8-1-2005

Efficiently Implementing a Large Number of LL/SC Objects

Prasad Jayanti
Dartmouth College

Srdjan Petrovic
Dartmouth College

Dartmouth Computer Science Technical Report TR2005-554
Efficiently Implementing a Large Number of LL/SC Objects

Prasad Jayanti and Srdjan Petrovic
Department of Computer Science
Dartmouth College
Hanover, NH 03755
spetrovic,prasad@cs.dartmouth.edu

Abstract

Over the past decade, a pair of instructions called load-linked (LL) and store-conditional (SC) have emerged as the most suitable synchronization instructions for the design of lock-free algorithms. However, current architectures do not support these instructions; instead, they support either CAS (e.g., UltraSPARC, Itanium) or restricted versions of LL/SC (e.g., POWER4, MIPS, Alpha). Thus, there is a gap between what algorithm designers want (namely, LL/SC) and what multiprocessors actually support (namely, CAS or RLL/RSC). To bridge this gap, a flurry of algorithms that implement LL/SC from CAS have appeared in the literature. The two most recent algorithms are due to Doherty, Herlihy, Luchangco, and Moir (2004) and Michael (2004). To implement M LL/SC objects shared by N processes, Doherty et al.'s algorithm uses only $O(N+M)$ space, but is only non-blocking and not wait-free. Michael's algorithm, on the other hand, is wait-free, but uses $O(N^2+M)$ space. The main drawback of his algorithm is the time complexity of the SC operation: although the *expected amortized* running time of SC is only $O(1)$, the *worst-case* running time of SC is $O(N^2)$. The algorithm in this paper overcomes this drawback. Specifically, we design a wait-free algorithm that achieves a space complexity of $O(N^2 + M)$, while still maintaining the $O(1)$ worst-case running time for LL and SC operations.

Keywords: wait-free algorithms, LL/SC, CAS, ABA problem.

1 Introduction

In shared-memory multiprocessors, multiple processes running concurrently on different processors cooperate with each other via shared data structures (e.g., queues, stacks, counters, heaps, trees). Atomicity of these shared data structures has traditionally been ensured through the use of locks. To perform an operation, a process obtains the lock, updates the data structure, and then releases the lock. Locking, however, has several drawbacks, including deadlocks, priority inversion, convoying, and lack of fault-tolerance to process crashes.

Wait-free implementations were conceived to overcome the above drawbacks of locking [11, 19, 24]. A wait-free implementation guarantees that every process completes its operation on the data structure in a bounded number of its steps, regardless of whether other processes are slow, fast, or have crashed. A weaker form of implementation, known as *non-blocking* implementation [19], guarantees that if a process p repeatedly takes steps, then the operation of *some* process (not necessarily p) will eventually complete. Thus, non-blocking implementations guarantee that the system as a whole makes progress, but admit starvation of individual processes. An even weaker form of implementation, known as *obstruction-free* implementation [9], guarantees that a process completes its operation on the data structure, provided that it eventually executes for a sufficient number of steps without interference from other processes. This progress condition therefore allows for a situation where all processes starve.

It is a well understood fact that whether lock-free algorithms (i.e., wait-free, non-blocking, or obstruction-free) can be efficiently designed depends crucially on what synchronization instructions are available for the task. After more than two decades of experience with different instructions, there is growing consensus among algorithm designers on the desirability of a pair of instructions known as *Load-Link* (LL) and *Store-Conditional* (SC). The LL and SC instructions act like read and conditional-write, respectively. More specifically, the LL instruction by process p returns the value of the memory word, and the SC(v) instruction by p writes v if and only if no process updated the memory word since p's latest LL. A more precise formulation of these instructions is presented in Figure 1.

Despite the desirability of LL/SC, no processor supports these instructions in hardware; instead, they support either *compare&swap*, also known as CAS (e.g., UltraSPARC [13], Itanium [5]) or restricted versions of LL/SC (e.g., POWER4 [7], MIPS [26], Alpha [25]). Although the restrictions on LL/SC vary from one architecture to another, Moir [23] noted that the LL/SC instructions supported by current architectures, henceforth referred to as RLL/RSC, satisfy at a minimum the semantics stated in Figure 3.

- LL($\mathcal{O}$) returns $\mathcal{O}$'s value.
- SC($\mathcal{O}$, v) by process p "succeeds" if and only if no process performed a successful SC on $\mathcal{O}$ since p's latest LL on $\mathcal{O}$. If SC succeeds, it changes $\mathcal{O}$'s value to v and returns *true*. Otherwise, $\mathcal{O}$'s value remains unchanged and SC returns *false*.
- VL($\mathcal{O}$) returns *true* if and only if no process performed a successful SC on $\mathcal{O}$ since p's latest LL on $\mathcal{O}$.

Figure 1: Definitions of operations LL, SC, and VL

- CAS(X, u, v) behaves as follows: if X's current value is u, X is assigned v and *true* is returned; otherwise, X is unchanged and *false* is returned.

Figure 2: Definition of the CAS operation

Since CAS suffers from the well-known ABA problem [3] and RLL/RSC impose severe restrictions on their use [23], it is difficult to design algorithms based on these instructions. Thus, there is a gap between what algorithm designers want (namely, LL/SC) and what multiprocessors actually support (namely, CAS or RLL/RSC). This gap must be bridged efficiently, which gives rise to the following problem:

Design an algorithm that implements LL/SC objects from memory words supporting either CAS or RLL/RSC operations. To be useful in practice, the time and space complexities must be kept small.

The above problem has been extensively studied in the literature [1, 2, 6, 14, 18, 17, 20, 22, 23]. The most efficient algorithm for implementing LL/SC from CAS is due to Moir [23]. His algorithm runs in constant time and has no space overhead. However, it can only implement small (e.g., 24 to 32 bit) LL/SC objects, which are inadequate for storing pointers, large integers and doubles. This size limitation is due to the fact that Moir's algorithm stores a sequence number along with the object's value in the same memory word. Since sequence number could take up to 32 to 40 bits, only 24 to 32 bits are left for the value field.

- `RLL/RSC` are similar to *LL* and *SC*, with two differences [23]: (i) there is a chance of *RSC* failing spuriously: *RSC* might fail even when *SC* would succeed, and (ii) a process must not access any shared variable between its *RLL* and the subsequent *RSC*.

Figure 3: Definition of operations RLL/RSC

Elsewhere, we presented an algorithm that implements a *word-sized* LL/SC object from a word-sized CAS object and registers (e.g., 64-bit LL/SC on a 64-bit machine) [17]. This algorithm stores a value and a sequence number in separate memory words, thus enabling values to be as big as 64 bits. The algorithm implements both LL and SC in $O(1)$ time and uses $O(N)$ space, where N is the maximum number of processes that the algorithm is designed to handle. Although these space requirements are modest when a single LL/SC object is implemented, the algorithm does not scale well when the number of LL/SC objects to be supported is large. In particular, in order to implement M LL/SC objects, the algorithm requires $O(NM)$ space. Furthermore, the algorithm requires that N is known in advance.

The recent algorithms by Doherty, Herlihy, Luchangco, and Moir [6] and Michael [22] have aimed to overcome the above two drawbacks. Doherty et al.'s algorithm [6] uses only $O(N+M)$ space and does not require knowledge of N, but is only non-blocking and not wait-free. Michael's [22] algorithm, on the other hand, is wait-free and does not require knowledge of N, but uses $O(N^2+M)$ space. The main drawback of this algorithm is the time complexity of the SC operation: although the *expected amortized* running time of SC is only $O(1)$, the *worst-case* running time of SC is $O(N^2)$. The algorithm in this paper overcomes this drawback, as described below.

We design a wait-free algorithm that achieves a space complexity of $O(N^2+M)$, while still maintaining the $O(1)$ worst-case running time for LL and SC operations. This algorithm too does not require knowledge of N. When constructing a large number of LL/SC objects (i.e., when $M=\omega(N)$), our implementation is the first to be simultaneously (1) wait-free, (2) time optimal, and (3) space efficient. Specifically, the algorithm by Doherty et al. [6], although more space efficient than ours, is not wait-free. Michael's algorithm [22] has the same space complexity as ours and is wait-free, but is not time optimal. Other algorithms are either not space efficient [1, 2, 14, 18, 17, 23], not wait-free [20], or implement small LL/SC objects [2, 14, 23].

We note that the algorithm in this paper, as well as the algorithms by Doherty et al. [6] and Michael [22], implement (the more general) *multiword* LL/SC object, i.e., an LL/SC object whose value spans across multiple machine words (e.g., 512- or 1024-bit LL/SC object). Many existing applications [1, 4, 16, 15] require support for such an object. When implementing a W-word LL/SC object, the time and space complexities increase by a factor of W, which is also the case with the algorithms of [6] and [22]. Specifically, the space complexity of our algorithm becomes $O((N^2+M)W)$, and the time complexity of LL and SC becomes $O(W)$.

Elsewhere, we presented an algorithm that implements M W-word LL/SC objects using $O(NMW)$ space [18]. This algorithm employs a helping scheme by which processes help each other complete their LL operations. We use a similar helping scheme in the present paper.

1.1 Related work

The quality of an LL/SC algorithm can be judged by several criteria: (1) the maximum size of the object that the algorithm is capable of implementing (e.g., small, word-sized, or multiword), (2) the strength of the progress condition that the algorithm satisfies (obstruction-free, non-blocking, or wait-free), (3) whether the algorithm requires explicit knowledge of N, and (4) the time and space complexities of the algorithm. With these criteria in mind, we present a comparison of related work in Table 1. We used the following notation: M is the number of implemented LL/SC objects, and N is the number of processes sharing those objects.

1.2 Correctness condition

The correctness condition that we use in the paper is *linearizability* [12]. Since this correctness condition is well known, we only describe it informally here.

A *shared object is linearizable* if operations applied to the object appear to act instantaneously, even though in reality each operation executes over an interval of time. More precisely, every operation applied to the object appears to take effect at some instant between its invocation and completion [12]. This instant (at which an operation appears to take effect) is called the *linearization point* for that operation. Our algorithms ensure that the implemented object $\mathcal{O}$ is linearizable whenever the primitive objects from which $\mathcal{O}$ is implemented are linearizable.

Algorithm	*Size of LL/SC*	*Progress Condition*	*Worst-case Time Complexity*		*Space Complexity*	*Knowledge of N*
			LL	*SC*		
1. This paper	W-word	wait-free	$O(W)$	$O(W)$	$O((N^2+M)W)$	not required
2. Israeli and Rappoport [14]	small	wait-free	$O(N)$	$O(N)$	$O(N^2+NM)$	required
3. Anderson and Moir [2], Figure 1	small	wait-free	$O(1)$	$O(1)$	$O(N^2M)$	required
4. Anderson and Moir [1], Figure 2	W-word	wait-free	$O(W)$	$O(W)$	$O(N^2MW)$	required
5. Moir [23], Figure 4	small	wait-free	$O(1)$	$O(1)$	$O(N+M)$	not required
6. Moir [23], Figure 7	small	wait-free	$O(1)$	$O(1)$	$O(N^2+NM)$	required
7. Luchangco et al. [20]†	63-bit	obstruction-free	–	–	$O(N+M)$	required
8. Jayanti and Petrovic [17]	64-bit	wait-free	$O(1)$	$O(1)$	$O(NM)$	required
9. Doherty et al. [6]	W-word	non-blocking	–	–	$O((N+M)W)$	not required
10. Michael [22]	W-word	wait-free	$O(W)$	$O(N^2+W)$‡	$O((N^2+M)W)$	not required
11. Jayanti and Petrovic [18]	W-word	wait-free	$O(W)$	$O(W)$	$O(NMW)$	required

Table 1: A comparison of algorithms that implement LL/SC from CAS or RLL/RSC.

1.3 Organization for the rest of the paper

We present our main result in two steps. First, we design an algorithm that implements an array of M LL/SC objects shared by N processes, where N is known in advance. Building on this algorithm, we present a more general algorithm that works without the knowledge of N. These two algorithms are described in Sections 2 and 3.

2 LL/SC for a known N

Figure 4 presents an algorithm that implements an array $\mathcal{O}[0\,..\,M-1]$ of M W-word LL/SC object shared by N processes. To make the presentation easier to follow, the algorithm is shown for the case when each process has at most one outstanding LL operation. Later, we explain how the algorithm can be trivially modified to handle any number of outstanding LL operations. We provide below an intuitive description of the algorithm.

2.1 The variables used

We begin by describing the variables used in the algorithm. `BUF`$[0\,..\,M+(N+1)N-1]$ is an array of $M+(N+1)N$ buffers. Of these, M buffers hold the current values of objects $\mathcal{O}[0], \mathcal{O}[1], \ldots, \mathcal{O}[M-1]$, while the remaining $(N+1)N$ buffers are "owned" by processes, $N+1$ buffer by each process. Each process p, however, uses only one of its $N+1$ buffers at any given time. The index of the buffer that p is currently using is stored in the local variable $mybuf_p$, and the indices of the remaining N buffers are stored in p's local queue Q_p. Array `X`$[0\,..\,M-1]$ holds the tags associated with the current values of objects $\mathcal{O}[0], \mathcal{O}[1], \ldots, \mathcal{O}[M-1]$. A tag in `X`$[i]$ consists of two fields: (1) the index of the buffer that holds $\mathcal{O}[i]$'s current value, and (2) the sequence number associated with $\mathcal{O}[i]$'s current value. The sequence number increases by 1 with each successful SC on $\mathcal{O}[i]$, and the buffer holding $\mathcal{O}[i]$'s current value is not reused until some process performs at least N more successful SC's (on any $\mathcal{O}[j]$). Process p's local variable x_p maintains the tag corresponding to the value returned by p's most recent LL operation; p will use this tag during the subsequent SC operation to check whether the object still holds the same value (i.e., whether it has been modified). Finally, it turns out that a process p might need the help of other processes in completing its LL operation on $\mathcal{O}$. The shared variables `Help`$[p]$ and `Announce`$[p]$, as well as p's local variables $lseq_p$ and $index_p$, are used to facilitate this helping scheme. Additionally, an extra word is kept in each buffer along with the value. Hence, all the buffers in the algorithm are of length $W+1$.[1]

2.2 The helping mechanism

The crux of our algorithm lies in its helping mechanism by which SC operations help LL operations. This helping mechanism is similar to that of [18], but whereas the mechanism of [18] requires $O(NMW)$ space, the mechanism in this paper requires only $O((N^2+M)W)$ space. Below, we describe this mechanism in detail.[2]

A process p begins its LL operation on some object $\mathcal{O}[i]$ by announcing its operation to other processes. It then attempts to read the buffer containing $\mathcal{O}[i]$'s current value. This reading has two possible outcomes: either

†This algorithm implements a weaker form of LL/SC in which an LL operation by a process can cause some other process's SC operation to fail.

‡The *expected amortized* running time for SC is $O(W)$.

[1] The only exception are the buffers passed as an argument to procedures LL and SC, which are of length W.

[2] Some of the text to follow has been taken directly from [18].

Types

valuetype = **array** $[0\,..\,W]$ **of** 64-bit value

xtype = **record** *seq*: $(64 - \lg(M + (N+1)N))$-bit number; *buf*: $0\,..\,M + (N+1)N - 1$ **end**

helptype = **record** *seq*: $(63 - \lg(M + (N+1)N))$-bit number; *helpme*: $\{0, 1\}$; *buf*: $0\,..\,M + (N+1)N - 1$ **end**

Shared variables

X: **array** $[0\,..\,M-1]$ of xtype; Announce: **array** $[0\,..\,N-1]$ **of** $0\,..\,M-1$; Help: **array** $[0\,..\,N-1]$ **of** helptype

BUF: **array** $[0\,..\,M + (N+1)N - 1]$ **of** $*$valuetype

Local persistent variables at each $p \in \{0, 1, \ldots, N-1\}$

$mybuf_p$: $0\,..\,M + (N+1)N - 1$; Q_p: Single-process queue; x_p: xtype;

$lseq_p$: $(63 - \lg(M + (N+1)N))$-bit number; $index_p$: $0\,..\,N-1$

Initialization

$\texttt{X}[k] = (0, k)$, for all $k \in \{0, 1, \ldots, M-1\}$

$\texttt{BUF}[k]$ = the desired initial value of $\mathcal{O}[k]$, for all $k \in \{0, 1, \ldots, M-1\}$

For all $p \in \{0, 1, \ldots, N-1\}$

enqueue$(Q_p, M + (N+1)p + k)$, for all $k \in \{0, 1, \ldots, N-1\}$

$mybuf_p = M + (N+1)p + N$; $\texttt{Help}[p] = (0, 0, *)$; $index_p = 0$; $lseq_p = 0$

procedure LL$(p, i, retval)$

1: Announce$[p] = i$
2: Help$[p] = (++lseq_p, 1, mybuf_p)$
3: $x_p = \texttt{X}[i]$
4: **copy** $*\texttt{BUF}[x_p.buf]$ **into** $*retval$
5: **if** $\neg\texttt{CAS}(\texttt{Help}[p], (lseq_p, 1, mybuf_p), (lseq_p, 0, mybuf_p))$
6: $\quad mybuf_p = \texttt{Help}[p].buf$
7: $\quad x_p = \texttt{BUF}[mybuf_p][W]$
8: $\quad$ **copy** $*\texttt{BUF}[mybuf_p]$ **into** $*retval$
9: **return**

procedure VL(p, i) **returns** boolean

10: **return** $(\texttt{X}[i] == x_p)$

procedure SC(p, i, v) **returns** boolean

11: **copy** $*v$ **into** $*\texttt{BUF}[mybuf_p]$
12: **if** $\neg\texttt{CAS}(\texttt{X}[i], x_p, (x_p.seq + 1, mybuf_p))$
13: $\quad$ **return** *false*
14: *enqueue*$(Q_p, x_p.buf)$
15: $mybuf_p = dequeue(Q_p)$
16: **if** $(\texttt{Help}[index_p] \equiv (s, 1, b))$
17: $\quad j = \texttt{Announce}[index_p]$
18: $\quad x = \texttt{X}[j]$
19: $\quad$ **copy** $*\texttt{BUF}[x.buf]$ **into** $*\texttt{BUF}[mybuf_p]$
20: $\quad \texttt{BUF}[mybuf_p][W] = x$
21: $\quad$ **if** $\texttt{CAS}(\texttt{Help}[index_p], (s, 1, b), (s, 0, mybuf_p))$
22: $\quad\quad mybuf_p = b$
23: $index_p = (index_p + 1) \bmod N$
24: **return** *true*

Figure 4: Implementation of $\mathcal{O}[0\,..\,M-1]$: an array of M N-process W-word LL/SC objects

p correctly obtains the value in the buffer or p obtains an inconsistent value because the buffer is overwritten while p reads it. In the latter case, the key property of our algorithm is that p is helped (and informed that it is helped) before the completion of its reading of the buffer. Thus, in either case, p has a valid value: either p reads a valid value in the buffer (former case) or it is handed a valid value by a helper process (latter case). The implementation of such a helping scheme is sketched in the following paragraph.

Consider any process p that performs a successful SC operation. During that SC, p checks whether a single process—say, q—has an ongoing LL operation that requires help. If so, p helps q by passing it a valid value and a tag associated with that value. (We will see later how p obtains that value.) If several processes try to help, only one will succeed. Process p makes a decision on which process to help by consulting its variable $index_p$: if $index_p$ holds value j, then p helps process j. The algorithm ensures that $index_p$ is incremented by 1 modulo N after every successful SC operation by p. Hence, during the course of N successful SC operations, process p examines all N processes for possible help. Recall the earlier stated property that the buffer holding an $\mathcal{O}[i]$'s current value is not reused until some process performs at least N successful SC's (on any $\mathcal{O}[j]$). As a consequence of the above facts, if a process q begins reading the buffer that holds $\mathcal{O}[i]$'s current value and the buffer happens to be reused while q still reads it (because some process p has since performed N successful SC's), then p is sure to have helped q by handing it a valid value of $\mathcal{O}[i]$ and a tag associated with that value.

