Efficient Wait-Free Implementation of Multiword LL/SC Variables

Prasad Jayanti  
*Dartmouth College*

Srdjan Petrovic  
*Dartmouth College*

Follow this and additional works at: [https://digitalcommons.dartmouth.edu/cs_tr](https://digitalcommons.dartmouth.edu/cs_tr)

Part of the [Computer Sciences Commons](https://digitalcommons.dartmouth.edu/cs_tr)

**Dartmouth Digital Commons Citation**


This Technical Report is brought to you for free and open access by the Computer Science at Dartmouth Digital Commons. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Efficient Wait-Free Implementation of Multiword LL/SC Variables

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

Abstract

Since the design of lock-free data structures often poses a formidable intellectual challenge, researchers are constantly in search of abstractions and primitives that simplify this design. The multiword LL/SC object is such a primitive: many existing algorithms are based on this primitive, including the nonblocking and wait-free universal constructions [1], the closed objects construction [4] and the snapshot algorithms [12, 13].

In this paper, we consider the problem of implementing a $W$-word LL/SC object shared by $N$ processes. The previous best algorithm, due to Anderson and Moir [1], is time optimal (LL and SC operations run in $O(W)$ time), but has a space complexity of $O(N^2W)$. We present an algorithm that uses novel buffer management ideas to cut down the space complexity by a factor of $N$ to $O(NW)$, while still being time optimal.
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. Lock-based implementations, however, have several shortcomings: they impose waiting, limit parallelism, suffer from convoying, priority inversion and deadlocks, and are not fault-tolerant. Lock-free implementations, classified as wait-free and nonblocking, were proposed to overcome these drawbacks [8, 9, 15, 18]. A wait-free implementation of a shared object $O$ guarantees that every process $p$ completes its operation on $O$ in a bounded number of its steps, regardless of whether other processes are slow, fast or have crashed. A nonblocking implementation extends a weaker guarantee that some operation (not necessarily $p$’s) completes in a bounded number of $p$’s steps.

It is a well understood fact that whether lock-free data structures can be efficiently designed depends crucially on what synchronization instructions are supported by the hardware. After more than two decades of experience with different instructions (including test&set, swap and fetch&add), there is growing consensus among architects and system 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.) These instructions are highly flexible: any read-modify-write operation can be implemented by a short three instruction sequence consisting of an LL, manipulation of local processor register, and an SC. For instance, to fetch&increment a memory word $X$, a process performs LL to read the value of $X$ into a local register, increments that register, and then performs SC to write the register’s value to $X$. In the unlikely scenario that SC fails (because of interference from a successful SC by another process), $p$ will simply re-execute the instruction sequence.

Despite the desirability of LL/SC, no processor supports these instructions in hardware because it is impractical to maintain (in hardware) the state information needed to determine the success or failure of each process’ SC operation on each word of memory. Consequently, modern processors support only close approximations to LL/SC, namely, either compare&swap, also known as CAS (e.g., UltraSPARC [10] and Itanium [5]) or restricted versions of LL/SC, known as RLL/RSC (e.g., POWER4 [7], MIPS [20], and Alpha [19] processors). Since CAS suffers from the well-known ABA problem [3] and RLL/RSC impose severe restrictions on their use \(^1\) [17], it is difficult to design algorithms based on these instructions.

Thus, there is a gap between what the algorithm designers want (namely, LL/SC) and what the multiprocessors actually support (namely, CAS or RLL/RSC). Designing efficient algorithms to bridge this hardware-software gap has been the goal of a lot of recent research [1, 2, 6, 11, 14, 16, 17]. Most of this research is focused on implementing small LL/SC objects, i.e., LL/SC objects whose value fits in a single machine word (which is 64-bits in the case of most machines) [2, 6, 11, 14, 16, 17]. However, many existing applications [1, 4, 12, 13] need large LL/SC objects, i.e., LL/SC objects whose value does not fit in a single machine word. To address this need, Anderson and Moir [1] designed an algorithm that implements a multi-word LL/SC object from word-sized LL/SC objects and atomic registers. Their algorithm is wait-free and implements a W-word LL/SC object $O$, shared by $N$ processes, with the following time and space complexity. A process completes an LL or SC operation on $O$ in $O(W)$ hardware instructions (thus, the algorithm is clearly time optimal). The space complexity—the number of hardware words needed to implement $O$—is $O(N^2 W)$.\(^2\) In this paper, we use novel buffer management ideas to design a wait-free algorithm that cuts down the space complexity by a factor of $N$ to $O(N W)$, while still being time optimal. Our main result is summarized as follows:

- $LL(p, O)$ returns $O$’s value.
- $SC(p, O, v)$ either “succeeds” or “fails”. In the following we explain (i) what it means for SC to succeed or fail, and (ii) the rule for determining the SC’s success or failure.

If $SC(p, O, v)$ succeeds, it changes $O$’s value to $v$ and returns $true$ to $p$. If it fails, $O$’s value remains unchanged and SC returns $false$ to $p$.

