{PC}[p, i]$. \square

$$\text{invariant (Mutual Exclusion)} \quad |\{p :: p \in \{17..34, 37\}\}| \leq 1 \quad (\text{I61})$$

Proof: From the specification of `ENTRY` and `EXIT` routines, (I61) could be falsified only if a process p executes statement 13. p while $(\exists q : q \neq p :: q \in \{13..35\} \wedge p.\text{node} = q.\text{node})$ holds. However, this is precluded by (I18). \square

$$\text{invariant (Contention Sensitivity)} \quad p \in \{12..17\} \Rightarrow \text{lev}(p.\text{node}) < \text{PC}[p, 1] \quad (\text{I62})$$

Proof: The antecedent is established only by statement 11. p . By (I40), (I41), and (I53), 11. p also establishes

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

Therefore, by (I59) and (I60), the following is established as well.

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

Given the length of this sequence, we have $\text{PC}[p, 1] \geq \text{PC}[p, p.\text{node}] + p.\text{level}$. Thus, by (I36), $\text{PC}[p, 1] \geq \text{PC}[p, p.\text{node}] + \text{lev}(p.\text{node})$ holds after 11. p is executed. Note that $p \in \{12\}$ implies that $\text{PC}[p, p.\text{node}] \geq 1$ (the point contention for some process at a splitter must at least include that process). Hence, if 11. p establishes the antecedent of (I62), then $\text{PC}[p, 1] > \text{lev}(p.\text{node})$ holds after its execution.

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

It should be clear that the number of remote memory references executed by a process p to enter and then exit its critical section is $\Theta(n)$, where n is the value of $\text{lev}(p.\text{node})$ when p reaches statement 12. By (I62), n is at most *twice* the point contention experienced by p while executing within statements 1 through 11. (Recall that in updating PC values, we sometimes count a process q as *two* processes, q and $q + N$.) Thus, p executes $\Theta(k)$ remote memory references to enter and then exit its critical section, where k is the point contention it experiences in its entry section.