2.3 The roles of Help$[p]$ and Announce$[p]$

The variables Help$[p]$ and Announce$[p]$ play an important role in the helping scheme. Help$[p]$ has three fields: (1) a binary value (that indicates if p needs help), (2) a buffer index, and (3) a sequence number (independent from the sequence numbers in tags).

Announce[p] has only one field: an index in the range $0..M-1$. When p initiates an LL operation on some object $\mathcal{O}[i]$, it first announces the index of that object by writing i into Announce[p] (see Line 1), and then seeks the help of other processes by writing $(s, 1, b)$ into Help[p], where b is the index of the buffer that p owns (see Line 2) and s is p's local sequence number incremented by one. If a process q helps p, it does so handing over its buffer c containing a valid value of $\mathcal{O}[i]$ to p by writing $(s, 0, c)$. (This writing is performed with a CAS operation to ensure that at most one process succeeds in helping p.) Once q writes $(s, 0, c)$ in Help[p], p and q exchange the ownership of their buffers: p becomes the owner of the buffer indexed by c and q becomes the owner of the buffer indexed by b. (This buffer management scheme is the same as in Herlihy's universal construction [8].) Before q hands over buffer c to process p, it also writes a tag associated with that value into the Wth location of the buffer.

2.4 How the helper obtains a valid value

We now explain an important feature of our algorithm, namely, the mechanism by which a process p obtains a valid value to help some other process q with. Suppose that process p wishes to help process q complete its LL operation on some object $\mathcal{O}[i]$. To obtain a valid value to help q with, p first attempts to read the buffer containing $\mathcal{O}[i]$'s current value. This reading has two possible outcomes: either p correctly obtains the value in the buffer or p obtains an inconsistent value because the buffer is overwritten while p reads it. In the latter case, by an earlier stated property, p knows that there exists some process r that has performed at least N successful SC operations (on any $\mathcal{O}[j]$). Therefore, r must have already helped q, in which case p's attempt to help q will surely fail. Hence, it does not matter that p obtained an inconsistent value of $\mathcal{O}[i]$ because p will anyway fail in giving that value to q. As a result, if p helps q complete its LL operation on some object $\mathcal{O}[i]$, it does so with a valid value of $\mathcal{O}[i]$.

2.5 Code for LL

A process p performs an LL operation on some object $\mathcal{O}[i]$ by executing the procedure LL(p, i, *retval*), where *retval* is a pointer to a block of W-words in which to place the return value. First, p announces its operation to inform others that it needs their help (Lines 1 and 2). It then attempts to obtain the current value of $\mathcal{O}[i]$ by performing the following steps. First, p reads the tag stored in X[i] to determine the buffer holding $\mathcal{O}[i]$'s current value (Line 3), and then reads that buffer (Line 4). While p reads the buffer at Line 4, the value of $\mathcal{O}[i]$ might change because of successful SC's by other processes. Specifically, there are three possibilities for what happens while p executes Line 4: (i) no process performs a successful SC, (ii) no process performs more than $N-1$ successful SC's, or (iii) some process performs N or more successful SC's. In the first case, it is obvious that p reads a valid value at Line 4. Interestingly, in the second case too, the value read at Line 4 is a valid value. This is because, as remarked earlier, our algorithm does not reuse a buffer until some process performs at least N successful SC's. In the third case, p cannot rely on the value read at Line 4. However, by the helping mechanism described earlier, a helper process would have made available a valid value (and a tag associated with that value) in a buffer and written the index of that buffer in Help[p]. Thus, in each of the three cases, p has access to a valid value as well as a tag associated with that value. Further, as we now explain, p can also determine which of the three cases actually holds. To do this, p performs a CAS on Help[p] to try to revoke its request for help (Line 5). If p's CAS succeeds, it means that p has not been helped yet. Therefore, Case (i) or (ii) must hold, which implies that *retval* has a valid value of $\mathcal{O}$. Hence, p returns from the LL operation at Line 9.

If p's CAS on Help[p] fails (Line 5), p knows that it has been helped, and that a helper process must have written in Help[p] the index of a buffer containing a valid value U of $\mathcal{O}[i]$ (as well as a tag associated with U). So, p reads U and its associated tag (Lines 7 and 8), and takes ownership of the buffer it was helped with (Line 6). Finally, p returns from the LL operation at Line 9.

2.6 Code for SC

A process p performs an SC operation on some object $\mathcal{O}[i]$ by executing the procedure SC(p, i, v), where v is the pointer to a block of W-words which contain the value to write to $\mathcal{O}[i]$ if SC succeeds. First, p writes the value v into its local buffer (Line 11), and then tries to make its SC operation take effect by changing the value in X[i] from the tag it had witnessed in its latest LL operation to a new tag consisting of (1) the index of p's local buffer and (2) a sequence number (of the previous tag) incremented by one (Line 12). If the CAS operation fails, it follows that some other process performed a successful SC after p's latest LL, and hence p's SC must fail. Therefore, p terminates its SC procedure, returning *false* (Line 13). On the other hand, if CAS succeeds,

then p's current SC operation has taken effect. In that case, p gives up ownership of its local buffer, which now holds $\mathcal{O}[i]$'s current value, and becomes the owner of the buffer B holding $\mathcal{O}[i]$'s old value. To remain true to the promise that the buffer that held $\mathcal{O}[i]$'s current value (B, in this case) is not reused until some process performs at least N successful SC's, p enqueues the index of buffer B into its local queue (Line 14), and then dequeues some other buffer index from the queue (Line 15). Notice that, since p's local queue contains N buffer indices when p inserts B's index into it, p will not reuse buffer B until it performs at least N successful SC's.

Next, p tries to determine whether some process needs help with its LL operation. As we stated earlier, the process to help is $q = index_p$. So, p reads `Help`$[q]$ to check whether q needs help (Line 16). If it does, p consults variable `Announce`$[q]$ to learn the index j of the object that q needs help with (Line 17). Next, p reads the tag stored in `X`$[j]$ to determine the buffer holding $\mathcal{O}[j]$'s current value (Line 18), and then copies the value from that buffer into its own buffer (Line 19). Then, p writes into the Wth location of the buffer the tag that it read from `X`$[j]$ (Line 20). Finally, p attempts to help q by handing it p's buffer (Line 21). If p succeeds in helping q, then, by the earlier discussion, the buffer that p handed over to q contains a valid value of $\mathcal{O}[j]$. Hence, p gives up its buffer to q and assumes ownership of q's buffer (Line 22). (Notice that p's CAS at Line 21 fails if and only if, while p executed Lines 16–21, either another process already helped q or q withdrew its request for help.) Regardless of whether process q needed help or not, p increments the $index_p$ variable by 1 modulo N (Line 23) to ensure that in the next successful SC operation it helps some other process (Line 23), and then terminates its SC procedure by returning *true* (Line 24).

The procedure VL is self-explanatory (Line 10). The following theorem summarizes the above discussion. Its proof is presented in Appendix A.1.

Theorem 1 *The algorithm in Figure 4 is a linearizable, wait-free implementation of an array $\mathcal{O}[0\,..\,M-1]$ of W-word LL/SC objects, shared by N processes. The time complexities of LL, SC and VL operations on any $\mathcal{O}[i]$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The space complexity of the implementation is $O((N^2+M)W)$.*

2.7 Remarks

2.7.1 Sequence number wrap-around

Each 64-bit variable `X`$[i]$ stores in it a buffer index and an unbounded sequence number. The algorithm relies on the assumption that during the time interval when some process p executes one LL/SC pair, the sequence number stored in `X`$[i]$ does not cycle through all possible values. If we reserve 32 bits for the buffer index (which allows the implementation of up to 2^{31} LL/SC objects, shared by up to $2^{15} = 32,768$ processes), we still will have 32 bits for the sequence number, which is large enough that sequence number wraparound is not a concern in practice.

2.7.2 The number of outstanding LL operations

Modifying the code in Figure 4 to handle multiple outstanding LL/SC operations is straightforward. Simply require that each LL operation, in addition to returning a value, also returns the tag associated with that value. Then, when calling an SC operation on some object, the caller p must also provide the tag that was returned by p's latest LL operation on that object.

3 LL/SC for an unknown N

In this section, we present a modified version of the algorithm in Figure 4 that does not require N to be known in advance. In particular, the algorithm supports two new operations—*Join*(p) and *Leave*(p)—which allow a process p to join and leave the algorithm at any given time. If K is the maximum number of processes that have simultaneously participated in the algorithm (i.e., have joined the algorithm but have not yet left it), then the space complexity of the algorithm is $O((K^2+M)W)$. The time complexities of procedures Join, Leave, LL, SC, and VL are $O(K)$, $O(1)$, $O(W)$, $O(W)$, and $O(1)$, respectively.

The algorithm is given in three steps. First, we introduce an important building block of the algorithm, namely, an implementation of a dynamic array that supports constant-time read and write operations (with some restrictions). Next, we restate the LL/SC algorithm in Figure 4, but with small modifications that will make it easier to remove the assumption of N. Finally, we present our main result, namely, an algorithm that implements an array of M W-word LL/SC objects shared by an unknown number of processes. These three steps are described in Sections 3.1, 3.2, and 3.3.

3.1 Dynamic arrays

A dynamic array is just like a regular array except that it places no bounds on the highest location that can be written. In particular, a process can write into the ith location of the dynamic array for any natural number

i. At all times, the size of the array must stay proportional to the highest location written so far. Furthermore, all reads and writes in the array must complete in $O(1)$ time. In this paper, we consider only a weaker version of dynamic array that has the following restrictions: (1) all writes into the same location write the same value, (2) a write into a location i must precede a read on that location, and (3) a write into a location i must precede a write into location $i+1$. We capture the above restrictions in an object that we call a *DynamicArray* object. This object is formally defined as follows.

A DynamicArray object supports two operations: *write*(i, v) and *read*(i). The *write*(i, v) operation writes value v into the ith location of the array, while the *read*(i) operation returns the value stored in the ith location of the array. The following restrictions are placed on the usage of *read* and *write* operations:

- Before *write*$(k+1, *)$ is invoked, at least one *write*$(k, *)$ must complete.
- Before *read*(k) is invoked, at least one *write*$(k, *)$ must complete.
- If *write*(k, v) and *write*(k, v') are invoked, then $v = v'$.

We present in Figure 5 an algorithm that implements a DynamicArray object from a CAS object and registers. In the following, we first describe the main idea behind the algorithm, and then describe the algorithm in detail.

3.1.1 The main idea

The main idea of the algorithm is as follows. We maintain two (static) arrays at all times: array A of length k, and array B of length $2k$. (Initially, A is of length 1, and B is of length 2.) When a process writes a value into some array location j, for $j \geq k/2$, it writes that value into both $A[j]$ and $B[j]$. Additionally, it copies the array location $A[j-k/2]$ into $B[j-k/2]$. By this mechanism, when the array A fills up (i.e., when the location $A[k-1]$ is written), all the locations of array A have been copied into array B. Therefore, B contains the same values as A, and can hence be used in place of A. A new array of length $4k$ is then allocated (and used in place of B), and the algorithm proceeds the same way as before.

3.1.2 The algorithm

The algorithm is presented in Figure 5. Central to the algorithm is variable `D`, which stores a pointer to the block containing three fields: (1) a pointer to array A, (2) a pointer to array B, and (3) the length of array A.

Types
arraytype = **array of** 64-bit value
dtype = **record** *size*: 64-bit number; A, B: $*$arraytype **end**
Shared variables
`D`: $*$dtype
Initialization
`D` = `malloc(sizeof` dtype)
`D`$\rightarrow$*size* = 1;
`D`$\rightarrow A$ = `malloc`(1);
`D`$\rightarrow B$ = `malloc`(2);

procedure `write`$(p, i, v, \mathcal{D})$
1: d = `D`
2: **if** ($d\rightarrow$*size* $> i$)
3: $d\rightarrow A[i] = v$
4: $d\rightarrow B[i] = v$
5: $d\rightarrow B[i-c'(i)] = d\rightarrow A[i-c'(i)]$
6: **else** *newD* = `malloc(sizeof` dtype)
7: *newD*$\rightarrow A = d\rightarrow B$
8: *newD*$\rightarrow B$ = `malloc`$(4 * d\rightarrow$*size*$)$
9: *newD*$\rightarrow$*size* $= 2 * d\rightarrow$*size*
10: **if** $\neg$`CAS(D,` d, *newD*) `free(`*newD*$\rightarrow B$); `free(`*newD*$)$
11: `write`$(i, v, \mathcal{D})$

procedure `read`$(p, i, \mathcal{D})$ **returns** valuetype
12: d = `D`
13: **return** $d\rightarrow A[i]$

Figure 5: An implementation of a DynamicArray object. Function $c'(i)$ returns the largest power of 2 smaller or equal to i. (If $i = 0$, then $c'(i) = 0$.)

We now explain the `write`$(p, i, v, \mathcal{D})$ procedure that describes how a process p writes a value v into the ith location of DynamicArray $\mathcal{D}$. In the following, let $c'(i)$ denote the largest power of 2 smaller than or equal to i.[3] First, p reads `D` to obtain a pointer to the block containing arrays A and B (Line 1). Next, p checks whether the length of array A is greater than i (Line 2). If it is, then p writes v into $A[i]$ (Line 3). To help with the amortized copying of array A into B, p writes v into $B[i]$ (Line 4) and copies the location $A[i-c'(i)]$ into $B[i-c'(i)]$ (Line 5).

Notice that, by initialization, the lengths of A and B are powers of 2 at all times. Let k be the length of A when p tries to write v into $A[i]$. Then, if $i \geq k/2$, we have $c'(i) = k/2$ (since k is a power of 2). Hence, p copies the location $A[i-k/2]$ into $B[i-k/2]$, which is consistent with our main idea presented earlier. If $i < k/2$, then, by definition of $c'(i)$, we have $i - c'(i) \geq 0$. Hence, p copies some location $A[j]$ into $B[j]$, for $j \in \{0, 1, \ldots, k/2-1\}$, which causes no harm. Hence, by copying $A[i-c'(i)]$ into $B[i-c'(i)]$, p remains faithful

[3] If $i = 0$, then $c'(i) = 0$.

to the earlier idea of amortized copying of array A into array B.

If the length of array A is equal to i, then p knows that the array A has been filled up. Furthermore, by an earlier discussion, all the values in A have already been copied into B. So, p prepares a new block *newD* that will hold pointers to the new values for arrays A and B. Next, p sets *newD.A* to B (Line 7), *newD.B* to a newly allocated array twice the size of B (Line 8), and *newD.size* to the size of B (Line 9). Then, p attempts to swing the pointer in `D` from the block that p had witnessed at Line 1, to the new block *newD* (Line 10). If p's CAS is successful, then p has successfully installed the new block *newD* in `D`. Otherwise, some other process must have installed its own block into `D`, and so p frees up the memory occupied by the block *newD* (Line 10). In either case, the length of an array A in the new block is sure to be greater than i. So, p calls the `write` procedure again to complete installing value v into $\mathcal{D}$ (Line 11). Notice that, since the size of the new array A is strictly greater than i, p will not make another recursive call to `write`, thus ensuring a constant running time for the `write` operation.

The `read`$(p, i, \mathcal{D})$ procedure is very simple: a process p simply reads `D` to obtain a pointer to the block containing the most recent values of A and B (Line 12), and then returns the value stored in $A[i]$ (Line 13). Notice that, since we require that at least one *write*$(i, *)$ operation completes before *read*(i) starts, the length of the array A is at least $i+1$ when p reads $A[i]$. Furthermore, by the above discussion, location $A[i]$ contains the value written by *write*$(i, *)$. Therefore, p returns the correct value.

We now calculate the space complexity of the algorithm at some time t. First, notice that there are only two arrays at time $t = 0$: one array of length 1 and one of length 2. During the first *write*$(1, *)$ operation, a new array of length 4 is allocated. Similarly, during the first *write*$(2, *)$ operation, a new array of length 8 is allocated. In general, during the first *write*$(2^j, *)$ operation, a new array of length 2^{j+2} is allocated. So, if *write*$(K, *)$ is the operation with the highest index among all operations invoked prior to time t, then at time t the largest allocated array is of length $2^{\lfloor \lg K \rfloor + 2}$. Hence, the lengths of all allocated arrays at time t are $1, 2, 4, 8, \ldots, 2^{\lfloor \lg K \rfloor + 1}$, and $2^{\lfloor \lg K \rfloor + 2}$. Consequently, the space occupied by the arrays at time t is $2^{\lfloor \lg K \rfloor + 3} - 1$, and the space occupied by the blocks at time t is $\lfloor \lg K \rfloor + 1$. Therefore, the space permanently used by the algorithm at time t is $O(K)$. However, we also have to count the space occupied by the blocks and arrays allocated at Lines 6 and 8 that were not successfully installed in `D` but have not yet been freed from memory (at Line 10). The number of such blocks and arrays at time t is at most n, where n is the number of processes executing the algorithm at time t. Since the largest allocated array is of length at most $2^{\lfloor \lg K \rfloor + 2}$, the space used by the blocks and arrays at time t is $O(nK)$. Therefore, the space used by the algorithm at time t is $O(nK)$.

Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.2.

Theorem 2 *The algorithm in Figure 5 is wait-free and implements a DynamicArray object $\mathcal{D}$ from a word-sized CAS object and registers. The time complexity of read and write operations on $\mathcal{D}$ is $O(1)$. The space used by the algorithm at any time t is $O(nK)$, where n is the number of processes executing the algorithm at time t, and K is the highest location written in $\mathcal{D}$ prior to time t.*

3.2 Restatement of the algorithm in Figure 4

We now restate our known-N LL/SC algorithm from Figure 4. We introduce small modifications to this algorithm that will make it easier to remove the the assumption of N. Figure 6 shows the resulting algorithm. In the following, we refer to the algorithms in Figures 4 and 6 by the names A and B, respectively.

The main difference between algorithms A and B lies in the way the variables are organized. Below, we summarize the differences between the two organizations.

1. In algorithm A, process p's shared variables `Help`$[p]$ and `Announce`$[p]$ are located in shared arrays `Help` and `Announce`, respectively. Hence, if a process q wishes to access p's shared variables, it can do so by simply reading `Help`$[p]$ or `Announce`$[p]$. In algorithm B, on the other hand, process p's shared variables are stored in p's own block of memory, and an array `NameArray` holds pointers to memory blocks of all processes. Hence, if a process q wishes to access p's shared variables, it must first read `NameArray`$[p]$ to obtain the address l of p's memory block, and then read the variables $l \rightarrow$ `Help` and $l \rightarrow$ `Announce`.