The following rule determines the success or failure: An $SC(p, O, v)$ succeeds if and only if no process performed a successful SC on $O$ since process $p$’s latest LL operation on $O$.

- $VL(p, O)$ returns $true$ to $p$ if and only if no process performed a successful SC on $O$ since $p$’s latest LL operation on $O$.

Figure 1: Effect of process $p$ executing $LL$, $SC$ and $VL$ operations on an object $O$

---

\(^1\)The RLL/RSC semantics are weaker than LL/SC semantics in two respects [17]: (i) SC may experience spurious failures, i.e., SC might sometimes fail even when it should have succeeded, and (ii) a process must not access any shared variable between its $LL$ and the subsequent SC.

\(^2\)More efficient algorithms were also given by Anderson and Moir [1] and Moir [17], but these algorithms implement weaker objects, known in the literature as WLL/SC objects. Unlike LL, the WLL operation sometimes fails to return the object’s value, rendering WLL/SC objects not useful for many applications [4, 12, 13]. This paper is concerned only with multi-word LL/SC objects, and not with WLL/SC objects.
Statement of the main result: Consider the problem of implementing a $W$-word LL/SC object $O$, shared by $N$ processes, from word-sized LL/SC objects and word-sized registers supporting normal read and write operations. We design a wait-free algorithm that guarantees that each process completes an LL or SC operation on $O$ in $O(W)$ machine instructions. The algorithm’s space complexity is $O(NW)$.

We believe that this result is important for two reasons. First, it introduces novel buffer management ideas that significantly reduce the number of buffer replicas while still preventing race conditions. Second, many existing algorithms employ $W$-word LL/SC object as the underlying primitive (examples include the recent snapshot algorithms [12, 13], universal constructions [1] and the construction of closed objects [4]). By the result of this paper, the space complexity of all of these algorithms comes down by a factor of $N$.

2 Implementing the $W$-word LL/SC Object

Figure 2 presents an algorithm for implementing a $W$-word LL/SC/VL object $O$. In the rest of this section, we describe informally how the algorithm works.

2.1 The variables used

We begin by describing the variables used in the algorithm. $\text{BUF}[0..3N-1]$ is an array of $3N$ $W$-word safe buffers. Of these, $2N$ buffers hold the $2N$ most recent values of $O$ and the remaining $N$ buffers are “owned” by processes, one buffer by each process. Process $p$’s local variable, $\text{mybuf}_p$, is the tag associated with the current value of $O$ and consists of two fields: the index of the buffer that holds $O$’s current value and the sequence number associated with $O$’s current value. The sequence number increases by 1 (modulo $2N$) with each successful SC on $O$. The buffer holding $O$’s current value is not reused until $2N$ more successful SC’s are performed. Thus, at any point, the $2N$ most recent values of $O$ are available and may be accessed as follows. If the current sequence number is $k$, the sequence numbers of the $2N$ most recent successful SC’s (in the order of their recentness) are $k, k-1, \ldots, 0, 2N-1, 2N-2, \ldots, k+1$; and $\text{Bank}[j]$ is the index of the buffer that holds the value written to $O$ by the most recent successful SC with sequence number $j$. Finally, it turns out that a process $p$ might need the help of other processes in completing its LL operation on $O$. The variable $\text{Help}[p]$ facilitates coordination between $p$ and the helpers of $p$.

2.2 The helping mechanism

The crux of our algorithm lies in its helping mechanism by which SC operations help LL operations. Specifically, a process $p$ begins its LL operation by announcing its operation to other processes. It then attempts to read the buffer containing $O$’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, 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 an LL operation on $O$ and obtains a value $V$ associated with sequence number $s$ (i.e., the latest SC before $p$’s LL wrote $V$ in $O$ and had the sequence number $s$). Following its LL, suppose that $p$ invokes an SC operation. Before attempting to make this SC operation (of sequence number $(s+1) \mod 2N$) succeed, our algorithm requires $p$ to check if the process $s \mod N$ has an ongoing LL operation that requires help (thus, the decision of which process to help is based on sequence number). If so, $p$ hands over the buffer it owns containing the value $V$ to the process $s \mod N$. If several processes try to help, only one will succeed. Thus, the process numbered $s \mod N$ is helped (if necessary) every time the sequence number changes from $s$ to $(s+1) \mod 2N$. Since sequence number increases by 1 with each successful SC, it follows that every process is examined twice for possible help in a span of $2N$ successful SC operations. Recall further the earlier stated property that the buffer holding $O$’s current value is not reused until $2N$ more successful SC’s are performed. As a consequence of the above facts, if a process $p$ begins reading the buffer that holds $O$’s current value and the buffer happens to be reused while $p$ still reads it (because $2N$ successful SC’s have since taken place), some process is sure to have helped $p$ by handing it a valid value of $O$.