2. In algorithm A, there is a single array `BUF` of length $M + N(N+1)$ which holds all the buffers used by the algorithm. The index of a buffer is therefore a number in the range $[0\,..\,M + N(N+1) - 1]$. Hence, if a process wishes to access a buffer with index b, it simply reads the location `BUF`$[b]$. In algorithm B, on the other hand, array `BUF` is divided into $N+1$

Types

valuetype = **array** $[0\,..\,W]$ **of** 64-bit value

indextype = **record** *type*: {0, 1}; **if** (*type* == 0) (*bindex*: 31-bit number)
else (*name*: 15-bit number; *bindex*: 15-bit number) **end**

xtype = **record** *seq*: 32-bit number; *buf*: indextype **end**

helptype = **record** *seq*: 31-bit number; *helpme*: {0, 1}; *buf*: indextype **end**

blocktype = **record** Announce: 31-bit number; Help: helptype; Q: Single-process queue; BUF: **array** $[0\,..\,N]$ **of** $*$valuetype;
index: 15-bit number; *mybuf*: indextype; *lseq*: 31-bit number; x: xtype **end**

Shared variables

X: **array** $[0\,..\,M-1]$ of xtype; BUF: **array** $[0\,..\,M-1]$ **of** $*$valuetype; NameArray: **array** $[0\,..\,N-1]$ **of** $*$blocktype

Local persistent variables at each p

loc_p: blocktype

Initialization

X$[k] = (0, k)$, for all $k \in \{0, 1, \ldots, M-1\}$

BUF$[k]$ = the desired initial value of $\mathcal{O}[k]$, for all $k \in \{0, 1, \ldots, M-1\}$

For all $p \in \{0, 1, \ldots, N-1\}$

NameArray$[p] = \&loc_p$; *enqueue*$(loc_p.Q, (1, p, k))$, for all $k \in \{0, 1, \ldots, N-1\}$

$loc_p.mybuf = (1, p, N)$; loc_p.Help $= (0, 0, *)$; $loc_p.index = 0$; $loc_p.lseq = 0$

procedure LL$(p, i, retval)$

1: loc_p.Announce $= i$
2: loc_p.Help $=$ (++$loc_p.lseq$, 1, $loc_p.mybuf$)
3: $loc_p.x =$ X$[i]$
4: **copy** $*$GetBuf$(loc_p.x.buf)$ **into** $*retval$
5: **if** ¬CAS(loc_p.Help, ($loc_p.lseq$, 1, $loc_p.mybuf$), ($loc_p.lseq$, 0, $loc_p.mybuf$))
6: $loc_p.mybuf = loc_p$.Help$.buf$
7: $b =$ GetBuf$(loc_p.mybuf)$
8: $loc_p.x = b[W]$
9: **copy** $*b$ **into** $*retval$
10: **return**

procedure GetBuf(b) **returns** $*$valuetype

11: **if** $(b.type == 0)$ **return** BUF$[b.bindex]$
12: $l =$ NameArray$[b.name]$
13: **return** $l\rightarrow$BUF$[b.bindex]$

procedure VL(p, i) **returns** boolean

14: **return** (X$[i] == loc_p.x$)

procedure SC(p, i, v) **returns** boolean

15: **copy** $*v$ **into** $*$GetBuf$(loc_p.mybuf)$
16: **if** ¬CAS(X$[i]$, $loc_p.x$, $(loc_p.x.seq + 1, loc_p.mybuf)$)
17: **return** *false*
18: *enqueue*$(loc_p.Q, loc_p.x.buf)$
19: $loc_p.mybuf =$ *dequeue*$(loc_p.Q)$
20: $l =$ NameArray$[loc_p.index]$
21: **if** $(l\rightarrow$Help $\equiv (s, 1, c))$
22: $j = l\rightarrow$Announce
23: $x =$ X$[j]$
24: $d =$ GetBuf$(loc_p.mybuf)$
25: **copy** $*$GetBuf$(x.buf)$ **into** $*d$
26: $d[W] = x$
27: **if** CAS($l\rightarrow$Help, $(s, 1, c)$, $(s, 0, loc_p.mybuf)$)
28: $loc_p.mybuf = c$
29: $loc_p.index = (loc_p.index + 1) \bmod N$
30: **return** *true*

Figure 6: A slightly modified version of the algorithm in Figure 4

smaller arrays: (1) a central array of length M, and (2) N arrays of length $N+1$ each, which are kept at processes' memory blocks (one array per process). The index of a buffer is therefore either a pair $(0, i)$, where i is the index into the central array, or a tuple $(1, p, i)$, where i is the index into the array located at process p's memory block. Hence, if a process wishes to access a buffer with index $b = (0, i)$, it simply reads the location BUF$[i]$. If, on the other hand, a process wishes to access a buffer with index $b = (1, p, i)$, it must first read NameArray$[p]$ to obtain the address l of p's memory block, and then read the location l.BUF$[i]$. The above method for accessing a buffer given its index is captured by the procedure GetBuf (see Figure 6).

As in algorithm A, we will need to store a sequence number and a buffer index together in a single machine word. From the previous paragraph, the buffer index consists of one bit (to distinguish between the central array and an array stored at a process's memory block) and $\lg(\max(M, N(N+1)))$ bits to describe either (1) an index within the central array, or (2) an index within a process's array and the name of that process. Assuming 64 bits per machine word, this leaves $64 - 1 - \lg(\max(M, N(N+1)))$ bits for the sequence number. Rather than using these long expressions, in the rest of the paper we assume the values 2^{31} and 2^{15} for M and N, respectively, which then leaves 32 bits for the sequence number.

3. In algorithm A, each process p maintains the follow-

ing persistent local variables: $mybuf_p$, $lseq_p$, $index_p$, x_p, and Q_p. In algorithm B, on the other hand, all of the above variables are located in p's memory block and p maintains the address of that memory block in its persistent local variable loc_p.

Given the above discussion, the code in Figure 6 is self-explanatory.

3.3 The unknown-N LL/SC algorithm

The algorithm is presented in Figure 7. The statements given in rectangular boxes represent the differences with the algorithm from Figure 6. Operations `da_read` and `da_write` denote read and write operations on the dynamic array. We now describe the changes that were made to the original algorithm.

Arrays `NameArray` and `BUF` (located at processes' memory blocks) are now dynamic arrays. Variable `N` holds the maximum number of processes that have simultaneously participated in the algorithm so far. Each process maintains its own estimate N of `N`, which it periodically updates to match `N`. At all times, the algorithm ensures that the length of process's local queue Q is at least N and the length of process's array `BUF` is at least $N + 1$. Processes' memory blocks are no longer allocated in advance. Instead, when a process joins the algorithm (by executing the Join procedure), it will either (1) allocate a new memory block, or (2) get a memory block from another process that has left the algorithm. In either case, the implementation of Join guarantees that each process p (participating in the algorithm) has a unique memory block. The algorithm also assigns (during the Join procedure) a unique name to each participating process. This name is guaranteed to be small: if K process are currently participating in the algorithm, then a new process joining the algorithm will be assigned a name in the range $[0\,..\,K]$. (Hence, a process' name is sure the be smaller or equal to `N`.) A process stores this name in a variable *name* located at that process's memory block. If a process p has a name n, then the nth location in (dynamic) array `NameArray` holds a pointer to the memory block owned by p. When p leaves the algorithm, it leaves its memory block in the nth location of `NameArray`; this block will be used later by another process that obtains the name n.

We now explain the code at Lines 19–23. After a process p inserts the index of the previous current buffer into its local queue (Line 18), it checks whether its estimate N matches the actual value `N` (Line 19). If it doesn't, then p increments N by one (Line 20). Next, p allocates a new buffer and writes that buffer into the Nth location of its array `BUF` (Line 21). By doing so, p implicitly increments the length of array `BUF` to $N+1$, thus maintaining the earlier stated invariant on the length of `BUF`. Then, p takes the newly allocated buffer and uses it as its own local buffer (Line 22). Notice that, in this case, p does not dequeue an index from its local queue; hence, p implicitly increases the length of its queue to N, thus maintaining the earlier stated invariant on the size of Q.

If, on the other hand, N does match the value of `N`, then p withdraws a new buffer index from its local queue and uses that buffer as its own local buffer (Line 23). The only other major change is at Line 33, where p increments its variable *index* by 1 modulo its local estimate N (versus a fixed N in the original algorithm). The changes at Lines 12, 13, and 24 are due to the fact that arrays `NameArray` and `BUF` (located at process' memory blocks) are now dynamic arrays.

Recall that in the original algorithm where N was fixed, each process p made a promise not to reuse the buffer B that held some $\mathcal{O}[i]$'s current value until p performed at least N successful SC operations. (Process p kept its promise by enqueueing B's index into its local queue, which was of length at least N at all times.) This promise gave p enough time to help all other processes (that are interested in B) obtain valid values for their LL operations. To ensure that all N processes are helped during this time, p would help a process with name $j = index$ during an SC operation, and then increment *index* by 1 modulo N. Since N is not fixed in the new algorithm, and since each process increments its *index* variable modulo *its local estimate* N, it is not clear that the above property still holds. We now show that it does.

Suppose that some process q reads (at Line 3 of its LL) the tag of a buffer B that holds the current value of an object $\mathcal{O}[i]$. Suppose further that after q performs that read, some process p performs a successful SC on $\mathcal{O}[i]$. Then, we will show that p does not reuse B before it checks whether q needs help, thereby ensuring that the above property holds. In the following, we let n and j denote, respectively, the values of p's estimate N and p's variable *index* at the time t when p inserts B's index its local queue Q (Line 13).

Notice that, by the algorithm, there are n items already in the local queue when B's index is inserted at time t. Hence, B is not written until p performs at least $n + 1$ dequeues on its local queue. Notice further that, each time p satisfies the condition at Line 19, the following holds: (1) p does not dequeue an element from its local queue, and (2) the values of N and *index* both increase by one. Moreover, each time p does not sat-

```
Types
    valuetype = array [0 .. W] of 64-bit value
    indextype = record type: {0, 1}; if (type == 0) (bindex: 31-bit number)
                                    else (name: 15-bit number; bindex: 15-bit number) end
    xtype = record seq: 32-bit number; buf: indextype end
    helptype = record seq: 31-bit number; helpme: {0, 1}; buf: indextype end
    blocktype = record Announce: 31-bit number; Help: helptype; Q: Single-process queue; BUF: dynamic array of *valuetype;
                       index: 15-bit number; mybuf: indextype; lseq: 31-bit number;
                       x: xtype; name: 15-bit number; N: 15-bit number end
Shared variables
    X: array [0 .. M − 1] of xtype; BUF: array [0 .. M − 1] of *valuetype; NameArray: dynamic array of *blocktype; N: 15-bit number
Local persistent variables at each p
    loc_p: blocktype
Initialization
    X[k] = (0, (0, k)), for all k ∈ {0, 1, ..., M − 1}; BUF[k] = the desired initial value of O[k], for all k ∈ {0, 1, ..., M − 1}; N = 0

    procedure LL(p, i, retval)
1:  loc_p.Announce = i
2:  loc_p.Help = (++loc_p.lseq, 1, loc_p.mybuf)
3:  loc_p.x = X[i]
4:  copy *GetBuf(loc_p.x.buf) into *retval
5:  if ¬CAS(loc_p.Help, (loc_p.lseq, 1, loc_p.mybuf),
                        (loc_p.lseq, 0, loc_p.mybuf))
6:      loc_p.mybuf = loc_p.Help.buf
7:      b = GetBuf(loc_p.mybuf)
8:      loc_p.x = b[W]
9:      copy *b into *retval
10: return

    procedure GetBuf(b) returns *valuetype
11: if (b.type == 0) return BUF[b.bindex]
12: l = da_read(NameArray, b.name)
13: return da_read(l→BUF, b.bindex)

    procedure VL(p, i) returns boolean
14: return (X[i] == loc_p.x)

    procedure SC(p, i, v) returns boolean
15: copy *v into *GetBuf(loc_p.mybuf)
16: if ¬CAS(X[i], loc_p.x, (loc_p.x.seq + 1, loc_p.mybuf))
17:     return false
18: enqueue(loc_p.Q, loc_p.x.buf)
19: if (loc_p.N < N)
20:     loc_p.N++
21:     da_write(loc_p.BUF, loc_p.N, malloc(W + 1))
22:     loc_p.mybuf = (1, loc_p.name, loc_p.N)
23: else loc_p.mybuf = dequeue(loc_p.Q)
24: l = da_read(NameArray, loc_p.index)
25: if (l→Help ≡ (s, 1, c))
26:     j = l→Announce
27:     x = X[j]
28:     d = GetBuf(loc_p.mybuf)
29:     copy *GetBuf(x.buf) into *d
30:     d[W] = x
31:     if CAS(l→Help, (s, 1, c), (s, 0, loc_p.mybuf))
32:         loc_p.mybuf = c
33: loc_p.index = (loc_p.index + 1) mod loc_p.N
34: return true
```

Figure 7: Implementation of $\mathcal{O}[0\,..\,M-1]$: an array of M W-word LL/SC objects shared by an unknown number of processes.

isfy the condition at Line 19, the following holds: (1) p dequeues an element from its local queue, and (2) the value of N remains the same and *index* increases by one (modulo N). As a result, the value of *index* wraps around to 0 after p dequeues exactly $n - j$ elements from its local queue. Let $t' > t$ be the first time after t when *index* wraps around to 0, and let n' be the value of N at time t'. Then, p dequeues at most j elements from the local queue before *index* again reaches value j. Consequently, at the moment when p performs the $(n+1)$st dequeue from its local queue (which returns the index of B), variable *index* has gone through the values $j, j+1, \ldots, n'-1, 0, 1, \ldots j-1, j$, and processes with names $j, j+1, \ldots, n'-1, 0, 1, \ldots, j-1$ have been helped by p. Since q has obtained a name prior to time t, it follows that q's name is certainly less than n' Therefore, p would have checked whether q needs help before reusing B, which proves the above property.

3.3.1 Implementation of Join and Leave

Figure 8 present the code for Join and Leave operations. As we stated earlier, the Join operation must (1) give each process a unique name, (2) give each process a unique memory block, (3) ensure that if K processes are participating in an algorithm then a new process obtains a name in the range $[0\,..\,K]$, (4) guarantee that if a process obtains a name n, then a pointer to its memory block has been written into the nth location of the array NameArray, and (5) ensure that variable N holds the maximum number of processes that have simultaneously participated in the algorithm so far. We now explain how the implementation in Figure 8 ensures these properties.

```
Types
    nodetype = record owned: boolean; loc: blocktype; next: *nodetype end
Shared variables
    Head: *nodetype
Local persistent variables at each process p
    node_p: *nodetype
Initialization
    Head = malloc(sizeof nodetype); Head→owned = false; Head→next = ⊥
    Head→loc = malloc(sizeof blocktype); Head→loc→Help = (0, 0, *)
    Init(*(Head→loc), 0)

    procedure Join(p)
35: mynode = malloc(sizeof nodetype)
36: mynode→owned = true
37: mynode→next = ⊥
38: mynode→loc = malloc(sizeof blocktype)
39: mynode→loc→Help = (0, 0, *)
40: name = 0
41: cur = Head
42: while (true)
43:     if CAS(cur→owned, false, true)
44:         free(mynode→loc)
45:         free(mynode)
46:         break
47:     da_write(NameArray, name, cur→loc)
48:     name++
49:     if (cur→next == ⊥)
50:         if CAS(cur→next, ⊥, mynode)
51:             cur = mynode
52:             break
53:     cur = cur→next
54: da_write(NameArray, name, cur→loc)
55: while ((n = N) < name + 1)
56:     CAS(N, n, name + 1)
57: loc_p = *(cur→loc)
58: node_p = cur
59: if (cur == mynode)
60:     Init(loc_p, name)

    procedure Leave(p)
61: node_p→owned = false

    procedure Init(loc, name)
62: loc.N = 1
63: da_write(loc.BUF, 0,
                malloc(W + 1))
64: da_write(loc.BUF, 1,
                malloc(W + 1))
65: loc.mybuf = (1, name, 0)
66: enqueue(loc.Q, (1, name, 1))
67: loc.index = 0
68: loc.name = name
```

Figure 8: Implementation of Join and Leave procedures, based on the renaming algorithm of Herlihy et al. [10] and the algorithm for allocating hazard-pointer records by Michael [21]

The algorithm for Join and Leave is essentially the same as the renaming algorithm of Herlihy et al. [10] and the algorithm for allocating new hazard-pointer records of Michael [21]. The algorithm maintains a linked list of nodes, with variable Head pointing to the head of the list. Each node in the list has a boolean field owned, which indicates whether the node is owned by some process or not. A node can be owned by at most one process at any given time. If a process p captures ownership of the kth node in the list, then its also captures ownership of the name k.[4] Each node in the list also has a field loc which holds the pointer to a memory block. The idea is that when a process p captures ownership of some node, it also captures ownership of the memory block at that node and will use that memory block in the LL/SC algorithm. Each node in the list has a field next which holds the pointer to the next node in the list. Finally, process p's local persistent variable $node_p$ holds the pointer to the node owned by p.

We now explain how the algorithm works. When a process p wishes to join the algorithm, it first prepares a new node that it will attempt to insert into the linked list (Lines 35–39). Next, it initializes its local variable *name* to 0, and, starting at the head of the list, tries to capture the first available node in the list (Lines 41–53). As we stated earlier, if p succeeds in capturing the kth node in the list, then it has also captured ownership of the name k as well as the memory block stored at that node. While traversing through the list, process p also makes sure that array NameArray matches the contents of the

[4] We assume that the list starts with the 0th node.

linked list, i.e., that the jth location in the array holds the pointer to the memory block stored at the jth node in the list.

In order to capture a node, p performs the CAS operation on the `owned` field of that node (Line 43), trying to change its value from *false* (indicating that no process owns the node), to *true* (indicating that the node is owned by some process). If p's CAS succeeds, it means that p has successfully captured the node, and so p terminates the loop and frees up the node that it had previously allocated (Lines 44–46). If p fails to in capturing a node (because a node was already owned by some other process or because some other process's CAS succeeded before p's), p increments its variable *name* (Line 48) and then writes the memory block at that node into array `NameArray` (Line 47). Next, p checks whether it is at the last node in the list (Line 49), and if so, it tries to insert its own node at the back of the list (Line 50). If p's CAS succeeds, it means that p has successfully installed its node at the end of the list. Furthermore, since p had already set the `owned` field of that node to *true* (at Line 36), it means that p has ownership of that node. Hence, p terminates its loop at Line 52. If, on the other hand, p's CAS fails, it means that some other process must have inserted its own node into the list. In that case, the node that p was currently visiting is no longer the last node in the list. So, p moves on to the next node in the list (Line 53) and repeats the above steps.

By the above algorithm, at the moment when p exits the loop, its variable *name* holds the position of the node in the list that p had captured (which is the same as p's new name). Since p had not previously written that node into array `NameArray`, p does so at Line 54. Notice that, if p captures the kth node in the list (i.e., if p's name is k), it means that p must have found the first k nodes to be owned by other processes. (Recall that the list starts with the 0th node.) Hence, the number of processes participating in the algorithm when p captures the kth node is $k+1$ or more [10]. To ensure that variable `N`, which holds the maximum number of processes participating in the algorithm so far, is up to date, process p performs the following steps. First, p reads `N` (Line 55). If the value of `N` is smaller than $k+1$, p tries to write $k+1$ into `N` (Line 56). There are two possibilities: either p's CAS succeeds or it fails. In the former case, `N` has been correctly updated; furthermore, the next time next time p tests the condition at Line 55, it will break out of the loop. In the latter case, some other process must have written into `N` and p may have to repeat the loop. However, since `N` is increased by at least one with each write, p will repeat the loop at most $k+1$ times. Consequently, after p's last iteration of the loop, `N` will hold a value greater than or equal to $k+1$.

Next, p sets its two persistent variables $node_p$ and loc_p to point to, respectively, the node in the list that p had captured and the memory block stored at that node (Lines 57 and 58). Finally, p checks whether it captured the same node that it had allocated at the beginning of the Join operation (Line 59). If so, it initializes the memory block stored at that node (Line 60). If p had captured some other node, then the memory block at that node has already been initialized (by a process who inserted that node into the list), and so there is no need for p to initialize that memory block.

The initialization of a block proceeds as follows. First, p sets the estimate of N to 1 (Line 62). Next, it allocated two new buffers and writes them at locations 0 and 1 of the array `BUF` (Lines 63 and 64). Then, p takes one of the two buffers to be its local buffer (Line 65) and enqueues the index of the other buffer into the local queue Q (Line 66). Finally, p sets its variable *index* to 0 and its variable *name* to *name* (Lines 67 and 68).

Operation Leave is extremely simple: p simply releases the ownership of the node it had previously captured during its Join operation (Line 61)

Based on the above discussion, we have the following theorem. Its proof is given in Appendix A.3.

Theorem 3 *The wait-free implementation in Figures 7 and 8 of an array $\mathcal{O}[0\,..\,M-1]$ of M W-word LL/SC objects is linearizable. The time complexity of LL, SC and VL operations on some variable in $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The time complexity of Join and Leave operations is $O(K)$ and $O(1)$, respectively, where K is the maximum number of processes that have simultaneously participated in the algorithm. The space complexity of the implementation is $O((K^2+M)W)$.*

References

[1] J. Anderson and M. Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182, September 1995.

[2] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–194, August 1995.

[3] IBM T.J Watson Research Center. *System/370 Principles of operation*, 1983. Order Number GA22-7000.

[4] T.D. Chandra, P. Jayanti, and K. Y. Tan. A polylog time wait-free construction for closed objects. In *Proceedings of the 17th Annual Symposium on Principles of Distributed Computing*, pages 287–296, June 1998.

[5] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 1: Application Architecture*, 2002. Revision 2.1.

[6] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 31–39, 2004.

[7] IBM Server Group. *IBM e server POWER4 System Microarchitecture*, 2001.

[8] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[9] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[10] M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking data structures. In *Proceedings of Computing: Australasian Theory Symposium*, 2003.

[11] M.P. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[12] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[13] SPARC International. *The SPARC Architecture Manual*. Version 9.

[14] A. Israeli and L. Rappoport. Disjoint-Access-Parallel implementations of strong shared-memory primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 151–160, August 1994.

[15] P. Jayanti. f-arrays: implementation and applications. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 270 – 279, 2002.

[16] P. Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, 2005.

[17] P. Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.

[18] P. Jayanti and S. Petrovic. Efficient wait-free implementation of multiword LL/SC variables. In *Proceedings of the 25th International Conference on Distributed Computing Systems*, 2005.

[19] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[20] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323, 2003.

[21] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[22] M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th Annual Conference on Distributed Computing*, pages 144–158, 2004.

[23] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 219–228, August 1997.

[24] G. L. Peterson. Concurrent reading while writing. *ACM TOPLAS*, 5(1):56–65, 1983.

[25] R. Site. *Alpha Architecture Reference Manual*. Digital Equipment Corporation, 1992.

[26] MIPS Computer Systems. *MIPS64™ Architecture For Programmers Volume II: The MIPS64™ Instruction Set*, 2002. Revision 1.00.

A Proofs

A.1 Proof of the algorithm in Figure 4

Let $\mathcal{H}$ be finite execution history of the algorithm in Figure 4. Let OP be some LL operation, OP$'$ some SC operation, and OP$''$ some VL operation on $\mathcal{O}[i]$ in $\mathcal{H}$, for

some i. Then, we define the linearization points (LPs) for OP, OP′, and OP″ as follows. If the CAS at Line 5 of OP succeeds, then LP(OP) is Line 3 of OP. Otherwise, let t be the time when OP executes Line 2, and t' be the time when OP performs the CAS at Line 5. Let v be the value that OP reads from BUF at Line 8 of OP. Then, we show that there exists a successful SC operation SC_q on $\mathcal{O}[i]$ such that (1) at some point t'' during (t, t'), SC_q is the latest successful SC on $\mathcal{O}[i]$ to execute Line 12, and (2) SC_q writes v into $\mathcal{O}[i]$. We then set LP(OP) to time t''. We set LP(OP′) to Line 12 of OP′, and LP(OP″) to Line 10 of OP″.

Lemma 1 *Let p be some process, and LL_p some LL operation by p in $\mathcal{H}$. Let t and t' be the times when p executes Line 2 and Line 5 of LL_p, respectively. Let t'' be either (1) the time when p executes Line 2 of its first LL operation after LL_p, if such operation exists, or (2) the end of $\mathcal{H}$, otherwise. Then, the following statements hold:*

(S1) During the time interval $(t, t']$, exactly one *write into* Help[p] *is performed.*

(S2) Any value written into Help[p] *during (t, t'') is of the form $(*, 0, *)$.*

(S3) Let $t''' \in (t, t']$ be the time when the write from statement (S1) takes place. Then, during the time interval (t''', t''), no process writes into Help[p].

Proof. Statement (S2) follows trivially from the fact that the only two operations that can affect the value of Help[p] during (t, t'') are (1) the CAS at Line 5 of LL_p, and (2) the CAS at Line 21 of some other process' SC operation, both of which attempt to write $(*, 0, *)$ into Help[p].

We now prove statement (S1). Suppose that (S1) does not hold. Then, during $(t, t']$, either (1) two or more writes on Help[p] are performed, or (2) no writes on Help[p] are performed. In the first case, we know (by an earlier argument) that each write on Help[p] during $(t, t']$ is performed either by the CAS at Line 5 of LL_p, or by the CAS at Line 21 of some other process' SC operation. Let CAS_1 and CAS_2 be the first two CAS operations on Help[p] to write into Help[p] during $(t, t']$. Then, by the algorithm, both CAS_1 and CAS_2 are of the form CAS(Help[p], $(*, 1, *)$, $(*, 0, *)$). Since CAS_1 succeeds and Help[p] doesn't change between CAS_1 and CAS_2, it follows that CAS_2 fails, which is a contradiction.

In the second case (where no writes on Help[p] take place during $(t, t']$), Help[p] doesn't change throughout $(t, t']$. Therefore, p's CAS at Line 5 of LL_p succeeds, which is a contradiction to the fact that no writes on Help[p] take place during $(t, t']$. Hence, statement (S1) holds.

We now prove statement (S3). Suppose that (S3) does not hold. Then, at least one write on Help[p] takes place during (t''', t''). By an earlier argument, any write on Help[p] during (t''', t'') is performed either by the CAS at Line 5 of LL_p, or by the CAS at Line 21 of some other process' SC operation. Let CAS_3 be the first CAS operation on Help[p] to write into Help[p] during (t''', t''). Then, by the algorithm, CAS_3 is of the form CAS(Help[p], $(*, 1, *)$, $(*, 0, *)$). Since Help[p] holds the value $(*, 0, *)$ at time t''' (by (S2)), and since Help[p] doesn't change between time t''' and CAS_3, it follows that CAS_3 fails, which is a contradiction. Hence, we have statement (S3). □

In Figure 9 we present a number of invariants satisfied by the algorithm. In the following, we let $PC(p)$ denote the value of process p's program counter. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in \{1, 2, 7-13, 16-21, 23, 24\}$ or $PC(p) \in \{3-5\} \wedge$ Help[p] $\equiv (*, 1, *)$. We let $\mathcal{P}'$ denote a set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) \in \{3-6\} \wedge$ Help[p] $\equiv (*, 0, *)$. We let $\mathcal{P}''$ denote a set of processes such that $p \in \mathcal{P}''$ if and only if $PC(p) = 14$. We let $\mathcal{P}'''$ denote a set of processes such that $p \in \mathcal{P}'''$ if and only if $PC(p) = 22$. Finally, we let $|Q_p|$ denote the length of process p's local queue Q_p.

Lemma 2 *The algorithm satisfies the invariants in Figure 9.*

Proof. (By induction) For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1-4, 7-11, 13, 16-20, 23, 24\}$, or if $PC(p) = \{5, 12, 21\}$ and p's CAS fails, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 5$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'$ and writes $mybuf_p$ into Help[p].*buf*. Consequently, invariant 5 holds by IH:4 and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 6$, then, by Lemma 1, p was in $\mathcal{P}'$ at time t. Furthermore, p is in $\mathcal{P}$ at time t'. Since p writes Help[p].*buf* into $mybuf_p$, invariant 4 holds by IH:5 and invariant 9 by IH:9. All other invariants trivially hold.

1. For any process p, we have $|Q_p| \geq N$.
2. For any process p such that $PC(p) = 15$, we have $|Q_p| \geq N + 1$.
3. For any process p and any value b in Q_p, we have $b \in [0\,..\,M + (N+1)N - 1]$.
4. For any processes $p \in \mathcal{P}$, we have $mybyf_p \in [0\,..\,M + (N+1)N - 1]$.
5. For any process $p \in \mathcal{P}'$, we have $\texttt{Help}[p].buf \in [0\,..\,M + (N+1)N - 1]$.
6. For any process $p \in \mathcal{P}''$, we have $x_p.buf \in [0\,..\,M + (N+1)N - 1]$.
7. For any process $p \in \mathcal{P}'''$, we have $b(p) \in [0\,..\,M + (N+1)N - 1]$.
8. For any index $i \in [0\,..\,M-1]$, we have $\texttt{X}[i].buf \in [0\,..\,M + (N+1)N - 1]$.
9. Let p and q, (respectively, p' and q', p'' and q'', p''' and q'''), be any two processes in $\mathcal{P}$ (respectively, $\mathcal{P}'$, $\mathcal{P}''$, $\mathcal{P}'''$). Let r be any process and b_1 and b_2 any two values in Q_r. Let i and j be any two indices in $[0\,..\,M-1]$. Then, we have $mybuf_p \neq mybuf_q \neq b_1 \neq b_2 \neq \texttt{X}[i].buf \neq \texttt{X}[j].buf \neq \texttt{Help}[p'].buf \neq \texttt{Help}[q'].buf \neq x_{p''}.buf \neq x_{q''}.buf \neq b(p''') \neq b(q''')$.

Figure 9: The invariants satisfied by the algorithm in Figure 4

If $PC(p) = 12$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}''$. Let $\texttt{X}[i]$ be the variable that p writes to. Then, since p's CAS is successful, we have $\texttt{X}[i].buf = x_p.buf$ at time t, and $\texttt{X}[i].buf = mybuf_p$ at time t'. Consequently, invariant 8 holds by IH:4, invariant 6 by IH:8, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 14$, then p leaves $\mathcal{P}''$. Furthermore, p enqueues $x_p.buf$ into the queue Q_p. Consequently, invariant 1 holds by IH:1, invariant 2 by IH:1, invariant 3 by IH:6, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 15$, then p joins $\mathcal{P}$. Furthermore, p reads and dequeues the front element of the queue Q_p. Consequently, invariant 4 holds by IH:3, invariant 1 by IH:2, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 21$ and p's CAS succeeds, then p moves from $\mathcal{P}$ to $\mathcal{P}'''$. Let $\texttt{Help}[q]$ be the variable that p writes into during this step. Then, we have $b(p) = \texttt{Help}[q].buf$ at time t, $\texttt{Help}[q].buf = mybuf_p$ at time t', and $\texttt{Help}[q]$ changes from value $(*, 1, *)$ at time t to a value $(*, 0, *)$ at time t'. Therefore, by Lemma 1, we have $PC(q) \in \{3-5\}$ at times t and t', and $\texttt{Help}[q].buf = mybuf_q$. Hence, q moves from $\mathcal{P}$ to $\mathcal{P}'$, and $b(p) = mybuf_q$. Consequently, invariant 5 holds by IH:4, invariant 7 by IH:4, and invariant 9 by IH:9. All other invariants trivially hold.

If $PC(p) = 22$, then p moves from $\mathcal{P}'''$ to $\mathcal{P}$. Furthermore, p writes $b(p)$ into $mybuf_p$. Consequently, invariant 4 holds by IH:7 and invariant 9 by IH:9. All other invariants trivially hold. □

Lemma 3 *Let $t_0 < t_1 < \ldots < t_K$ be all the times in $\mathcal{H}$ when some variable $\texttt{X}[i]$ is written to (by a successful CAS at Line 12). Then, for all $j \in \{0, 1, \ldots, K\}$, the value written into $\texttt{X}[i]$ at time t_j is of the form $(j, *)$.*

Proof. Suppose not. Let j be the smallest index such that, at time t_j, a value $k \neq j$ is written into $\texttt{X}[i]$ by some process p. (By initialization, we have $j \geq 1$.) Then, by the algorithm, p's CAS at time t_j is of the form $\texttt{CAS}(\texttt{X}[i], (k-1, *), (k, *))$. Since $\texttt{X}[i]$ holds value $j-1$ at time t_j, and since $k \neq j$, it follows that p's CAS fails, which is a contradiction to the fact that p writes into $\texttt{X}[i]$ at time t_j. □

Lemma 4 *Let $\mathcal{O}[i]$ be an LL/SC object. Let t be the time when some process p reads $\texttt{X}[i]$ (at Line 3 or 18), and $t' > t$ the first time after t that p completes Line 4 or Line 19. Let* OP *be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line* 12 *prior to time t, and v the value that* OP *writes in $\mathcal{O}[i]$. If there exists some process q such that $\texttt{Help}[q]$ holds value $(*, 1, *)$ throughout (t, t') and doesn't change, then p reads value v from $\texttt{BUF}$ at Line 4 or Line 19 (during (t, t')).*

Proof. Let r be the process executing OP. Since OP is the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 12 prior to time t, it follows that p reads from $\texttt{X}[i]$ at time t the value that r writes in $\texttt{X}[i]$ at Line 12 of OP. Therefore, p reads during (t, t') the same buffer B that r wrote v into at Line 11 of OP. Let t_1 be the time when r starts writing into B at Line 11 of OP, t_2 the time when r completes writing into B at Line 11 of OP, t_3 the time when r writes into $\texttt{X}[i]$ at Line 12 of OP, and t'' the time when p starts reading B during (t, t'). Then, the following claim holds.

Claim 1 *During (t_1, t_2), no process other than r writes into B. During (t_2, t'), no process writes into B.*

Proof. Suppose not. Then, either some process other than r writes into B during (t_1, t_2), or some process writes into B during (t_2, t'). In the first case, let r_1 be the process that writes into B during (t_1, t_2). Then, at some point during (t_1, t_2), we have $mybuf_{r_1} = mybuf_r$, which is a contradiction to Invariant 9. In the second case, let r_2 be the first process to start writing into B at some time $\tau_1 \in (t_2, t')$, and k be the index of buffer B. Then, by an earlier argument, $\tau_1 \notin (t_2, t_3)$. Furthermore, by Invariant 9 , r_2 does not write into B as long as $\mathtt{X}[i]$ holds value $(*, k)$. Therefore, $\mathtt{X}[i]$ changes during (t_3, τ_1).

Since $\mathtt{X}[i]$ doesn't change during (t_3, t), it means that (1) $\tau_1 > t$ and (2) some process writes into $\mathtt{X}[i]$ during (t, τ_1). Let r_3 be the first such process, $\tau_2 \in (t, \tau_1)$ the time when r_3 writes into $\mathtt{X}[i]$, and SC_{r_3} the SC operation during which r_3 performs that write. Let τ_3 be the time when r_3 executes Line 14 of SC_{r_3}. Then, at time τ_3, r_3 enqueues k into Q_{r_3}. Furthermore, by Invariant 9, r_2 does not write into B during (τ_2, τ_3), nor does it write into B during the time Q_{r_3} contains value k. Therefore, we have $\tau_3 \in (\tau_2, \tau_1)$. Finally, we know that k is dequeued from Q_{r_3} during (τ_3, τ_1).

Let τ_4 be the first time after τ_3 that k is dequeued from Q_{r_3}. (Notice that, by the above argument, $\tau_4 \in (\tau_3, \tau_1)$.) Then, by Invariant 1, r_3 executes Lines 16–23 N times during (τ_3, τ_4). Since during each execution of Lines 16–23 r_3 increments variable $index_{r_3}$ by 1 modulo N, there exists an execution E of Lines 16–23 during which $index_{r_3} = q$. Because $\mathtt{Help}[q]$ holds value $(*, 1, *)$ throughout (t, t') and doesn't change, it follows that (1) r_3 satisfies the condition at Line 16 of E, and (2) r_3's CAS at Line 21 of E succeeds. This, however, is a contradiction to the fact that $\mathtt{Help}[q] = (*, 1, *)$ throughout (t, t'). Hence, we have the claim. □

The above claim shows that (1) during (t_1, t_2), no process other than r writes into B, and (2) during (t_2, t'), no process writes into B. Consequently, p reads v from B during (t, t'), which proves the lemma. □

Lemma 5 *Let $\mathcal{O}[i]$ be an LL/SC object and* OP *some LL operation on $\mathcal{O}[i]$. Let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line* 12 *prior to Line* 3 *of* OP*, and v_q the value that SC_q writes in $\mathcal{O}[i]$. If the CAS at Line* 5 *of* OP *succeeds, then* OP *returns value v_q.*

Proof. Let p be the process executing OP. Let t time when p executes Line 3 of OP, and $t' > t$ be the time when p completes Line 4 of OP. Since the CAS at Line 5 of OP succeeds, it follows by Lemma 1 that $\mathtt{Help}[p]$ holds value $(*, 1, *)$ throughout (t, t') and doesn't change during that time. Therefore, by Lemma 4, p reads v_q from BUF at Line 4 of OP, which proves the lemma. □

Lemma 6 *Let $\mathcal{O}[i]$ be an LL/SC object, and* OP *an LL operation on $\mathcal{O}[i]$ such that the CAS at Line* 5 *of* OP *fails. Let p be the process executing* OP*. Let t and t' be the times, respectively, when p executes Lines* 2 *and* 5 *of* OP*. Let x and v be the values that p reads from* BUF *at Lines 7 and 8 of* OP*, respectively. Then, there exists a successful SC operation SC_q on $\mathcal{O}[i]$ such that (1) at some point during (t, t'), SC_q is the latest successful SC on $\mathcal{O}[i]$ to execute Line* 12*, and (2) SC_q writes x into $\mathtt{X}[i]$ and v into $\mathcal{O}[i]$.*

Proof. Since p's CAS at time t' fails, it means that $\mathtt{Help}[p] = (s, 0, b)$ just prior to t'. Then, by Lemma 1, there exists a single process r that writes into $\mathtt{Help}[p]$ during (t, t') (at Line 21). Let $t_1 \in (t, t')$ be the time when r performs that write, and E be r's execution of Lines 16–22 during which r performs that write. Then, r's CAS at Line 21 of E (at time t_1) is of the form $\mathrm{CAS}(\mathtt{Help}[p], (s, 1, *), (s, 0, *))$, for some s. Therefore, at time t_1, $\mathtt{Help}[p]$ has value $(s, 1, *)$. Hence, by Lemma 1, p writes $(s, 1, *)$ into $\mathtt{Help}[p]$ at Line 2 of OP (at time t). Since a value of the form $(s, *, *)$ is written into $\mathtt{Help}[p]$ for the first time at time t, it follows that r reads $(s, 1, *)$ from $\mathtt{Help}[p]$ at Line 16 of E at some time $t_2 \in (t, t_1)$. Consequently, r reads variable $\mathtt{Announce}[p]$ at Line 17 of E at some time $t_3 \in (t_2, t_1)$. Since p writes i into $\mathtt{Announce}[p]$ at Line 1 of OP, it follows that r reads i from $\mathtt{Announce}[p]$ at time t_2. Hence, r reads $\mathtt{X}[i]$ at Line 18 of E.

Let t_4 be the time when r reads $\mathtt{X}[i]$ at Line 18 of E, t_5 the time when r starts Line 19 of E, t_6 the time when r completes Line 19 of E, and t_7 the time when r executes Line 20 of E. Let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 12 prior to time t_4, x_q the value that SC_q writes in $\mathtt{X}[i]$, and v_q the value that SC_q writes in $\mathcal{O}[i]$. Then, at time t_4, r reads x_q from $\mathtt{X}[i]$. Furthermore, since t_1 is the first (and only) time that $\mathtt{Help}[p]$ is written during (t, t'), it follows that $\mathtt{Help}[p]$ holds value $(*, 1, *)$ at all times during (t_4, t_6) and doesn't change during that time. Therefore, by Lemma 4, r reads v_q from BUF during (t_5, t_6).

Let B be the buffer that r writes v_q into during (t_5, t_6). Then, at time t_7, r writes x_q into $B[W]$. Furthermore, since r writes the index of buffer B into $\mathtt{Help}[p]$ at Line 21 of E (at time t_1), it follows that p reads buffer B at Lines 7 and 8 of OP. Let t_8 be the time when p

reads $B[W]$ at Line 7 of OP, t_9 the time when p starts reading B at Line 8 of OP, and t_{10} the time when p completes reading B at Line 8 of OP. Then, we show that the following claim holds.