2.3 The role of $\text{Help}[p]$

The variable $\text{Help}[p]$ plays an important role in the helping scheme. It has two fields, a binary value (that indicates if $p$ needs help) and a buffer index. When $p$ initiates an LL operation, it seeks the help of other processes by writing $(1, b)$, where $b$ is the index of the buffer that $p$ owns (see Line 1). If a process $q$ helps $p$, it does so handing over its buffer—say, $c$—containing a valid value of $O$ to $p$ by writing $(0, c)$. (This writing is performed with a SC operation to ensure that at most one process succeeds in helping $p$.) Once $q$ writes $(0, c)$ in $\text{Help}[p]$, $p$ 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 $b$.

The above ideas are implemented in our algorithm as follows. Before $p$ returns from its LL operation, it withdraws its request for help by executing the code at Lines 8–10. First, $p$ reads $\text{Help}[p]$ (Line 8). If $p$ was already helped (i.e., flag is 0), $p$ updates $\text{mybuf}_p$ to reflect that $p$’s ownership has changed.
to the buffer in which the helper process had left a valid value (Line 10). If \( p \) was not yet helped, \( p \) attempts to withdraw its request for help by writing 0 into the first field of Help[\( p \)] (Line 9). If \( p \) does not succeed, some process must have helped \( p \) while \( p \) was between Lines 8 and 9; in this case, \( p \) assumes the ownership of the buffer handed by that helper (Line 10). If \( p \) succeeds in writing 0, then the second field of Help[\( p \)] still contains the index of \( p \)’s own buffer, and so \( p \) reclaims the ownership of its own buffer (Line 10).

### 2.4 Two obligations of LL

In any implementation, there are two conditions that an LL operation must satisfy to ensure correctness. Our code will be easy to follow if these conditions are first understood, so we explain them below.

Consider an execution of the LL procedure by a process \( p \). Suppose that \( V \) is the value of \( O \) when \( p \) invokes the LL procedure and suppose that \( k \) successful SC’s take effect during the execution of this procedure, changing \( O \)’s value from \( V \) to \( V_1 \), \( V_1 \) to \( V_2 \), ... , \( V_{k-1} \) to \( V_k \). Then, any of \( V \), \( V_1 \), ..., \( V_k \) would be a valid value for \( p \)’s LL procedure to return. However, there is a significant difference between returning \( V_k \) (the current value) versus returning an older (but valid) value from \( V \), \( V_1 \), ... , \( V_{k-1} \): assuming that other processes do not perform successful SC’s between \( p \)’s LL and \( p \)’s subsequent SC, the specification of LL/SC operations requires \( p \)’s subsequent SC to succeed in the former case and fail in the latter case. Thus, \( p \)’s LL procedure, besides returning a valid value, has the additional obligation of ensuring the success or failure of \( p \)’s subsequent SC (or VL) based on whether or not its return value is current.

In our algorithm, the SC procedure (Lines 12–22) includes exactly one SC operation on the variable \( x \) (Line 19) and the former succeeds if and only if the latter succeeds. Therefore, we can restate the two obligations on \( p \)’s LL procedure as

<table>
<thead>
<tr>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>valuetype = array[0..W-1] of 64-bit word</td>
</tr>
<tr>
<td>xtype = record buf: 0..3N-1; seq: 0..2N-1 end</td>
</tr>
<tr>
<td>helptype = record helptype: {0,1}; buf: 0..3N-1 end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Shared variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>X: xtype</td>
</tr>
<tr>
<td>Bank: array[0..2N-1] of 0..3N-1</td>
</tr>
<tr>
<td>Help: array[0..N-1] of helptype</td>
</tr>
<tr>
<td>BUF: array[0..3N-1] of valuetype</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local persistent variables at each ( p \in {0, 1, \ldots, N-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>mybuf(_p): [0..3N-1]; ( x_p ): xtype</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X = (0, 0) ); BUF[0] = the desired initial value of ( O )</td>
</tr>
<tr>
<td>Bank[( k )] = ( k ), for all ( k \in {0, 1, \ldots, 2N-1} )</td>
</tr>
<tr>
<td>mybuf(_p) = ( 2N + p ), for all ( p \in {0, 1, \ldots, N-1} )</td>
</tr>
<tr>
<td>Help[( p )] = ( (0, \omega) ), for all ( p \in {0, 1, \ldots, N-1} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>procedure LL(( p ), ( O ), retval)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Help[( p )] = ( (1, \text{mybuf}_p) )</td>
</tr>
<tr>
<td>2: ( x_p = \text{LL}(X) )</td>
</tr>
<tr>
<td>3: \text{copy BUF}[x_p.buf] into \*retval</td>
</tr>
<tr>
<td>4: if \text{LL}(Help[( p )]) \equiv (0, b)</td>
</tr>
<tr>
<td>5: ( x_p = \text{LL}(X) )</td>
</tr>
<tr>
<td>6: \text{copy BUF}[x_p.buf] into \*retval</td>
</tr>
<tr>
<td>7: if \text{VL}(X) \text{copy BUF}[2] into \*retval</td>
</tr>
<tr>
<td>8: if \text{LL}(Help[( p )]) \equiv (1, c)</td>
</tr>
<tr>
<td>9: \text{SC}(Help[( p )], (0, c))</td>
</tr>
<tr>
<td>10: mybuf(_p) = Help[( p )].buf</td>
</tr>
<tr>
<td>11: \text{copy \*retval into BUF[mybuf}_p]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>procedure SC(( p ), ( O ), ( v )) returns boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>12: if (LL(Bank[( x_p, \text{seq} )]) \neq x_p.buf) \wedge \text{VL}(X)</td>
</tr>
<tr>
<td>13: \text{SC}(\text{Bank}[( x_p, \text{seq} )], x_p.buf)</td>
</tr>
<tr>
<td>14: if (\text{LL}(Help[( x_p, \text{seq} \mod N )]) \equiv (1, d) \wedge \text{VL}(X))</td>
</tr>
<tr>
<td>15: if \text{SC}(\text{Help}[( x_p, \text{seq} \mod N )], (0, mybuf(_p)))</td>
</tr>
<tr>
<td>16: mybuf(_p) = ( d )</td>
</tr>
<tr>
<td>17: \text{copy \*v into BUF[mybuf}_p]</td>
</tr>
<tr>
<td>18: ( e = \text{Bank}[x_p, \text{seq} + 1 \mod 2N] )</td>
</tr>
<tr>
<td>19: if (SC(( X ), (mybuf(_p), (x_p, \text{seq} + 1 \mod 2N))))</td>
</tr>
<tr>
<td>20: mybuf(_p) = ( e )</td>
</tr>
<tr>
<td>21: return true</td>
</tr>
<tr>
<td>22: return false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>procedure VL(( p ), ( O )) returns boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>23: return \text{VL}(X)</td>
</tr>
</tbody>
</table>

Figure 2: Implementation of the \( N \)-process \( W \)-word LL/SC/VL variable \( O \) from single-word LL/SC/VL procedures.
follows: (O1) It must return a valid value \( u \), and (O2) If other processes do not perform successful SC’s after \( p \)’s LL, \( p \)’s subsequent SC (or VL) on \( x \) must succeed if and only if the return value \( u \) is current.

2.5 Code for LL

A process \( p \) performs an LL operation on \( O \) by executing the procedure LL(\( p, O, retval \)), 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 (Line 1). It then attempts to obtain the current value of \( O \) (Lines 2–4), by performing the following steps. First, \( p \) reads \( x \) (Line 2) to determine the buffer holding \( O \)’s current value, and then reads that buffer (Line 3). While \( p \) reads the buffer on Line 3, the value of \( O \) might change because of successful SC’s by other processes. Specifically, there are three possibilities for what happens while \( p \) executes Line 3: (i) no successful SC is performed by any process, (ii) fewer than \( 2N - 1 \) successful SC’s are performed, or (iii) at least \( 2N \) successful SC’s are performed. In the first case, it is obvious that \( p \) reads a valid value on Line 3. Interestingly, in the second case too, the value read on Line 3 is a valid value. This is because, as remarked earlier, our algorithm does not reuse a buffer until \( 2N \) more successful SC’s have taken place. In the third case, \( p \) cannot rely on the value read on Line 3. However, by the helping mechanism described earlier, a helper process would have made available a valid 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. Further, as we now explain, \( p \) can also determine which of the three cases actually holds. To do this, \( p \) reads Help[\( p \)] to check if it has been helped (Line 4). If it has not been helped yet, Case (i) or (ii) must hold, which implies that \( retval \) has a valid value of \( O \). Hence, returning this value meets the obligation O1. It meets obligation O2 as well because the value in \( retval \) is the current value of \( O \) at the moment when \( p \) read \( x \) (Line 2); hence, \( p \)’s subsequent SC (or VL) on \( x \) will succeed if and only if \( x \) does not change, i.e., if and only if the value in \( retval \) is still current. So, \( p \) returns from the LL operation after withdrawing its request for help (Lines 8–10) and storing the return value into \( p \)’s own buffer (Line 11) (\( p \) will use this buffer in the subsequent SC operation to help another process complete its LL operation, if necessary).

If upon reading Help[\( p \)] (Line 4), \( p \) finds out that it has been helped, \( p \) knows that Case (iii) holds and a helper process must have already written in Help[\( p \)] the index of a buffer containing a valid value \( u \) of \( O \). However, \( p \) is unsure whether this valid value \( u \) is current or old. If \( u \) is current, it is incorrect to return \( u \): the return of \( u \) will fail to meet the obligation O2. This is because \( p \)’s subsequent SC on \( x \) will fail, contrary to O2 (it will fail because \( x \) has changed since \( p \) read it at Line 2). For this reason, although \( p \) has access to a valid value handed to it by the helper, it does not return it. Instead, \( p \) attempts once more to obtain the current value of \( O \) (Lines 5–7). To do this, \( p \) again reads \( x \) (Line 5) to determine the buffer holding \( O \)’s current value, and then reads that buffer (Line 6). Next, \( p \) validates \( x \) (Line 7). If this validation succeeds, it is clear that \( retval \) has a valid value and, by returning this value, the LL operation meets both its obligations (O1 and O2). If the validation fails, \( O \)’s value must have changed while \( p \) was between Lines 5 and 7. This implies that the value handed by the helper (which had been around even before \( p \) executed Line 5) is surely not current. Furthermore, the failure of VL (on Line 7) implies that \( p \)’s subsequent SC on \( x \) will fail. Thus, returning the value handed by the helper satisfies both obligations, O1 and O2. So, \( p \) copies the value handed by the helper into \( retval \) (Line 7), withdraws its request for help (Lines 8–10), and stores the return value into \( p \)’s own buffer (Line 11), to be used in \( p \)’s subsequent SC operation.