Claim 2 *During (t_5, t_6), no process other than r writes into B, and during (t_6, t_{10}), no process writes into B.*

Proof. Suppose not. Then, either some process other than r writes into B during (t_5, t_6), or some process writes into B during $(t_6, t_1 0)$. In the former case, let r_1 be the process that writes into B during (t_5, t_6). Then, at some point during (t_5, t_6), we have $mybuf_{r_1} = mybuf_r$, which is a contradiction to Invariant 9. In the latter case, let r_2 be the first process to write into B at some time $\tau_1 \in (t_6, t_{10})$. Then, by an earlier argument, we know that $\tau_1 \notin (t_6, t_1)$. We now show that $\tau_1 \notin (t_1, t_{10})$.

Let b be the index of buffer B. We know by Invariant 9 that r_2 does not write into B as long as (1) $\texttt{Help}[p] = (s, 0, b)$, and (2) p is between Lines 2 and 6 of OP. Furthermore, since p sets $mybuf_p$ to b at Line 6 of OP, r_2 does not write into B after p executes Line 6 of OP and before it completes OP. Therefore, throughout (t_1, t_{10}), r_2 does not write into B. Hence, $\tau_1 \notin (t_1, t_{10})$. Since, by an earlier argument, $\tau_1 \notin (t_6, t_1)$, it follows that $\tau_1 \notin (t_6, t_{10})$. This, however, is a contradiction to the fact that r_2 writes into B during (t_6, t_{10}). □

The above claim shows that (1) during (t_5, t_6), no process other than r writes into B, and (2) during (t_6, t_{10}), no process writes into B. Consequently, p reads x_q from $B[W]$ at time t_8 and v_q from B during (t_9, t_{10}). Since SC_q is the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 12 prior to time t_4, and since $t_4 \in (t, t')$, we have the lemma. □

Lemma 7 (Correctness of LL) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any LL operation on $\mathcal{O}[i]$, and* OP′ *be the latest successful SC operation on $\mathcal{O}[i]$ such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. We examine the following two cases: (1) the CAS at Line 5 of OP succeeds, and (2) the CAS at Line 5 of OP fails. In the first case, let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 12 prior to Line 3 of OP, and v_q be the value that SC_q writes in $\mathcal{O}[i]$. Since all SC operations are linearized at Line 12 and since OP is linearized at Line 3, we have $\text{SC}_q = \text{OP}'$. Furthermore, by Lemma 5, OP returns value v_q. Therefore, the lemma holds in this case.

In the second case, let t and t' be the times, respectively, when p executes Lines 2 and 5 of OP. Let v be the value that p reads from BUF at Line 8 of OP. Then, by Lemma 6, there exists a successful SC operation SC_r on $\mathcal{O}[i]$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}[i]$ to execute Line 12, and (2) SC_r writes v into $\mathcal{O}[i]$. Since all SC operations are linearized at Line 12 and since OP is linearized at time t'', we have $\text{SC}_r = \text{OP}'$. Therefore, the lemma holds. □

Lemma 8 (Correctness of SC) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any SC operation on $\mathcal{O}[i]$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}[i]$ by p prior to* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *on $\mathcal{O}[i]$ such that LP(*OP′*) < LP(*OP″*) < LP(*OP*).*

Proof. We examine the following two cases: (1) the CAS at Line 5 of OP′ succeeds, and (2) the CAS at Line 5 of OP′ fails. In the first case, let t_1 be the time when p executes Line 3 of OP′, and t_2 be the time when p executes Line 12 of OP. Then, we show that the following claim holds.

Claim 3 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}[i]$ that performs a successful CAS at Line 12 during (t_1, t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}[i]$ performs a successful CAS at Line 12 during (t_1, t_2). Then, $\texttt{X}[i]$ doesn't change during (t_1, t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}[i]$ does perform a successful CAS at Line 12 during (t_1, t_2). Then, by Lemma 3, $\texttt{X}[i]$ holds different values at times t_1 and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 12 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}[i]$ such that $\text{LP}(\text{OP}') < \text{LP}(\text{OP}'') < \text{LP}(\text{OP})$. Hence, the lemma holds in this case.

In the second case (when the CAS at Line 5 of OP′ fails), let t and t' be the times when p executes Lines 2 and 5 of OP′, respectively. Let x and v be the values that p reads from BUF at Lines 7 and 8 of OP′, respectively. Then, by Lemma 6, there exists a successful SC operation SC_r on $\mathcal{O}[i]$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}[i]$ to execute Line 12, and (2) SC_r writes x into $\texttt{X}[i]$ and v

into $\mathcal{O}[i]$. Therefore, at Line 7 of OP′, p reads the value that variable X[i] holds at time t''. We now prove the following claim.

Claim 4 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}[i]$ that performs a successful CAS at Line 12 during (t'', t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}[i]$ performs a successful CAS at Line 12 during (t'', t_2). Then, X[i] doesn't change during (t'', t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}[i]$ does perform a successful CAS at Line 12 during (t'', t_2). Then, by Lemma 3, X[i] holds different values at times t'' and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 12 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}[i]$ such that LP(OP′) < LP(OP″) < LP(OP). Hence, the lemma holds. □

Lemma 9 (Correctness of VL) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any VL operation on $\mathcal{O}[i]$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}[i]$ by p that precedes* OP*. Then,* OP *returns true if and only if there does not exist some successful SC operation* OP″ *on $\mathcal{O}[i]$ such that LP(*OP″*) ∈ (LP(*OP′*), LP(*OP*)).*

Proof. Similar to the proof of Lemma 8. □

Theorem 1 *The algorithm in Figure 4 is a linearizable, wait-free implementation of an array $\mathcal{O}[0\,..\,M-1]$ of W-word LL/SC objects, shared by N processes. The time complexities of LL, SC and VL operations on any $\mathcal{O}[i]$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The space complexity of the implementation is $O((N^2+M)W)$.*

Proof. The theorem follows immediately from Lemmas 7, 8, and 9. □

A.2 Proof of the algorithm in Figure 5

Let $\mathcal{H}$ be the complete execution history of the DynamicArray algorithm in Figure 5. Then, we show that the following lemmas hold.

Lemma 10 *Let $t_1, t_2, \ldots, t_m$ be all the times in $\mathcal{H}$ that variable* D *is written. Let d_i, for all $i \in \{1, 2, \ldots, m\}$, be the value written into* D *at time t_i. Let d_0 be the initializing value for* D*. Then, we have $d_0 \neq d_1 \neq \ldots \neq d_m$. Furthermore, for all $i \in \{0, 1, \ldots, m\}$, we have: (1) $d_i \rightarrow size = 2^i$, (2) $d_i \rightarrow A$ is an array of size 2^i, (3) $d_i \rightarrow B$ is an array of size 2^{i+1}, and (4) $d_i \rightarrow A = d_{i-1} \rightarrow B$ for $i > 0$.*

Proof. Suppose that the first part of the claim doesn't hold. Then, there exist some indices i and j in $\{0, 1, \ldots, m\}$ such that $d_i = d_j$. Let t'_i (respectively, t'_j) be the latest time prior to t_i (respectively, t_j) that d_i was returned by a `malloc` at Line 6 (or during initialization). Then, by the algorithm, d_i (respectively, d_j) is not freed after time t_i (respectively, t_j). Hence, by the uniqueness of allocated addresses, we have $t'_j < t'_i$ and $t'_i < t'_j$, which is a contradiction. Therefore, we have the first part of the claim.

We prove the second part of the claim by induction. Suppose that that the claim holds for all $i < j$; we show that the claim holds for j as well. Let p be the process that writes d_j into D at time t_j. Let t be the latest time prior to t_j that p reads D (at Line 1). Then, since p's CAS at time t_j succeeds, p reads d_{j-1} from D at time t. Furthermore, since, by the first part of the claim, we have $d_0 \neq d_1 \neq \ldots \neq d_{j-1} \neq d_j$, it follows that $t \in (t_{j-1}, t_j)$. By inductive hypothesis, we have (1) $d_{j-1} \rightarrow size = 2^{j-1}$, (2) $d_{j-1} \rightarrow A$ is of size 2^{j-1}, and (3) $d_{j-1} \rightarrow B$ is of size 2^j. Therefore, during time (t, t_j), p (1) writes 2^j into $d_j \rightarrow size$ (Line 9), (2) sets $d_j \rightarrow B$ to be a new array of size 2^{j+1} (Line 8), and (3) sets $d_j \rightarrow A$ to be $d_{j-1} \rightarrow B$ (Line 7). Hence, we have the claim. □

In the following, let K be the maximum i such that `write`$(i, *)$ is invoked in $\mathcal{H}$. Let $\mathcal{E}_i$, for all $i \in \{0, 1, \ldots, K\}$, be the collection of all executions of `write`$(i, *)$ in $\mathcal{H}$. Let t_i, for all $i \in \{0, 1, \ldots, K\}$, be the earliest time some execution in $\mathcal{E}_i$ is invoked. Let t'_i, for all $i \in \{0, 1, \ldots, K\}$, be the earliest time some execution in $\mathcal{E}_i$ completes. (Notice that, by the definition of DynamicArray, $t_i > t'_{i-1}$ for all $i \in \{1, 2, \ldots, K\}$.) Let v_i, for all $i \in \{0, 1, \ldots, K\}$, be the value written by an execution in $\mathcal{E}_i$. Let $c(i)$, for all $i \in \{1, 2, \ldots, K\}$, be the smallest power of 2 *greater or equal* to i, and $c'(i)$ be the largest power of 2 *smaller or equal* to i. (If $i = 0$, then $c'(i) = 0$.) If E is an execution of procedure `write` in $\mathcal{H}$, then, in the following, we slightly abuse notation, and say that "E executes Line 4", instead of "process p executing E executes Line 4."

Lemma 11 *For all $i \in \{0, 1, \ldots, K\}$, we have the following.*

If i is not a power of 2:

(1) $\mathcal{E}_i \neq \emptyset$.

(2) Let d be the value of variable `D` *at time t_i. Then, we have $d \rightarrow size = c(i)$.*

(3) For all $E \in \mathcal{E}_i$, E does not execute Lines 6–11. (Notice that this implies that E does not invoke the `write` *procedure at Line 11; therefore, we can say, for example, "Line 4 of E," without being ambiguous.)*

(4) Variable `D` *doesn't change during (t_i, t_i').*

(5) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E writes v_i into $d \rightarrow A[i]$ at Line 3.*

(6) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E writes v_i into $d \rightarrow B[i]$ at Line 4.*

(7) Let E be any execution in $\mathcal{E}_i$, and d be the value that E reads from `D` *at Line 1. Then, E copies $d \rightarrow A[i - c'(i)]$ into $d \rightarrow B[i - c'(i)]$ at Line 5.*

If i is a power of 2:

(8) $\mathcal{E}_i \neq \emptyset$.

(9) Let d be the value of variable `D` *at time t_i. Then, we have $d \rightarrow size = i$.*

(10) For all $E \in \mathcal{E}_i$, E executes `write` *at Line 11 at most once. (In the following, let $\mathcal{E}_i' \subset \mathcal{E}_i$ be the collection of all executions in $\mathcal{E}_i$ that execute* `write` *at Line 11, and $\mathcal{E}_i'' \subset \mathcal{E}_i$ be the collection of all executions in $\mathcal{E}_i$ that do not execute* `write` *at Line 11. Let $E(1)$ and $E(2)$, for all $E \in \mathcal{E}_i'$, denote E's first and second execution of Lines 1–11, respectively.)*

(11) Exactly one execution $E \in \mathcal{E}_i$ performs a successful CAS at Line 10. Furthermore, E performs this CAS at some time t_i'' during (t_i, t_i').

(12) Variable `D` *changes exactly once during (t_i, t_i'), at time t_i''.*

(13) Let E (respectively, E') be any execution in $\mathcal{E}_i'$ (respectively, $\mathcal{E}_i''$). Then, $E(2)$ (respectively, E') executes Line 1 after time t_i''.

(14) Let E (respectively, E') be any execution in $\mathcal{E}_i'$ (respectively, $\mathcal{E}_i''$), and d be the value that $E(2)$ (respectively, E') reads from `D` *at Line 1. Then, $E(2)$ (respectively, E') writes v_i into $d \rightarrow A[i]$ at Line 3.*

(15) Let E (respectively, E') be any execution in $\mathcal{E}_i'$ (respectively, $\mathcal{E}_i''$), and d be the value that $E(2)$ (respectively, E') reads from `D` *at Line 1. Then, $E(2)$ (respectively, E') writes v_i into $d \rightarrow B[i]$ at Line 3.*

(16) Let E (respectively, E') be any execution in $\mathcal{E}_i'$ (respectively, $\mathcal{E}_i''$), and d be the value that $E(2)$ (respectively, E') reads from `D` *at Line 1. Then, $E(2)$ (respectively, E') copies $d \rightarrow A[0]$ into $d \rightarrow B[0]$ at Line 5.*

Proof. (By induction) We assume that the lemma holds for all $i < j$, and show that it also holds for j. (During the proof of the inductive step, we will also prove the base case of $i = 0$.) We first prove the case when j is not a power of 2.

Let d be the value of variable `D` at time t_j. By inductive hypothesis, `D` has changed exactly $\lg(c(j))$ times prior to time t_j (by Statements (3) and (11)). Then, by Lemma 10, we have (1) $d \rightarrow size = c(j)$, (2) $d \rightarrow A$ is an array of size $c(j)$, and (3) $d \rightarrow B$ is an array of size $2c(j)$. Hence, Statement (2) holds. Since, by Lemma 10, the value of $\texttt{D} \rightarrow size$ only increases after time t_j, it follows that all executions in $\mathcal{E}_j$ satisfy the condition at Line 2. Therefore, no execution in $\mathcal{E}_j$ executes Line 6–11, which proves Statement (3). Statements (5), (6), and (7) follow directly from the algorithm. Statement (1) follows trivially by the definition of DynamicArray. Statement (4) follows directly from the following claim.

Claim 5 *During (t_j, t_j'), variable* `D` *doesn't change.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t_j'), no execution in $\mathcal{E}_j$ writes into `D` (by Statement (3)), and (2) during (t_j, t_j'), no execution in $\mathcal{E}_i$, for all $i < j$, writes into `D` (by inductive hypothesis for Statements (3) and (11)). □

We now prove the case when j is a power of 2. Let d be the value of variable `D` at time t_j. By inductive hypothesis, `D` has changed exactly $\lg j$ times prior to time t_j (by Statements (3) and (11)). Then, by Lemma 10, we have (1) $d \rightarrow size = j$, (2) $d \rightarrow A$ is an array of size j, and (3) $d \rightarrow B$ is an array of size $2j$. Hence, Statement (9) holds. We now prove the following claims.

Claim 6 *Let t be either (1) the earliest time during (t_j, t_j') that some execution $E \in \mathcal{E}_j$ performs a CAS at Line 10, or (2) t_j, if there is no such execution. Then, throughout (t_j, t), variable* `D` *holds value d.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t), no execution in $\mathcal{E}_j$ writes into `D`

(by definition of t), (2) during (t_j, t), no execution in $\mathcal{E}_i$, for all $i < j$, writes into D (by inductive hypothesis for Statements (3) and (11)), and (3) D holds value d at time t_j. □

Claim 7 *At least one execution $E \in \mathcal{E}_j$ performs a CAS at Line 10 during (t_j, t'_j).*

Proof. Suppose not. Then, during (t_j, t'_j), no execution in $\mathcal{E}_j$ performs a CAS at Line 10. Consequently, by Claim 6, variable D holds value d throughout (t_j, t'_j). Let $E \in \mathcal{E}_j$ be the execution that completes at time t'_j. Let $E(1)$ be the first execution of Lines 1–11 by E. Then, $E(1)$ reads d from D at Line 1. Therefore, $E(1)$ does not satisfy the condition at Line 2 and hence executes Line 6–11, which is a contradiction to the fact that no execution in $\mathcal{E}_j$ performs a CAS at Line 10 during (t_j, t'_j). □

Claim 8 *Let t be the earliest time (by Claim 7) during (t_j, t'_j) that some execution $E \in \mathcal{E}_j$ performs a CAS at Line 10. Then, E's CAS at time t succeeds.*

Proof. Notice that, by Claim 6, variable D holds value d throughout (t_j, t). Therefore, E reads d from D at Line 1, and E's CAS at time t succeeds, which proves the claim. □

Claim 9 *Let t be the earliest time (by Claim 7) during (t_j, t'_j) that some execution in $\mathcal{E}_j$ performs a CAS at Line 10. Let E be any execution in $\mathcal{E}_j$, and $E(l)$ any execution of Lines 1–11 of E. Then, if $E(l)$ executes Line 1 prior to time t, it executes Lines 6–11. Otherwise, it executes Lines 3–5.*

Proof. By Claim 6, we know that variable D holds value d throughout (t_j, t). Therefore, if $E(l)$ executes Line 1 before time t, it will find d in variable D. Consequently, $E(l)$ will not satisfy the condition at Line 2, and will hence execute Lines 6–11.

Suppose that $E(l)$ executes Line 1 after time t. Let d' be the value that $E(l)$ reads from D at Line 1. Then, by Lemma 10 and Claim 8, we have $d' \rightarrow size \geq 2j$. Therefore, $E(l)$ satisfies the condition at Line 2, and hence executes Lines 3–5. □

Claim 10 *At most one execution $E \in \mathcal{E}_j$ performs a successful CAS at Line 10.*

Proof. Suppose not. Then, two or more executions in $\mathcal{E}_j$ perform a successful CAS at Line 10. Let t and t' be the earliest two times that some execution in $\mathcal{E}_j$ performs a successful CAS at Line 10. Then, by Claim 8, t is the earliest time during (t_j, t'_j) that any execution in $\mathcal{E}_j$ performs a CAS at Line 10.

Let E be the execution in $\mathcal{E}_j$ that performs a successful CAS at time t'. Let $E(l)$ be the execution of Lines 1–11 of E during which E performs that CAS. Let t'' be the time when $E(l)$ executes Line 1, and d' be the value that $E(l)$ reads from D at time t''. Then, since $E(l)$'s CAS at time t' succeeds, it follows that $t'' \in (t, t')$ (by Lemma 10). Therefore, by Claim 9, $E(l)$ executes Lines 3–5, which is a contradiction to the fact that $E(l)$ performs a successful CAS at Line 10. □

Notice that, by Claim 9, if an execution $E \in \mathcal{E}_j$ executes `write` at Line 11, it will not execute Lines 6–11 again. Therefore, we have Statement (10). Statement (11) follows directly from Claims 8 and 10. Statement (13) follows directly from Claim 9. Statements (14), (15), and (16) follow directly by the algorithm. Statement (8) follows trivially by the definition of DynamicArray. Statement (12) follows directly from the following claim.

Claim 11 *During (t_j, t'_j), variable D changes exactly once, at time t''_i.*

Proof. The claim follows immediately from the fact that (1) during (t_j, t'_j), no execution in $\mathcal{E}_i$, for all $i < j$, writes into D (by inductive hypothesis for Statements (3) and (11)), and (2) during (t_j, t'_j), exactly one execution in $\mathcal{E}_j$ writes into D (by Statement (11)). □

□

Lemma 12 *No execution E in $\mathcal{H}$ writes or reads an unallocated memory region at Lines 3, 4, or 5.*

Proof. Let E be an execution in $\mathcal{H}$, and $\mathcal{E}_s$ the collection that E belongs to, for some $s \in \{1, 2, \ldots, K\}$. If s is not a power of 2, let d be the value of variable D that E reads at Line 1. Then, by Statement (2) of Lemma 11 and by Lemma 10, we have $d \rightarrow size \geq c(s)$. Furthermore, arrays $d \rightarrow A$ and $d \rightarrow B$ are of sizes at least $c(s)$ and $2c(s)$, respectively. Therefore, E writes into a valid array location at Lines 3 and 4. Since $s > c'(s)$, E reads and writes a valid array location at Line 5 as well.