2.6 Code for SC

A process \( p \) performs an SC operation on \( O \) by executing the procedure SC(\( p, O, v \)), where \( v \) is the pointer to a block of W-words which contain the value to write to \( O \) if SC succeeds. On the assumption that \( x \) hasn’t changed since \( p \) read it in its latest LL, i.e., \( x \) still contains the buffer index \( bindex \) and the sequence number \( s \) associated with the latest successful SC, \( p \) reads the buffer index \( b \) in Bank[s] (Line 12). The reason for this step is the possibility that Bank[s] has not yet been updated to hold \( bindex \), in which case \( p \) should update it. So, \( p \) checks whether there is a need to update Bank[s], by comparing \( b \) with \( bindex \) (Line 12). If there is a need to update, \( p \) first validates \( x \) (Line 12) to confirm its earlier assumption that \( x \) still contains the buffer index \( bindex \) and the sequence number \( s \). If this validation fails, it means that the values that \( p \) read from \( x \) have become stale, and hence \( p \) abandons the updating. (Notice that, in this case, \( p \)’s SC operation also fails.) If the validation succeeds, \( p \) attempts to update Bank[s] (Line 13). This attempt will fail if and only if some process did the updating while \( p \) executed Lines 12–13. Hence, by the end of this step, Bank[s] is sure to hold the value \( bindex \).

Next, \( p \) tries to determine whether some process needs help with its LL operation. Since \( p \)’s SC is attempting to change the sequence number from \( s \) to \( s + 1 \), the process to help is \( q = s \mod N \). So, \( p \) reads Help[\( q \)] to check whether \( q \) needs help (Line 14). If it does, \( p \) first validates \( x \) (Line 15) to make sure that \( x \) still contains the buffer index \( bindex \) and the sequence number \( s \). If this validation fails, it means that the values that \( p \) read from \( x \) have become stale, and hence \( p \) abandons the helping. (Notice that, in this case, \( p \)’s SC operation also fails.) If the validation succeeds, \( p \) attempts to help \( q \) by handing it \( p \)’s buffer which, by Line 11, contains a valid value of \( O \) (Line 15). If \( p \) succeeds in helping \( q \), \( p \) gives up its buffer to \( q \) and assumes ownership of \( q \)’s buffer (Line 16). (Notice that \( p \)’s SC on Line 15 fails if and only if, while \( p \) executed Lines 14–15, either another process already helped \( q \) or \( q \) withdrew its request for help.)

Next, \( p \) copies the value \( v \) to its buffer (Line 17). Then, \( p \)
reads the index $e$ of the buffer that holds $O$’s old value associated with the next sequence number, namely, $(s + 1) \mod 2N$ (Line 18). Finally, $p$ attempts its SC operation (Line 19) by trying to write in $x$ the index of its buffer and the next sequence number $s'$. This SC will succeed if and only if no successful SC was performed since $p$’s latest LL. Accordingly, the procedure returns true if and only if the SC on Line 19 succeeds (Lines 21–22). In the event that SC is successful, $p$ gives up ownership of its buffer, which now holds $O$’s current value, and becomes the owner of $BUF[e]$, the buffer holding $O$’s old value with sequence number $s'$, which can now be safely reused (Line 20).

The procedure VL is self-explanatory (Line 23). Based on the above discussion, we have:

**Theorem 1** The N-process wait-free implementation in Figure 2 of a W-word LL/SC/VL variable is linearizable. The time complexity of LL, SC and VL operations on $O$ are $O(W)$, $O(W)$ and $O(1)$, respectively. The implementation requires $O(NW)$ 64-bit safe registers and $O(N)$ 64-bit LL/SC/VL/read objects.

3 Proof of the algorithm

Let $E$ be any finite execution history of the algorithm in Figure 2. Let $OP$ be some LL operation, $OP'$ some SC operation, and $OP''$ some VL operation in $E$. Then, we define the linearization points (LPs) for $OP$, $OP'$ and $OP''$ as follows. If the condition at Line 4 of $OP$ fails (i.e., $LL(\text{Help}[p]) \neq (0, b)$), $LP(\text{OP})$ is Line 2 of OP. If the condition at Line 7 fails (i.e., $VL(x)$ returns true), $LP(\text{OP})$ is Line 5 of OP. If the condition at Line 7 succeeds, let $p$ be the process executing OP. Then, we show that (1) there exists exactly one SC operation $SC_q$ on $O$ that writes into $\text{Help}[p]$ during OP, and (2) the VL operation on $x$ at Line 14 of $SC_q$ is executed at some time $t$ during OP; we then set $LP(\text{OP})$ to time $t$. We set $LP(\text{OP'})$ to Line 19 of $OP'$, and $LP(\text{OP''})$ to Line 23 of OP''.

**Lemma 1** Let $E$ be any finite execution history of the algorithm in Figure 2. Let $SC_i$ be the $i$th successful SC operation in $E$, and $p_i$ the process executing $SC_i$. Then, at Line 19 of $SC_i$, $p_i$ writes the value of the form $(\_ , i \mod 2N)$ into $x$.

**Proof.** (By induction) For the base case (i.e., $i = 0$), the lemma holds trivially, since $SC_0$ is the “initializing” SC. The inductive hypothesis states that the lemma holds for $i = k$. We now show that the lemma holds for $i = k + 1$ as well. Let $SC_k^{X}$ and $SC_{k+1}^{X}$, be, respectively, the (successful) SC on $x$ at Line 19 of $SC_k$, and the (successful) SC on $x$ at Line 19 of $SC_{k+1}$. Let $LL_{op}$ be $p_{k+1}$’s latest LL operations to precede $SC_{k+1}$, and $LL^{X}$ the latest LL on $x$ during $LL_{op}$. Since $SC_{k+1}$ succeeds, it means that $LL^{X}$ takes place after $SC_k^{X}$. Furthermore, since $SC_{k+1}$ is the first successful SC after $SC_k$, it means that $x$ doesn’t change between $SC_k^{X}$ and $LL^{X}$. Consequently, the value of $x$ returned by $LL^{X}$ is of the form $(\_ , k \mod 2N)$. Hence, $SC_{k+1}^{X}$ writes into $x$ the value of the form $(\_ , (k + 1) \mod 2N)$.

**Lemma 2** Let $E$ be any finite execution history of the algorithm in Figure 2. Let $p$ be some process, and $LL_p$ some LL operations by $p$ in $E$. Let $t$ be the time when $p$ executes Line 1 of $LL_p$, and $t'$ the time just prior to Line 10 of $LL_p$. Let $t''$ be either (1) the moment when $p$ executes Line 1 of its first LL operation after $LL_p$, if such operation exists, or (2) the end of $E$, otherwise. Then, the following statements hold:

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

(S2) Any value written into $\text{Help}[p]$ 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 $\text{Help}[p]$.

**Proof.** Statement (S2) follows trivially from the fact that the only two operations that can affect the value of $\text{Help}[p]$ during $(t, t'')$ are (1) the SC at Line 9 of $LL_p$, and (2) the SC at Line 15 of some other process’ SC operation, both of which attempt to write $(0, \_)$. We now prove the statement (S1). Suppose that (S1) does not hold. Then, during $(t, t')$, either (1) two or more writes on $\text{Help}[p]$ are performed, or (2) no writes on $\text{Help}[p]$ are performed. In the first case, we know (by an earlier argument) that each write on $\text{Help}[p]$ during $(t, t')$ must have been performed either by the SC at Line 9 of $LL_p$, or by the SC at Line 15 of some other process’ SC operation. Let $SC_1$ and $SC_2$ be the first two SC operations on $\text{Help}[p]$ to write into $\text{Help}[p]$ during $(t, t')$. Let $q_1$ (respectively, $q_2$) be the process executing $SC_1$ (respectively, $SC_2$). Let $LL_1$ (respectively, $LL_2$) be the latest LL operations on $\text{Help}[p]$ by $q_1$ (respectively, $q_2$) to precede $SC_1$ (respectively, $SC_2$). Then, both $LL_1$ and $LL_2$ return a value of the form $(1, \_)$. Furthermore, $LL_2$ takes place after $SC_1$, or else $SC_2$ would fail. Since $\text{Help}[p]$ doesn’t change between $SC_1$ and $SC_2$, it means that $LL_2$ returns the value of the form $(0, \_)$, which is a contradiction.

In the second case (where no writes on $\text{Help}[p]$ take place during $(t, t')$), we examine two possibilities: either the LL operation at Line 8 of $LL_p$ returns a value of the form $(1, \_)$ or it doesn’t. In the first case, since there are no writes into $\text{Help}[p]$ during $(t, t')$, the SC at Line 9 of $LL_p$ must succeed, which is a contradiction to the fact that no writes into $\text{Help}[p]$ take place during $(t, t')$. In the second case, $\text{Help}[p]$ must have changed between the time $p$ executed Line 1 and the time $p$ executed Line 8, which is a contradiction to the fact that no writes into $\text{Help}[p]$ take place during $(t, t')$. Hence, the statement (S1) holds.