If s is not a power of 2, we examine two possibilities: either $E \in \mathcal{E}'_j$ or $E \in \mathcal{E}''_j$. In the first case, let d be the value of variable D that $E(2)$ reads at Line 1. Then, by Statement (13) of Lemma 11 and by Lemma 10, we have

$d \to size \geq 2s$. Furthermore, arrays $d \to A$ and $d \to B$ are of sizes at least $2s$ and $4s$, respectively. Therefore, $E(2)$ writes into a valid array location at Lines 3 and 4. Since $s = c'(s)$, E reads and writes a valid array location at Line 5 as well. (The argument for the second case is identical, and is therefore omitted.) □

Lemma 13 *Let R be any read$(i, *)$ operation in $\mathcal{H}$, for some $i \in \{0, 1, \ldots, K\}$. Then, R returns v_i.*

Proof. (In the following, let $D(t)$ to denote the value of variable D at time t.) Let t_d and t_r be the times when R executes Lines 12 and 13, respectively. Then, we show that the following claim holds:

Claim 12 *The length of the array $D(t_d) \to A$ is at least $i + 1$.*

Proof. Notice that, by the definition of DynamicArray, at least one execution $E \in \mathcal{E}_i$ completes before R starts. Then, the claim follows immediately by Statements (2), (9), and (12) of Lemma 11 and by Lemma 10. □

Suppose that the lemma doesn't hold. Then, the value that R reads from $D(t_d) \to A[i]$ at time t_r is different than v_i. By Lemma 11, we know that at least one execution in $\mathcal{E}_i$ writes v_i into both $D(t_i) \to A[i]$ and $d' \to B[i]$ during (t_i, t_i'). Furthermore, D doesn't change during (t_i, t_i') and no other execution in $\mathcal{E}_j$, for all $j < i$, writes into $D(t_i) \to A[i]$ and $D(t_i) \to B[i]$ during (t_i, t_i'). Therefore, we have $D(t_i') \to A[i] = v_i$ and $D(t_i') \to B[i] = v_i$ at time t_i'.

Let t be any time in (t_i', t_r). Let $t^1, t^2, \ldots, t^{m(t)}$ be all the times during (t_i', t) when D changes. Let A_j, for all $j \in \{1, 2, \ldots, m(t)\}$, be the array in $D(t^i) \to A$. Let B_j, for all $j \in \{1, 2, \ldots, m(t)\}$, be the array in $D(t^i) \to B$. Let A_0 be the array in $D(t_i') \to A$, B_0 the array in $D(t_i') \to B$, and $t^0 = t_i'$. Then, we prove the following claim.

Claim 13 *If no execution writes a value different than v_i at index i (of some array) during (t_i', t), then at all times during (t^j, t) and for all $j \in \{0, 1, 2, \ldots, m(t)\}$, we have $A_j[i] = v_i$.*

Proof. Suppose not. Then, let t' be the earliest time that the following occurs: for some k, $A_k[i] \neq v_i$ at time $t' \in (t^k, t)$. Notice that, since $A_0[i] = v_i$ at time t_i', and since no execution writes a value different than v_i at index i during (t_i', t), it follows that $k \neq 0$. Similarly, since $B_0[i] = v_i$ at time t_i', and since no execution writes a value different than v_i at index i during (t_i', t), it follows that $A_1[i] = v_i$ at time t^1 and $A_1[i] = v_i$ throughout (t^1, t). Therefore, we have $k \geq 2$.

It follows by Lemma 11 that some execution in $\mathcal{E}_{2^{k-2}c(i)}$ writes into D at time t^{k-1}. Furthermore, some execution in $\mathcal{E}_{2^{k-1}c(i)}$ writes into D at time t^k. Finally, during (t^{k-1}, t^k), some execution $E \in \mathcal{E}_{2^{k-2}c(i)+i}$ reads the value in $A_{k-1}[i]$ (at Line 5) and writes that value into $B_{k-1}[i]$ (at Line 5). Then, by definition of t', E reads v_i from $A_{k-1}[i]$. Therefore, E writes v_i into $B_{k-1}[i]$. Consequently, since no execution writes a value different than v_i at index i during (t_i', t), it follows that $A_k[i] = v_i$ at time t^k and $A_k[i] = v_i$ throughout (t^k, t), which is a contradiction to the fact that $A_k[i] \neq v_i$ at time t'. □

By Claim 13, some execution writes a value different than v_i at index i (of some array) during (t_i', t_r) (because R reads a value different than v_i at time t_r). Let t_e be the earliest time that some execution E writes a value different than v_i at index i (of some array) during (t_i', t_r). Then, by Lemma 11, (1) E writes into one of the arrays $A_0, A_1, \ldots, A_{m(t_r)}$ or $B_0, B_1, \ldots, B_{m(t_r)}$, (2) $E \in \mathcal{E}_j$, for some $j > i$, and (3) E's write at time t_e takes place at Line 5 of E. Therefore, E writes into $B_k[i]$ at time t_e, for some $k \in 0, 1, \ldots, m(t_r)$.

Notice that E reads $A_k[i]$ at Line 5 prior to time t_e. Since t_e is the first time during (t_i', t_r) that some execution writes a value different than v_i at index i (of some array), it follows that during (t_i', t_e), no execution writes a value different than v_i at index i (of some array). Therefore, by Claim 13, we have $A_k[i] = v_i$ throughout (t_i', t_e). Consequently, E reads v_i from $A_k[i]$ at Line 5, and therefore writes v_i into $B_k[i]$ at time t_e, which is a contradiction. Hence, we have the lemma. □

Theorem 2 *The algorithm in Figure 5 is wait-free and implements a DynamicArray object $\mathcal{D}$ from a word-sized CAS object and registers. The time complexity of read and write operations on $\mathcal{D}$ is $O(1)$. The space used by the algorithm at any time t is $O(nK)$, where n is the number of processes executing the algorithm at time t, and K is the highest location written in $\mathcal{D}$ prior to time t.*

Proof. The theorem follows immediately from Lemma 13. □

A.3 Proof of the algorithm in Figures 7 and 8

A.3.1 Lemmas associated with the algorithm in Figure 8

We say that node n is *allocated* at time t if there exists a call to `malloc` at time t (at Line 35) that returns n. Node n is *released* at time t if some process executes a `free` operation at Line 45 at time t with the argument n. Node n is *installed* at time t if some process executes a successful CAS at Line 50 at time t with the third argument n.

We say that node n is *alive* at time t if it has been allocated prior to time t but hasn't been released, or if n is the initial dummy node and n hasn't been released. Node n is *active* at time t if it has been installed prior to time t or if it is the initial dummy node. We let *Alive* denote the set of all nodes that are alive. We let L denote the sequence of nodes that are active, arranged in the order of their installation.

We let $PC(p)$ denote the value of process p's program counter at time t. For any register r at process p, we let $r(p)$ denote the value of that register at time t. We let $\mathcal{P}'$ denote the set of processes such that $p \in \mathcal{P}'$ if and only if $PC(p) \in \{36-45, 47-50, 53\}$. We let $\mathcal{P}''$ denote the set of processes such that $p \in \mathcal{P}''$ if and only if $PC(p) \in \{38-45, 47-50, 53\}$. We let $\mathcal{P}'''$ denote the set of processes such that $p \in \mathcal{P}'''$ if and only if $PC(p) \in \{42-45, 47-50, 53\}$. We let $|L|$ denote the length of L. We let n_i denote the ith element of L, for all $i \in \{0, 1, \ldots, |L|-1\}$. Then, the algorithm in Figure 8 satisfies the following invariants.

Lemma 14 *The algorithm in Figure 8 satisfies the invariants in Figure 10.*

Proof. (By induction) For the base case (i.e., $t = 0$), the lemma holds trivially by initialization. The inductive hypothesis states that the lemma holds at all times prior to $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the lemma holds at time t' as well.

Notice that, if $PC(p) \in \{36, 38-40, 42-44, 46-48, 51, 52, 54-68\}$, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 35$, then p joins $\mathcal{P}'$ and writes into $mybuf(p)$ a pointer to a newly allocated node. Consequently, invariant 8 holds by IH:2, invariant 9 by IH:7, and invariant 7 by definition of *Alive*. All other invariants trivially hold.

If $PC(p) = 37$, then p joins $\mathcal{P}''$ and writes $\perp$ into $mynode(p)\rightarrow$`next`. Hence, we have invariant 10. Furthermore, invariant 5 holds by IH:8. All other invariants trivially hold.

1. $|L| \geq 1$.
2. For any node $n \in L$, we have $n \in Alive$.
3. For any two nodes n_i and n_j such that $i \neq j$, we have $n_i \neq n_j$.
4. `Head` $= n_0$.
5. $*(n_i.\texttt{next}) = n_{i+1}$, for all $i \in \{0, 1, \ldots, |L|-2\}$.
6. $n_{|L|-1}.\texttt{next} = \perp$.
7. For all $p \in \mathcal{P}'$, we have $mynode(p) \in Alive$.
8. For all $p \in \mathcal{P}'$ and all $i \in \{0, 1, \ldots, |L|-1\}$, we have $*mynode(p) \neq n_i$.
9. For all p and q in $\mathcal{P}'$ such that $p \neq q$, we have $mynode(p) \neq mynode(q)$.
10. For all $p \in \mathcal{P}''$, we have $mynode(p)\rightarrow\texttt{next} = \perp$.
11. For all $p \in \mathcal{P}'''$, we have $*cur(p) = n_j$, for some $j \in \{0, 1, \ldots, |L|-1\}$.
12. For any $p \in \mathcal{P}'$ such that $PC(p) = 53$, we have $cur(p)\rightarrow\texttt{next} \neq \perp$.

Figure 10: The invariants satisfied by the algorithm in Figure 8

If $PC(p) = 41$, then p joins $\mathcal{P}'''$ and writes `Head` into $cur(p)$. Consequently, invariant 11 holds by IH:4. All other invariants trivially hold.

If $PC(p) = 45$, then p leaves $\mathcal{P}'$, $\mathcal{P}''$, and $\mathcal{P}'''$, and frees up the node $*\Updownarrow\dagger\backslash\wr\lceil\rceil(p)$. Consequently, invariant 2 holds by IH:8 and invariant 7 by IH:9. All other invariants trivially hold.

If $PC(p) = 49$, or if $PC(p) = 50$ and p's CAS fails, then we have $cur(p)\rightarrow\texttt{next} \neq \perp$ at both times t and t'. Consequently, invariant 12 holds. All other invariants trivially hold.

If $PC(p) = 50$ and p's CAS succeeds, then (1) p leaves $\mathcal{P}'$, $\mathcal{P}''$, and $\mathcal{P}'''$, (2) $*mynode(p)$ joins L, (3) $*cur(p).\texttt{next} = \perp$ at time t, and (4) $*cur(p).\texttt{next} = mynode(p)$ at time t'. Let l be the length of L at time t. Then, by IH:11, IH:5, and IH:6, it follows that $*cur(p) = n_{l-1}$. Therefore, invariant 5 holds. Furthermore, invariant 1 holds by IH:1, invariant 2 by IH:7, invariant 3 by IH:8, invariant 6 by IH:10, invariant 8 by IH:9. All other invariants trivially hold.

If $PC(p) = 53$, then p writes $cur(p)\rightarrow\texttt{next}$ into

$cur(p)$. Consequently, invariant 11 holds by IH:12, IH:5, and IH:6. All other invariants trivially hold. □

Lemma 15 *Let $n \neq n_0$ be any node in L and t be the time when n is installed in L. Let p be the process that installs n in L. Let t_1 (respectively, t_2, t_3, t_4, t_5) be the time when p executes Line 35 (respectively, Line 36, 37, 38, 39). Let B be the memory block that p allocates at time t_4. Then, we have the following: (1) p allocates n at time t_1, (2) $n.\texttt{owned}$ holds value* true *at all times during (t_2, t), (3) $n.\texttt{next}$ holds value $\perp$ at all times during (t_3, t), (4) $n.\texttt{loc}$ holds a pointer to B at all times during (t_4, t), (5) B is not released during (t_4, t), and (6) $B.\texttt{Help}$ holds value $(0, 0, *)$ at all times during (t_5, t).*

Proof. The lemma follows immediately by Invariant 9. □

Lemma 16 *Let n be any node in L and t be the time when n is installed in L. If $n \neq n_0$, let p be the process that installs n in L and $t' < t$ be the latest time prior to t when p executes Line 38. If $n = n_0$, let $t' = 0$. Let B be the memory block allocated at time t'. Then, (1) B is not released (at Line 44) after time t', and (2) $n.\texttt{loc}$ points to B at all times after t'.*

Proof. If $n = n_0$, then the lemma holds immediately by Invariant 8. We now show that the lemma also holds for $n \neq n_0$. Notice that, by Lemma 15, it follows that (1) p allocates n at Line 35 at some time $t'' < t'$, (2) $n.\texttt{loc}$ holds a pointer to B at all times during (t', t), and (3) B is not released during (t', t). Furthermore, by Invariant 8, no process writes into $n.\texttt{loc}$ after time t, and no process releases B after time t. Therefore, we have the lemma. □

Lemma 17 *For any two nodes n_i and n_j in L such that $i \neq j$, we have $n_i.\texttt{loc} \neq n_j.\texttt{loc}$.*

Proof. If $i \neq 0$ (respectively, $j \neq 0$), let p_i (respectively, p_j) be the process that installs n_i (respectively, n_j) in L, t_i (respectively, t_j) be the time when p_i (respectively, p_j) executes Line 38, and B_i (respectively, B_j) be the block that p_i (respectively, p_j) allocates at time t_i (respectively, t_j). If $i = 0$ (respectively, $j = 0$), let $t_i = 0$ (respectively, $t_j = 0$). Then, by Lemma 16, $n_i.\texttt{loc}$ (respectively, $n_j.\texttt{loc}$) has value B_i (respectively, B_j), at all times after t_i (respectively, t_j), and B_i (respectively, B_j) is not released after time t_i (respectively, t_j). Without loss of generality, let $t_i < t_j$. Then, by the uniqueness of allocated addresses, we have $B_j \neq B_i$, which proves the lemma. □

In the following, we let B_i, for all $i \in \{0, 1, \ldots, |L| - 1\}$, denote the memory block pointed by $n_i.\texttt{loc}$.

Lemma 18 *At the time when some process p starts its kth execution of the loop at Line 42 we have (1) $|L| \geq k$, (2) $\text{cur}(p) = n_{k-1}$, and (3) $\text{name}(p) = k - 1$.*

Proof. (By induction) For the base case (i.e., $k = 1$), notice that by Invariant 1, we have $|L| \geq 1$. Furthermore, by Invariant 4, we have $cur(p) = n_0$. Finally, by Line 40, we have $name(p) = 0$. Therefore, the lemma holds for the base case. The inductive hypothesis states that the lemma holds for some $k \geq 1$. We now show that the lemma holds for $k + 1$ as well.

By inductive hypothesis, we have $cur(p) = n_{k-1}$ and $name(p) = k - 1$ when p starts its kth iteration of the loop. Since p increments $name(p)$ at Line 48 during that iteration, it follows that $name(p) = k$ when p starts its $k + 1$st iteration of the loop. Furthermore, by Invariants 12 and 5, it follows that $L \geq k + 1$ and that $cur(p) = n_k$ when p starts its $k + 1$st iteration of the loop. Hence, we have the lemma. □

Definition 1 *If at some time a process p either (1) performs a successful CAS at Line 43 with $\text{cur}(p) = n$ or (2) performs a successful CAS at Line 50 with $\text{mynode}(p) = n$, then we say that p acquires ownership of a node $n \in L$. If i is the index of n in L (i.e., $n = n_i$), then we also say that p acquires ownership of name i and memory block B_i.*

Lemma 19 *If a process p exits the loop at Line 46 during the kth iteration of the loop, then we have (1) $|L| \geq k$, (2) p has ownership of n_{k-1}, (3) $\text{name}(p) = k - 1$, (4) $*\text{cur}(p) = n_{k-1}$, and (5) $*\text{mynode}(p) \neq n_{k-1}$.*

Proof. Claims 1, 2, 3, and 4 follow immediately from Lemma 18. Claim 5 follows immediately by Invariant 8. □

Lemma 20 *If a process p exits the loop at Line 52 during the kth iteration of the loop, then we have (1) $|L| \geq k+1$, (2) p has ownership of n_k, (3) $\text{name}(p) = k$, (4) $*\text{cur}(p) = n_k$, and (5) $*\text{mynode}(p) = n_k$.*

Proof. Notice that, by Lemma 18, when p begins its kth iteration of the loop, we have $|L| \geq k$, $name(p) = k - 1$, and $*cur(p) = n_{k-1}$. Since p's CAS at Line 50

is successful, it follows that $n_{k-1}.\texttt{next} = \perp$ just before that CAS. Hence, by Invariant 6, $|L| = k$ just before p executes Line 50. Consequently, p installs $*mynode(p)$ into the kth position in L, and so we have $*mynode(p) = n_k$ and $|L| = k + 1$ after p's CAS. Therefore, Claims 1, 2, and 5 hold. Furthermore, Claim 3 holds by Line 48 and Claim 4 by Line 51. □

Lemma 21 *If a process p captures a node $n \in L$, then p subsequently satisfies the condition at Line 59 if and only if p captured n at Line 52.*

Proof. The lemma follows immediately by Lemmas 19 and 20. □

Lemma 22 *If a process p acquires ownership of some node $n \in L$, then $*\text{node}_p = n$ at the time when p subsequently executes Line 61.*

Proof. The lemma follows immediately by Lemmas 19 and 20, and Line 58. □

Definition 2 *Let t be the time when p acquires ownership of some node $n \in L$, $t' > t$ be the first time after t when p executes Line 61, and i be the position of n in L (i.e., $n = n_i$). Then, we say that p releases ownership of node n (respectively, name i, memory block B_i) at time t', and that p owns node n (respectively, name i, memory block B_i) at all times during (t, t').*

Lemma 23 *For any node $n \in L$, at most one process owns n.*

Proof. Suppose not. Then, there exists some time such that two or more processes own some node in L. Let t be the earliest such time and n the node in L owned by two processes. Let p and q be those two processes. Without loss of generality, assume that p acquired ownership of n first, at some time $t' < t$. (Notice that, by definition of t, q acquires ownership of n at time t.) Then, by Invariant 3, q acquires ownership of n at Line 43. We examine two possibilities: either p acquires ownership if n at Line 43 or at Line 50. In the first case, p writes *true* into $n.\texttt{owned}$ at time t'. Furthermore, since t is the earliest time that two or more processes own the same node, it follows that during (t', t) no process writes *false* into $n.\texttt{owned}$. Therefore, $n.\texttt{owned} = \textit{true}$ at time t, and so q's CAS at time t fails. This, however, is a contradiction to the fact that q acquires ownership of n at time t.

In the second case (where p acquires ownership at Line 50), it follows by Lemma 15 that $n.\texttt{owned} = \textit{true}$ at time t'. By the same argument as above, $n.\texttt{owned} = \textit{true}$ at time t as well. Therefore, q's CAS at time t fails, which is a contradiction to the fact that q acquires ownership of n at time t. □

Lemma 24 *Let t be the time when some process p starts its kth execution of the loop at Line 44, and $t' < t$ be the latest time prior to t when p executes Line 42. Then, there exists some time $t'' \in (t', t)$ such that the number of nodes in $n_1, n_2, \ldots, n_k$ that are owned by some process at time t'' plus the number of processes (including p) that do not own any nodes but are in their jth execution of the loop at Line 44 at time t'', for $j \leq k$, is at least k.*

Proof. The proof is identical to the proof of Lemma A.4 in [10]. □

Definition 3 *A memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes* active *the first time some process that captures B_i (at Line 52) completes its Join procedure.*

Lemma 25 *At the moment when a memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes active, we have (1) $B_i.N = 1$, (2) $|B_i.BUF| = 2$, (3) $B_i.\text{mybuf} = (1, i, 0)$, (4) $|B_i.Q| = 1$, (5) the value in $B_i.Q$ is $(1, i, 1)$, (6) $B_i.\text{index} = 0$, and (7) $B_i.\text{name} = i$.*

Proof. Let p be the process that first captures B_i (at Line 52). Then, by Lemma 21, p subsequently satisfies the condition at Line 59. Hence, p executes steps at Lines 62–68. Since, by Lemma 21, no other process ever writes into B_i at Lines 62–68, the lemma follows trivially by Lines 62–68. □.

Lemma 26 *After a memory block B_i, $i \in \{0, 1, \ldots, |L| - 1$, becomes active, no process writes into B_i at Lines 62–68.*

Proof. The lemma follows immediately by Lemma 21. □

Lemma 27 *Any process that executes Line 47 during the ith iteration of the loop (at Line 42) writes a pointer to B_{i-1} into the $i - 1$st location of $\texttt{NameArray}$.*

Proof. The lemma follows immediately by Lemma 18. □

Lemma 28 *If a process p exits the loop at Line 43 during the ith iteration of the loop, then p writes B_{i-1} into the $i-1$st location of* `NameArray` *at Line 54. If a process p exits the loop at Line 52 during the ith iteration of the loop, then p writes B_i into the ith location of* `NameArray` *at Line 54.*

Proof. The lemma follows immediately by Lemmas 19 and 20. □

Lemma 29 *The algorithm in Figure 8 writes into array* `NameArray` *in accordance with the specification of the DynamicArray object.*

Proof. Notice that, by Lemma 27, any process that executes Line 47 during the ith iteration of the loop writes the same value (namely, a pointer to B_{i-1}) into the $i-1$st location of `NameArray`. Furthermore, by Lemma 28, if a process exits the loop at Line 43 (respectively, Line 52) during the ith iteration of the loop, then p writes the same value, namely, B_{i-1} (respectively, B_i), into the $i-1$st (respectively, ith) location of `NameArray` at Line 54. Therefore, all values written into the same location are the same, and each process writes into the locations of `NameArray` in order. Hence, we have the lemma. □

Lemma 30 *The value of* `N` *increases with each write into* `N`.

Proof. Suppose not. Let t be the first time that `N` is written and its value does not increase. Let p be the process that performs that write. Then, p's CAS at Line 56 must be of the form `CAS`(N, i, j), where $j \leq i$. This, however, is a contradiction to the fact that p had satisfied the condition at Line 55 prior to this CAS. □

Lemma 31 *At the moment when a process p exits the loop at Line 55, we have* $\text{name}(p) < N$.

Proof. The lemma follows immediately by Line 55. □

Lemma 32 *Any process p completes the loop at Line 55 after at most* $\text{name}(p) + 1$ *iterations.*

Proof. Suppose not. Then, p executes the loop at Line 55 for at least $name(p) + 2$ iterations. Notice that, during the first $name(p) + 1$ iterations, p does not perform a successful CAS at Line 56 (because, by Lemma 30, p would exit the loop right after that CAS). Therefore, the value in `N` changes at least $name(p) + 1$ times during those $name(p) + 1$ iterations. Then, by Lemma 30, it follows that after p executes $name(p) + 1$ iterations of the loop the value of `N` is at least $name(p) + 1$. Therefore, p does not execute the $(name(p) + 2)$nd iteration of the loop, which is a contradiction. □

Lemma 33 *Location i in* `NameArray` *holds value B_i at all times, for all $i < N$.*

Proof. Let j be the value of `N`. If $j = 0$, then the lemma trivially holds. Otherwise, let p be the process that first wrote j into `N`, and t be the time when p performed that write. Then, we have $name(p) = j - 1$ at time t. Hence, p had executed Line 48 exactly $j - 1$ times prior to t. Consequently, by Lemma 27, p had written pointers to memory blocks $B_0, B_1, \ldots, B_{j-2}$ into locations $0, 1, \ldots, j-2$ of `NameArray`, respectively. Furthermore, by Lemma 28, p had written a pointer to B_{j-1} into `NameArray` at Line 54. Thus, we have the lemma. □

Lemma 34 *For any memory block B_i, $i \in \{0, 1, \ldots, |L| - 1\}$, if B_i is active then $i < N$.*

Proof. Let p be the process that first captures B_i (at Line 52). Then, by Lemma 20, we have $name(p) = i$ when p exits the loop (at Line 42). Consequently, by Lemma 31, at the moment when p exits the loop at Line 55, we have $i <$ `N`. Since, by Lemma 30, the value in `N` never decreases, we have the lemma. □

Corollary 1 *For any memory block B_i, $i \in \{0, 1, \ldots, |L| - 1\}$, if B_i is active then location i in* `NameArray` *holds value B_i.*

We let $\mathcal{P}$ denote a set of processes such that $p \in \mathcal{P}$ if and only if $PC(p) \in [1 \ldots 34]$.

Lemma 35 *For any process $p \in \mathcal{P}$, the following holds:*

1. $\text{loc}_p = B_i$, *for some* $i \in 0, 1, \ldots, |L| - 1$.
2. *For all $q \in \mathcal{P}$ such that $p \neq q$, we have* $\text{loc}_p \neq \text{loc}_q$.

Proof. The first claim follows immediately by Lemmas 19 and 20. The second claim follows immediately by the first claim and Lemma 17. □

A.3.2 Lemmas associated with the algorithm in Figure 7

Let $\mathcal{H}$ be any finite execution history of the algorithm in Figures 7 and 8. Let OP be some LL operation, OP′ some SC operation, and OP″ some VL operation on $\mathcal{O}[i]$ in E, for some i. Then, we define the linearization points (LPs) for OP, OP′, and OP″ as follows. If the CAS at Line 5 of OP succeeds, then LP(OP) is Line 3 of OP. Otherwise, let t be the time when OP executes Line 2, and t' be the time when OP performs the CAS at Line 5. Let v be the value that OP reads at Line 9 of OP. Then, we show that there exists a successful SC operation SC_q on $\mathcal{O}[i]$ such that (1) at some point t'' during (t, t'), SC_q is the latest successful SC on $\mathcal{O}[i]$ to execute Line 16, and (2) SC_q writes v into $\mathcal{O}[i]$. We then set LP(OP) to time t''. We set LP(OP′) to Line 16 of OP′, and LP(OP″) to Line 14 of OP″.

Lemma 36 *Let B_i, for $i \in \{0, 1, \ldots, |L| - 1$ be any memory block. Let t be the time when B_i is installed in L. Let t' be the end of $\mathcal{H}$. Let $LL_1, LL_2, \ldots, LL_m$ be the sequence of all LL operations in $\mathcal{H}$ such that, if a process p is executing LL_j, then $\text{loc}_p = B_i$ during LL_j, for all $j \in \{1, 2, \ldots, m\}$. Let t_j and t'_j, for all $j \in \{1, 2, \ldots, m\}$, be the times when Lines 2 and 5 of LL_j are executed. Then, the following statements hold:*

(S1) During the time interval $(t_j, t'_j]$, for all $j \in \{1, 2, \ldots, m\}$, exactly one *write into B_i.Help is performed.*

(S2) Any value written into B_i.Help during $(t_j, t'_j]$, for all $j \in \{1, 2, \ldots, m\}$, is of the form $(*, 0, *)$.

(S3) During the time intervals $(t, t_1), (t'_1, t_2), (t'_2, t_3), \ldots, (t'_{m-1}, t_m), (t'_m, t')$, the value in B_i.Help is of the form $(*, 0, *)$ and doesn't change.

Proof. Let j be any index in $\{1, 2, \ldots, m\}$. Statement (S2) follows trivially from the fact that the only two operations that can affect the value of B_i.Help during (t_j, t'_j) are (1) the CAS at Line 5 of LL_j, and (2) the CAS at Line 31 of some other process' SC operation, both of which attempt to write $(*, 0, *)$ into B_i.Help.

We now prove statement (S1). Suppose that (S1) does not hold. Then, during $(t_j, t'_j]$, either (1) two or more writes on B_i.Help are performed, or (2) no writes on B_i.Help are performed. In the first case, we know (by an earlier argument) that each write on B_i.Help during $(t_j, t'_j]$ is performed either by the CAS at Line 5 of LL_j, or by the CAS at Line 31 of some other process' SC operation. Let CAS_1 and CAS_2 be the first two CAS operations on B_i.Help to write into B_i.Help during $(t_j, t'_j]$. Then, by the algorithm, both CAS_1 and CAS_2 are of the form $\text{CAS}(B_i.\text{Help}, (*, 1, *), (*, 0, *))$. Since CAS_1 succeeds and B_i.Help doesn't change between CAS_1 and CAS_2, it follows that CAS_2 fails, which is a contradiction.

In the second case (where no writes on B_i.Help take place during $(t_j, t'_j]$), B_i.Help doesn't change throughout $(t_j, t'_j]$. Therefore, the CAS at Line 5 of LL_j succeeds, which is a contradiction to the fact that no writes on B_i.Help take place during $(t_j, t'_j]$. Hence, statement (S1) holds.

We now prove statement (S3). Suppose that (S3) statement doesn't hold. Let (t'', t''') be any of the intervals $(t, t_1), (t'_1, t_2), (t'_2, t_3), \ldots, (t'_{m-1}, t_m), (t'_m, t')$ during which the statement doesn't hold. Notice that, by Lemma 15, $B_i.\text{Help} = (*, 0, *)$ at time t. Furthermore, by statements (S1) and (S2), $B_i.\text{Help} = (*, 0, *)$ at times $t'_1, t'_2, \ldots, t'_m$. Hence, $B_i.\text{Help} = (*, 0, *)$ at time t''. Let CAS_3 be the first CAS operation on B_i.Help to write into B_i.Help during (t'', t'''). Then, by the algorithm, CAS_3 is of the form $\text{CAS}(B_i.\text{Help}, (*, 1, *), (*, 0, *))$. Since B_i.Help doesn't change between time t'' and CAS_3, it follows that CAS_3 fails, which is a contradiction. Hence, we have statement (S3). □

In Figure 11 we present a number of invariants that the algorithm satisfies. In the following, we let $PC(p)$ denote the value of process p's program counter. Without loss of generality, we assume that when a process completes any of the procedures its program counter immediately jumps to the start of the next procedure it wishes to execute. For any register r at process p, we let $r(p)$ denote the value of that register. We let $\mathcal{P}_1$ denote a set of processes such that $p \in \mathcal{P}_1$ if and only if $PC(p) \in \{1, 2, 7 - 17, 24 - 31, 33, 34\}$ or $PC(p) \in \{3 - 5\} \wedge loc_p.\text{Help} \equiv (*, 1, *)$. We let $\mathcal{P}_2$ denote a set of processes such that $p \in \mathcal{P}_2$ if and only if $PC(p) \in \{3 - 6\} \wedge loc_p.\text{Help} \equiv (*, 0, *)$. We let $\mathcal{P}_3$ denote a set of processes such that $p \in \mathcal{P}_3$ if and only if $PC(p) = 18$. We let $\mathcal{P}_4$ denote a set of processes such that $p \in \mathcal{P}_4$ if and only if $PC(p) = 32$. We let $\mathcal{B}_1$ (respectively, $\mathcal{B}_2$, $\mathcal{B}_3$) denote a set of memory blocks such that $B \in \mathcal{B}_1$ (respectively, $\mathcal{B}_2$, $\mathcal{B}_3$) if and only if (1) $B \in \mathcal{B}$, and (2) there exists some process p such that $loc_p = B$ and $p \in \mathcal{P}_1$ (respectively, $\mathcal{P}_2$, $\mathcal{P}_3$). We let $\mathcal{B}_0$ denote a set of memory blocks such that $B \in \mathcal{B}_0$ if and only if (1) $B \in \mathcal{B}$, and (2) there does not exist any process $p \in \mathcal{P}$ such that $loc_p = B$. Finally, for any buffer $B \in \mathcal{B}$, we let $|B.Q|$ denote the length of the queue $B.Q$,

1. For any memory block $B \in \mathcal{B}$, we have $|B.Q| \geq B.N$.
2. For any memory block $B \in \mathcal{B}$, we have $|B.\texttt{BUF}| \leq B.N + 1$.
3. For any process $p \in \mathcal{P}$ such that $PC(p) \in \{19, 20, 23\}$, we have $|loc_p.Q| \geq loc_p.N + 1$.
4. For any process $p \in \mathcal{P}$ such that $PC(p) = 21$, we have $|loc_p.\texttt{BUF}| \leq loc_p.N$.
5. Let B^0 (respectively, B^1, B^2, B^3) be any memory block in $\mathcal{B}_0$ (respectively, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$). Let B be any memory block in $\mathcal{B}$, and b be any value in $B.Q$. Let p_4 be any process in $\mathcal{P}_4$. Let i be any index in $[0\,..\,M-1]$. If b (respectively, $B^0.mybuf$, $B^1.mybuf$, $B^2.\texttt{Help}.buf$, $B^3.x.buf$, $c(p_4)$, $\texttt{X}[j].buf$) is of the form $(0, j)$, then we have $j \in [0\,..\,M-1]$. If b (respectively, $B^0.mybuf$, $B^1.mybuf$, $B^2.\texttt{Help}.buf$, $B^3.x.buf$, $c(p_4)$, $\texttt{X}[j].buf$) is of the form $(1, n, j)$, then we have (1) $B_n \in \mathcal{B}$, (2) the jth location in array $B_n.\texttt{BUF}$ holds a pointer to a buffer of length $W + 1$, and (3) there does not exist any process $p \in \mathcal{P}$ such that $loc_p = B_n$, $PC(p) = 22$, and $loc_p.N = j$.
6. Let B^0 and C^0 (respectively, B^1 and C^1, B^2 and C^2, B^3 and C^3) be any two memory blocks in $\mathcal{B}_0$ (respectively, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$). Let B be any memory block in $\mathcal{B}$, and b_1 and b_2 any two values in $B.Q$. Let p_4 and q_4 be any two processes in $\mathcal{P}_4$. Let i and j be any two indices in $[0\,..\,M-1]$. Then, we have $B^0.mybuf \neq C^0.mybuf \neq B^1.mybuf \neq C^1.mybuf \neq b_1 \neq b_2 \neq \texttt{X}[i].buf \neq \texttt{X}[j].buf \neq B^2.\texttt{Help}.buf \neq C^2.\texttt{Help}.buf \neq B^3.x.buf \neq C^3.x.buf \neq c(p_4) \neq c(q_4)$.
7. For any memory block $B \in \mathcal{B}$, we have $0 < B.N \leq \texttt{N}$.
8. For any process $p \in \mathcal{P}$ such that $PC(p) = 20$, we have $0 < loc_p.N < \texttt{N}$.
9. For any memory block $B \in \mathcal{B}$, we have $B.index < B.N$.

Figure 11: The invariants satisfied by the algorithm in Figure 7

and $|B.\texttt{BUF}|$ denote the length of the array $B.\texttt{BUF}$.

Lemma 37 *The algorithm in Figures 7 and 8 satisfies the invariants in Figure 11.*

Proof. For the base case, (i.e., $t = 0$), all the invariants hold by initialization. The inductive hypothesis states that the invariants hold at time $t \geq 0$. Let t' be the earliest time after t that some process, say p, makes a step. Then, we show that the invariants holds at time t' as well.

First, notice that if $PC(p) = \{1-4, 7-9, 11-13, 15, 17, 24-30, 35-58, 60-67\}$, or if $PC(p) = \{5, 16, 31\}$ and p's CAS fails, then none of the invariants are affected by p's step and hence they hold at time t' as well. In the following, let $B = loc_p$.

If $PC(p) = 5$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_2$ and p writes $B.mybuf$ into $B.\texttt{Help}.buf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 6$, then, by Lemma 36, B was in $\mathcal{B}_2$ at time t. Furthermore, B is in $\mathcal{B}_1$ at time t'. Since p writes $B.\texttt{Help}.buf$ into $B.mybuf$, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) \in \{10, 14, 17, 34\}$, then B moves either from $\mathcal{B}_1$ to $\mathcal{B}_1$ (if the next operation that p executes is LL, VL, or SC), or from $\mathcal{B}_1$ to $\mathcal{B}_0$ (if the next operation that p executes is Leave). In either case, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 16$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_3$. Let $\texttt{X}[i]$ be the variable that p writes to. Then, since p's CAS is successful, we have $\texttt{X}[i].buf = B.x.buf$ at time t, and $\texttt{X}[i].buf = B.mybuf$ at time t'. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 18$, then B leaves $\mathcal{B}_3$. Furthermore, p enqueues $B.x.buf$ into the queue $B.Q$. Consequently, invariant 1 holds by IH:1, invariant 3 by IH:1, invariant 5 by IH:5, and invariant 6 by IH:6. The other two invariants trivially hold.

If $PC(p) = 19$, then we have $B.N < \texttt{N}$. Consequently, invariant 8 holds by IH:7. All other invariants trivially hold.

If $PC(p) = 20$, then p increments $B.N$ by one. Consequently, invariant 1 holds by IH:3, invariant 4 by IH:2, and invariant 7 by IH:8. All other invariants trivially hold.

If $PC(p) = 21$, then the length of $B.\texttt{BUF}$ increases to at least $B.N + 1$. Consequently, invariant 2 holds

by IH:4. Notice that, by IH:4, the length of B.BUF was at most $B.N$ prior to this step. Therefore, location $B.N$ does not hold a pointer to a buffer of length $W+1$. Consequently, by IH:5, none of the values appearing in the inequality of invariant 6 were of the form $(1, B.name, B.N)$ prior to this step. Hence, invariant 5 holds. All other invariants trivially hold.

If $PC(p) = 22$, then B joins $\mathcal{B}_1$. Furthermore, p writes $(1, B.name, B.N)$ into $B.mybuf$. Notice that, by IH:5, none of the values appearing in the inequality of invariant 6 were of the form $(1, B.name, B.N)$ prior to this step. Consequently, invariant 6 holds. Invariant 5 holds by the fact that p had written a buffer of length $W+1$ into location $B.N$ of B.BUF at Line 21. All other invariants trivially hold.

If $PC(p) = 23$, then B joins $\mathcal{B}_1$. Furthermore, p reads and dequeues the front element of the queue $B.Q$. Consequently, invariant 1 holds by IH:3, invariant 5 by IH:5, and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 31$ and p's CAS succeeds, then B moves from $\mathcal{B}_1$ to $\mathcal{B}_4$. Let B' be the memory block pointed to by $l(p)$. Then, we have $c(p) = B'$.Help.*buf* at time t, B'.Help.*buf* $= B.mybuf$ at time t', B'.Help changes from value $(*, 1, *)$ at time t to a value $(*, 0, *)$ at time t'. Therefore, by Lemma 36, there exists some process q such that (1) $loc_q = B'$ and $PC(q) \in \{3-5\}$ at times t and t', and (2) B'.Help.*buf* $= B'.mybuf$ at time t. Hence, B' moves from $\mathcal{B}_1$ to $\mathcal{B}_2$, and $c(p) = B'.mybuf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 32$, then B moves from $\mathcal{B}_4$ to $\mathcal{B}_1$. Furthermore, p writes $c(p)$ into $B.mybuf$. Consequently, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold. □

If $PC(p) = 59$ and p does not satisfy the condition at Line 59, then, by Lemma 21, B moves either from $\mathcal{B}_0$ to $\mathcal{B}_1$ (if the next operation that p executes is LL), or from $\mathcal{B}_0$ to $\mathcal{B}_0$ (if the next operation that p executes is Leave). In the latter case, none of the invariants are affected by p's step and hence they hold at time t'. In the former case, invariant 5 holds by IH:5 and invariant 6 by IH:6. All other invariants trivially hold.

If $PC(p) = 59$ and p satisfies the condition at Line 59, then none of the invariants are affected by p's step and hence they hold at time t' as well.