We now prove the statement (S3). Suppose that (S3) does not hold. Then, at least one write on $\text{Help}[p]$ takes place during $(t'', t''')$. By an earlier argument, any write on $\text{Help}[p]$
during $(t'', t''')$ must have been performed either by the SC at Line 9 of LLp, or by the SC at Line 15 of some other process’ SC operation. Let SC3 be the first SC operations on Help[p] to write into Help[p] during $(t'', t''')$. Let q3 be the process executing SC3. Let LLt be the latest LL operations on Help[p] by q3 to precede SC3. Then, LLt returns a value of the form $(1, _)$. Furthermore, LLt must take place after time $t'''$, or else SC3 would fail. Since Help[p] doesn’t change between time $t''$ and SC3, it means that LLt returns the value of the form $(0, _)$, which is a contradiction. Hence, we have the statement (S3).

\[\square\]

**Invariants:** Let $E$ be any finite execution history of the algorithm in Figure 2, and $t$ some time during $E$. Let $PC^i_p$ be the value of process $p$’s program counter at time $t$. For any shared variable $A$, let $A^t$ be the value of that variable at time $t$. For any local variable $a$, let $a^t$ be the value of that variable at time $t$. For any register $r$ at process $p$, let $r^t$ be the value of that register at time $t$. Then, the following invariants holds at time $t$.

(1) Let $m_p(t)$, for all $p \in \{0, 1, \ldots, N - 1\}$, be defined as follows:

- if $PC^i_p(t) \in (2 \ldots 10) \land Help[p] \equiv (0, b)$, then $m_p(t) = b$,
- if $PC^i_p(t) = 16$, then $m_p(t) = d^t(p)$,
- if $PC^i_p(t) = 20$, then $m_p(t) = e^t(p)$,
- otherwise, $m_p(t) = mybuf^t_p$.

Let $(a, k)$ be the value of $X$ at time $t$ (i.e., $X^t = (a, k)$). Let $b_i(t)$, for all $i \in \{0, 1, \ldots, 2N - 1\}$, be defined as follows: $b_i(t) = Bank^k[i]$, for all $i \neq k$, and $b_k(t) = a$. Then, at time $t$, we have $m_0(t) \neq m_1(t) \neq \ldots \neq m_{2N-1}(t) \neq b_0(t) \neq b_1(t) \neq \ldots \neq b_{2N-1}(t)$.

(12) Let $(b_k, k)$ be the value of $X$ at time $t$ (i.e., $X^t = (b_k, k)$). Let $t_k < t$ be the time during $E$ when $(b_k, k)$ was written into $X$. If $t_k \neq 0$, let $t_{k-1} < t_k$ be the time during $E$ when $(b_{k-1}, (k - 1) \mod 2N)$ was written into $X$, for some value $b_{k-1}$. If $t_k \neq 0$, then during $(t_{k-1}, t_k)$, exactly one write into $Bank^k[(k - 1) \mod 2N]$ is performed, and the value written by that write is $b_{k-1}$.

Furthermore, no other location in $Bank$ is written into during $(t_{k-1}, t_k)$.

**Proof.** (By induction) For the base case for (11), (i.e., $t = 0$), the invariants hold trivially. The base case for (12) is more complicated, and is established and proved by the following claim.

**Claim 1** Let $t_2$ be the time just before $X$ is written to for the second time after time 0. Then, during $(0, t_2)$, invariant (12) holds.

**Proof.** Let $t_1$ be the first time after time 0 that $X$ is written to. Then, during $(0, t_1)$, the invariant (12) holds trivially. To show that the invariant holds during $(t_1, t_2)$, we assume that the initialization phase initializes Bank[0] (to 0) at time 0 and all other locations just before time 0. Then, it is clear from the algorithm that any process to execute Line 12 during $(0, t_1)$ must (1) perform the LL on Bank[0], and (2) discover that Bank[0] already has value 0. Therefore, it follows that (1) no write into Bank[0] (except the initialization write) takes place during $(0, t_1)$, and (2) no other location in Bank is written into during $(0, t_1)$, which proves the claim.

\[\square\]

The inductive hypothesis states that the invariant (11) holds at time $t \geq 0$, and the invariant (12) at time $t \geq t_2$. Let $t' \geq t$ be some time after $t$ that some process, say $p$, makes a step. Then, we show that the invariants hold at time $t'$ as well. We first prove the invariant (12).

Notice that, if $PC^i_p(t) \neq 19$, the invariant trivially holds. If $PC^i_p(t) = 19$, then at time $t'$, $p$ writes $(b_{k-1}, (k + 1) \mod 2N)$ into $X$, for some value $b_{k-1}$ (by Lemma 1). In the next five claims, we will show that during $(t_k, t')$, (1) exactly one write into Bank[k mod 2N] is performed, (2) the value written by that write is $b_k$, and (3) no other location in Bank is written into.

**Claim 2** If some process $q$ writes into the Bank array during $(t_k, t')$, then $q$ performed its latest LL on $X$ during $(t_k, t')$.