If $PC(p) = 68$, then, by Lemma 21, B joins $\mathcal{B}$. Furthermore, B either joins $\mathcal{B}_1$ (if the next operation that p executes is LL), or $\mathcal{B}_0$ (if the next operation that p executes is Leave). Let i be the name that p captures during the Join procedure. Then, we have $B = B_i$. Notice that, by Lemma 25, we have invariants 1, 2, 5, and 9. Furthermore, by IH:5, all the values in the inequality of invariant 6 are either of the form $(0, *)$ or $(1, n, *)$, for some $B_n \in \mathcal{B}$. Since B has just joined $\mathcal{B}$ and since all memory blocks in $\mathcal{B}$ are different (by Lemma 17), it follows that $n \neq i$. Hence, by Lemma 25, we have invariant 6. Invariant 7 follows immediately by Lemmas 25 and 34. All other invariants trivially hold. □

Lemma 38 *The algorithm in Figure 7 reads array* NameArray *in accordance with the specification of the DynamicArray object.*

Proof. Let p be any process. Then, the only two places where p reads NameArray is at Lines 12 and 24 of the algorithm. If p reads NameArray at Line 12, let i be the location in NameArray that p reads. Then, by Invariant 5, it follows that $B_i \in \mathcal{B}$. Consequently, by Corollary 1, location i of NameArray had already been written, which means that the lemma holds.

If p reads NameArray at Line 24, let j be the location in NameArray that p reads. Then, by Invariants 9 and 7, it follows that $i <$ N. Consequently, by Lemma 33, location i of NameArray had already been written, which means that the lemma holds. □

Lemma 39 *Let B_i, for $i \in \{0, 1, \ldots, |L|-1\}$ be any memory block. Then, the algorithms in Figures 7 and 8 read and write into the array B_i.BUF in accordance with the specification of the DynamicArray object.*

Proof. Let p be the first process that captures ownership of B_i. Then, by Lemma 21, p is the only process that writes into B_i.BUF at Lines 63 and 64, and that p writes into locations 0 and 1 of B_i.BUF. Notice that, by Lemma 25, $B_i.N = 1$ when B_i becomes active. Since B_i.BUF is written at Line 21 only after $B_i.N$ is incremented at Line 20, it follows that locations $2, 3, \ldots$ are written in order. Hence, the writes into B_i.BUF are in accordance with the specification of the DynamicArray object.

Let q be any process that reads some location j of array B_i.BUF at Line 13. Then, by Invariant 5, we have (1) $B_i \in \mathcal{B}$ and (2) the jth location in B_i.BUF had already been written. Consequently, the lemma holds. □

Lemma 40 *Let $t_0 < t_1 < \ldots < t_K$ be all the times in $\mathcal{H}$ when some variable $X[i]$ is written to (by a successful CAS at Line 16). Then, for all $j \in \{0, 1, \ldots, K\}$, the value written into $X[i]$ at time t_j is of the form $(j, *)$.*

Proof. Suppose not. Let j be the smallest index such that, at time t_j, a value $k \neq j$ is written into $\mathtt{X}[i]$ by some process p. (By initialization, we have $j \geq 1$.) Then, by the algorithm, p's CAS at time t_j is of the form $\mathtt{CAS}(\mathtt{X}[i], (k-1, *), (k, *))$. Since $\mathtt{X}[i]$ holds value $j-1$ at time t_j, and since $k \neq j$, it follows that p's CAS fails, which is a contradiction to the fact that p writes into $\mathtt{X}[i]$ at time t_j. □

Lemma 41 *Let $\mathcal{O}[i]$ be an LL/SC object. Let t be the time when some process p reads $\mathtt{X}[i]$ (at Line 3 or 27), and $t' > t$ the first time after t that p completes Line 4 or Line 29. Let* OP *be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line* 16 *prior to time t, and v the value that* OP *writes in $\mathcal{O}[i]$. If there exists some process $q \in \mathcal{P}$ such that $\mathrm{loc}_q.\mathtt{Help}$ holds value $(*, 1, *)$ throughout (t, t') and doesn't change, then p reads value v at Line 4 or Line 29 (during (t, t')).*

Proof. Let r be the process executing OP. Let B be the buffer that q owns (i.e., $B = loc_q$). Since OP is the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 16 prior to time t, it follows that p reads from $\mathtt{X}[i]$ at time t the value that r writes in $\mathtt{X}[i]$ at Line 16 of OP. Therefore, p reads during (t, t') the same buffer b that r wrote v into at Line 15 of OP. Let t_1 be the time when r starts writing into b at Line 15 of OP, t_2 the time when r completes writing into b at Line 15 of OP, t_3 the time when r writes into $\mathtt{X}[i]$ at Line 16 of OP, and t'' the time when p starts reading b during (t, t'). Then, the following claim holds.

Claim 14 *During (t_1, t_2), no process other than r writes into b. During (t_2, t'), no process writes into b.*

Proof. Suppose not. Then, either some process other than r writes into b during (t_1, t_2), or some process writes into b during (t_2, t'). In the first case, let r_1 be the process that writes into b during (t_1, t_2). Then, at some point during (t_1, t_2), we have $loc_{r_1}.mybuf = loc_r.mybuf$, which is a contradiction to Invariant 6. In the second case, let r_2 be the first process to start writing into b at some time $\tau_1 \in (t_2, t')$, and k be the index of buffer b. Then, by an earlier argument, $\tau_1 \notin (t_2, t_3)$. Furthermore, by Invariant 6 , r_2 does not write into b as long as $\mathtt{X}[i]$ holds value $(*, k)$. Therefore, $\mathtt{X}[i]$ changes during (t_3, τ_1).

Since $\mathtt{X}[i]$ doesn't change during (t_3, t), it means that (1) $\tau_1 > t$ and (2) some process writes into $\mathtt{X}[i]$ during (t, τ_1). Let r_3 be the first such process, $\tau_2 \in (t, \tau_1)$ the time when r_3 writes into $\mathtt{X}[i]$, SC_{r_3} the SC operation during which r_3 performs that write, and B' the memory block that r_3 owns during SC_{r_3} (i.e., $B' = loc_{r_3}$). Let τ_3 be the time when r_3 executes Line 18 of SC_{r_3}. Then, at time τ_3, r_3 enqueues k into $B'.Q$. Furthermore, by Invariant 6, r_2 does not write into b during (τ_2, τ_3), nor does it write into b during the time $B.Q$ contains value k. Therefore, we have $\tau_3 \in (\tau_2, \tau_1)$. Finally, we know that k is dequeued from $B'.Q$ during (τ_3, τ_1).

Let τ_4 be the first time after τ_3 that k is dequeued from $B'.Q$. (Notice that, by the above argument, $\tau_4 \in (\tau_3, \tau_1)$.) Then, we have the following subclaim.

Subclaim 1 *There exists some process r_5 and an execution E of Lines 24–32 by r_5 such that (1) E takes place during (τ_3, τ_4), (2) $\mathrm{loc}_{r_5} = B'$ during E, and (3) $B'.\mathrm{index} = B.\mathrm{name}$ during E.*

Proof. Let n, m, and j denote the values of $B'.N$, shared variable $\mathtt{N}$, and $B'.index$, respectively, at time τ_3. Then, by Invariant 1, there are n items already in $B'.Q$ before b's index is inserted at time τ_3. So, b is not dequeued until at least $n+1$ dequeues are performed on $B'.Q$. Notice that, each time some process $r_6 \in \mathcal{P}$, such that $loc_{r_6} = B'$, satisfies the condition at Line 19, the following holds: (1) r_6 does not dequeue an element from $B'.Q$, and (2) the values of $B'.N$ and $B'.index$ both increase by one (at Lines 20 and 33). Moreover, each time r_6 does not satisfy the condition at Line 19, the following holds: (1) r_6 dequeues an element from $B'.Q$, and (2) the value of $B'.N$ remains the same and $B'.index$ increases by one modulo $B'.N$ (at Line 33). As a result of the above two facts, the value of $B'.index$ wraps around to 0 (at Line 33) after exactly $n - j$ elements are dequeued from $B'.Q$. Let $\tau_5 > \tau_3$ be the first time after τ_3 when $B'index$ wraps around to 0, and let n' be the value of $B'.N$ at time τ_5. Notice that, since b is not dequeued until at least $n+1$ dequeues are performed on $B'.Q$, we have $\tau_5 \in (\tau_3, \tau_4)$. By the same argument as above, at most j elements are dequeued from $B'.Q$ before $B'.index$ again reaches value j (at Line 33). Therefore, during (τ_3, τ_5), variable $B'.index$ has gone through the values $j, j+1, \ldots, n'-1, 0, 1, \ldots j-1, j$. Since B has become active prior to time τ_3, it follows by Lemma 34 that $B.name < n'$. Therefore, there exists some process r_5 and an execution E of Lines 24–32 by r_5 such that (1) E takes place during (τ_3, τ_5), (2) $loc_{r_5} = B'$ during E, and (3) $B'.index = B.name$ during E. Hence, we have the subclaim. □

Since $B.\mathtt{Help}$ holds value $(*, 1, *)$ throughout (t, t') and doesn't change, it follows that (1) r_5 reads B at Line 24 of E (by Lemma 1), (2) r_5 satisfies the condition at Line 25 of E, and (3) r_5's CAS at Line 31 of E succeeds. This, however, is a contradiction to the fact

that B.Help $= (*, 1, *)$ throughout (t, t'). Hence, we have the claim. □

The above claim shows that (1) during (t_1, t_2), no process other than r writes into b, and (2) during (t_2, t'), no process writes into b. Consequently, p reads v from b during (t, t'), which proves the lemma. □

Lemma 42 *Let $\mathcal{O}[i]$ be an LL/SC object and* OP *some LL operation on $\mathcal{O}[i]$. Let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line* 16 *prior to Line* 3 *of* OP*, and v_q the value that SC_q writes in $\mathcal{O}[i]$. If the CAS at Line* 5 *of* OP *succeeds, then* OP *returns value v_q.*

Proof. Let p be the process executing OP. Let t time when p executes Line 3 of OP, and $t' > t$ be the time when p completes Line 4 of OP. Since the CAS at Line 5 of OP succeeds, it follows by Lemma 36 that loc_p.Help holds value $(*, 1, *)$ throughout (t, t') and doesn't change during that time. Therefore, by Lemma 41, p reads v_q at Line 4 of OP, which proves the lemma. □

Lemma 43 *Let $\mathcal{O}[i]$ be an LL/SC object, and* OP *an LL operation on $\mathcal{O}[i]$ such that the CAS at Line* 5 *of* OP *fails. Let p be the process executing* OP*. Let t and t' be the times, respectively, when p executes Lines* 2 *and* 5 *of* OP*. Let x and v be the values that p reads at Lines 8 and 9 of* OP*, respectively. Then, there exists a successful SC operation SC_q on $\mathcal{O}[i]$ such that (1) at some point during (t, t'), SC_q is the latest successful SC on $\mathcal{O}[i]$ to execute Line* 16*, and (2) SC_q writes x into* X$[i]$ *and v into $\mathcal{O}[i]$.*

Proof. Since p's CAS at time t' fails, it means that loc_p.Help $= (s, 0, k)$ just prior to t'. Then, by Lemma 36, there exists a single process r that writes into loc_p.Help during (t, t') (at Line 31). Let $t_1 \in (t, t')$ be the time when r performs that write, and E be r's execution of Lines 24–32 during which r performs that write. Then, r's CAS at Line 31 of E (at time t_1) is of the form CAS(loc_p.Help, $(s, 1, *)$, $(s, 0, *)$), for some $s > 1$. Therefore, at time t_1, loc_p.Help has value $(s, 1, *)$. Hence, by Lemma 36, p writes $(s, 1, *)$ into loc_p.Help at Line 2 of OP (at time t). Since a value of the form $(s, *, *)$ is written into loc_p.Help for the first time at time t, it follows that r reads $(s, 1, *)$ from loc_p.Help at Line 25 of E at some time $t_2 \in (t, t_1)$. Consequently, r reads variable loc_p.Announce at Line 26 of E at some time $t_3 \in (t_2, t_1)$. Since p writes i into loc_p.Announce at Line 1 of OP, it follows that r reads i from loc_p.Announce at time t_2. Hence, r reads X$[i]$ at Line 27 of E.

Let t_4 be the time when r reads X$[i]$ at Line 27 of E, t_5 the time when r starts Line 29 of E, t_6 the time when r completes Line 29 of E, and t_7 the time when r executes Line 30 of E. Let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 16 prior to time t_4, x_q the value that SC_q writes in X$[i]$, and v_q the value that SC_q writes in $\mathcal{O}[i]$. Then, at time t_4, r reads x_q from X$[i]$. Furthermore, since t_1 is the first (and only) time that loc_p.Help is written during (t, t'), it follows that loc_p.Help holds value $(*, 1, *)$ at all times during (t_4, t_6) and doesn't change during that time. Therefore, by Lemma 41, r reads v_q at Line 29.

Let d be the buffer that r writes v_q into during (t_5, t_6). Then, at time t_7, r writes x_q into $d[W]$. Furthermore, since r writes the index of buffer d into loc_p.Help at Line 31 of E (at time t_1), it follows that p reads buffer d at Lines 8 and 9 of OP. Let t_8 be the time when p reads $d[W]$ at Line 8 of OP, t_9 the time when p starts reading d at Line 8 of OP, and t_{10} the time when p completes reading d at Line 8 of OP. Then, we show that the following claim holds.

Claim 15 *During (t_5, t_6), no process other than r writes into d, and during (t_6, t_{10}), no process writes into d.*

Proof. Suppose not. Then, either some process other than r writes into d during (t_5, t_6), or some process writes into d during $(t_6, t_1 0)$. In the former case, let r_1 be the process that writes into d during (t_5, t_6). Then, at some point during (t_5, t_6), we have $loc_{r_1}.mybuf = loc_r.mybuf$, which is a contradiction to Invariant 6. In the latter case, let r_2 be the first process to write into d at some time $\tau_1 \in (t_6, t_{10})$. Then, by an earlier argument, we know that $\tau_1 \notin (t_6, t_1)$. We now show that $\tau_1 \notin (t_1, t_{10})$.

Let k' be the index of buffer d. We know by Invariant 6 that r_2 does not write into d as long as (1) loc_p.Help $= (s, 0, k')$, and (2) p is between Lines 2 and 6 of OP. Furthermore, since p sets $loc_p.mybuf$ to k' at Line 6 of OP, r_2 does not write into d after p executes Line 6 of OP and before it completes OP. Therefore, throughout (t_1, t_{10}), r_2 does not write into d. Hence, $\tau_1 \notin (t_1, t_{10})$. Since, by an earlier argument, $\tau_1 \notin (t_6, t_1)$, it follows that $\tau_1 \notin (t_6, t_{10})$. This, however, is a contradiction to the fact that r_2 writes into d during (t_6, t_{10}). □

The above claim shows that (1) during (t_5, t_6), no process other than r writes into d, and (2) during (t_6, t_{10}), no process writes into d. Consequently, p reads x_q from $d[W]$ at time t_8 and v_q from d during (t_9, t_{10}). Since SC_q

is the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 16 prior to time t_4, and since $t_4 \in (t, t')$, we have the lemma. □

Lemma 44 (Correctness of LL) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any LL operation on $\mathcal{O}[i]$, and* OP′ *be the latest successful SC operation on $\mathcal{O}[i]$ such that LP(*OP′*) < LP(*OP*). Then,* OP *returns the value written by* OP′.

Proof. Let p be the process executing OP. We examine the following two cases: (1) the CAS at Line 5 of OP succeeds, and (2) the CAS at Line 5 of OP fails. In the first case, let SC_q be the latest successful SC operation on $\mathcal{O}[i]$ to execute Line 16 prior to Line 3 of OP, and v_q be the value that SC_q writes in $\mathcal{O}[i]$. Since all SC operations are linearized at Line 16 and since OP is linearized at Line 3, we have $\mathrm{SC}_q =$ OP′. Furthermore, by Lemma 42, OP returns value v_q. Therefore, the lemma holds in this case.

In the second case, let t and t' be the times, respectively, when p executes Lines 2 and 5 of OP. Let v be the value that p reads at Line 9 of OP. Then, by Lemma 43, there exists a successful SC operation SC_r on $\mathcal{O}[i]$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}[i]$ to execute Line 16, and (2) SC_r writes v into $\mathcal{O}[i]$. Since all SC operations are linearized at Line 16 and since OP is linearized at time t'', we have $\mathrm{SC}_r =$ OP′. Therefore, the lemma holds. □

Lemma 45 (Correctness of SC) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any SC operation on $\mathcal{O}[i]$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}[i]$ by p prior to* OP. *Then,* OP *succeeds if and only if there does not exist any successful SC operation* OP″ *on $\mathcal{O}[i]$ such that LP(*OP′*) < LP(*OP″*) < LP(*OP*).*

Proof. We examine the following two cases: (1) the CAS at Line 5 of OP′ succeeds, and (2) the CAS at Line 5 of OP′ fails. In the first case, let t_1 be the time when p executes Line 3 of OP′, and t_2 be the time when p executes Line 16 of OP. Then, we show that the following claim holds.

Claim 16 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}[i]$ that performs a successful CAS at Line 16 during (t_1, t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}[i]$ performs a successful CAS at Line 16 during (t_1, t_2). Then, $\mathrm{X}[i]$ doesn't change during (t_1, t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}[i]$ does perform a successful CAS at Line 16 during (t_1, t_2). Then, by Lemma 40, $\mathrm{X}[i]$ holds different values at times t_1 and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 16 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}[i]$ such that LP(OP′) < LP(OP″) < LP(OP). Hence, the lemma holds in this case.

In the second case (when the CAS at Line 5 of OP′ fails), let t and t' be the times when p executes Lines 2 and 5 of OP′, respectively. Let x and v be the values that p reads at Lines 8 and 9 of OP′, respectively. Then, by Lemma 43, there exists a successful SC operation SC_r on $\mathcal{O}[i]$ such that (1) at some time $t'' \in (t, t')$, SC_r is the latest successful SC on $\mathcal{O}[i]$ to execute Line 16, and (2) SC_r writes x into $\mathrm{X}[i]$ and v into $\mathcal{O}[i]$. Therefore, at Line 8 of OP′, p reads the value that variable $\mathrm{X}[i]$ holds at time t''. We now prove the following claim.

Claim 17 *Process p's CAS at time t_2 succeeds if and only if there does not exist some other SC operation on $\mathcal{O}[i]$ that performs a successful CAS at Line 16 during (t'', t_2).*

Proof. Suppose that no other SC operation on $\mathcal{O}[i]$ performs a successful CAS at Line 16 during (t'', t_2). Then, $\mathrm{X}[i]$ doesn't change during (t'', t_2), and hence p's CAS at time t_2 succeeds.

Suppose that some SC operation SC_q on $\mathcal{O}[i]$ does perform a successful CAS at Line 16 during (t'', t_2). Then, by Lemma 40, $\mathrm{X}[i]$ holds different values at times t'' and t_2. Hence, p's CAS at time t_2 fails, which proves the claim. □

Since all SC operations are linearized at Line 16 and since OP′ is linearized at time t_1, it follows from the above claim that OP succeeds if and only if there does not exist some successful SC operation OP″ on $\mathcal{O}[i]$ such that LP(OP′) < LP(OP″) < LP(OP). Hence, the lemma holds. □

Lemma 46 (Correctness of VL) *Let $\mathcal{O}[i]$ be some LL/SC object. Let* OP *be any VL operation on $\mathcal{O}[i]$ by some process p, and* OP′ *be the latest LL operation on $\mathcal{O}[i]$ by p that precedes* OP. *Then,* OP *returns true if and only if there does not exist some successful SC operation* OP″ *on $\mathcal{O}[i]$ such that LP(*OP″*) ∈ (LP(*OP′*), LP(*OP*)).*

Proof. Similar to the proof of Lemma 45. □

Theorem 3 *The wait-free implementation in Figures 7 and 8 of an array $\mathcal{O}[0..M-1]$ of M W-word LL/SC objects is linearizable. The time complexity of LL, SC and VL operations on some variable in $\mathcal{O}$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The time complexity of Join and Leave operations is $O(K)$ and $O(1)$, respectively, where K is the maximum number of processes that have simultaneously participated in the algorithm. The space complexity of the implementation is $O((K^2+M)W)$.*

Proof. The theorem follows immediately from Lemmas 44, 45, and 46. □