**Proof.** Suppose not. Then, there exists some $i \in \{0, 1, \ldots, 2N - 1\}$ and some process $q$, such that $q$ writes into Bank[i] during $(t_k, t')$, yet it performed its latest LL on $X$ prior to $t_k$. Since $q$ writes into the $i$th location in Bank, it means that (1) there exists a time $t_{i+2mN} < t_k$ when the value $(b_{i+2mN,i})$ is written into $X$, for some $b_{i+2mN,i}$, (2) there exists a time $t_{i+2mN+1} < t_k$ when the value $(b_{i+2mN+1}, i + 1 \mod 2N)$ is written into $X$, for some $b_{i+2mN+1,i}$, (3) $t_{i+2mN+1}$ is the first time after $t_{i+2mN}$ that $X$ changes, (4) $q$ performed its latest LL on $X$ during $(t_{i+2mN,N}, t_{i+2mN+1})$, (5) $q$’s latest LL on $X$ returned the value $(b_{i+2mN,i})$, and (6) $q$ performed its latest VL on $X$ (Line 12) during $(t_{i+2mN,N}, t_{i+2mN+1})$. Consequently, $q$ performed its LL on Bank[i] during $(t_{i+2mN,N}, t_{i+2mN+1})$ as well. By inductive hypothesis, there exists a time $t'_{i+2mN, N} \in (t_{i+2mN,N}, t_{i+2mN+1})$ when the value $b_{i+2mN}$ is written into Bank[i]. Then, $q$ must have performed its LL on Bank[i] after time $t'_{i+2mN, N}$ (or else $q$’s SC at Line 15 would fail). In that case, however, $q$’s LL on Bank[i] returns $b_{i+2mN,i}$. Therefore, $q$ does not perform the SC on Bank[i] at all (due to the failure of the first condition at Line 12), which is a contradiction.

\[\square\]

**Claim 3** During $(t_k, t')$, the only value that can be written into Bank[k mod 2N] is $b_k$.

**Proof.** Suppose not, i.e., suppose that there exists some process $q$ that writes into Bank[k mod 2N] a value different than $b_k$. Then, $q$ must have performed its latest LL on $X$ before time $t_k$, which is a contradiction to Claim 2.

\[\square\]
Claim 4 During \((t_k, t')\), at most one write into \(\text{Bank}[k \mod 2N]\) is performed.

**Proof.** Suppose not. Then, two or more writes into \(\text{Bank}[k \mod 2N]\) take place during \((t_k, t')\). Let \(SC_1\) and \(SC_2\) be the first two SC operations on \(\text{Bank}[k \mod 2N]\) to write into \(\text{Bank}[k \mod 2N]\) during \((t_k, t')\). Let \(q_1\) (respectively, \(q_2\)) be the process executing \(SC_1\) (respectively, \(SC_2\)). Let \(SC_{q_1}\) (respectively, \(SC_{q_2}\)) be the SC operation on \(\mathcal{O}\) during which \(SC_1\) (respectively, \(SC_2\)) is executed. Let \(LL_1\) (respectively, \(LL_2\)) be the LL operations on \(\text{Bank}[k \mod 2N]\) at Line 12 of \(SC_{q_1}\) (respectively, \(SC_{q_2}\)). Then, by Claim 3, both \(SC_1\) and \(SC_2\) write \(b_k\) into \(\text{Bank}[k \mod 2N]\). Furthermore, \(LL_2\) takes place after \(SC_1\) (or else \(SC_2\) would fail). Since \(\text{Bank}[k \mod 2N]\) doesn’t change between \(SC_1\) and \(SC_2\), it means that \(LL_2\) reads \(b_k\) from \(\text{Bank}[k \mod 2N]\). By Claim 2, the latest LL operation on \(x\) by \(q_2\) prior to \(SC_{q_2}\) returns the value \(b_k\). Therefore, the first condition at Line 12 of \(SC_{q_2}\) must fail. Hence, \(SC_2\) is never executed, which is a contradiction. \(\square\)

Claim 5 During \((t_k, t')\), at least one write into \(\text{Bank}[k \mod 2N]\) is performed.

**Proof.** Suppose not. Then, no write into \(\text{Bank}[k \mod 2N]\) is performed during \((t_k, t')\). Let \(p_k\) be the process that wrote \((b_k, k)\) into \(x\) at time \(t_k\). By inductive hypothesis for (I1), we know that at the time just before \(t_k\), the value of \(\text{Bank}[k \mod 2N]\) is different than the value of \(\text{mybuf}_{p_k}\). Furthermore, just before \(t_k\), \(\text{mybuf}_{p_k} = b_k\). Therefore, at time \(t_k\), the value of \(\text{Bank}[k \mod 2N]\) is different than \(b_k\).

Let \(SC_p\) be the SC operation on \(\mathcal{O}\) during which \(p\) performs an SC on \(x\) at time \(t'\). Since \(p\)’s SC on \(x\) succeeds, it means that (1) \(p\)’s latest LL on \(x\) happens during \((t_k, t')\) and returns \((b_k, k \mod 2N)\), (2) \(p\)’s LL on \(\text{Bank}[k \mod 2N]\) at Line 12 of \(SC_p\) happens during \((t_k, t')\), and (3) \(p\)’s VL on \(x\) at Line 12 of \(SC_p\) happens during \((t_k, t')\) and returns true. Since no write into \(\text{Bank}[k \mod 2N]\) is performed during \((t_k, t')\), and, by the previous argument, the value of \(\text{Bank}[k \mod 2N]\) at time \(t_k\) is different than \(b_k\), it means that \(p\)’s LL on \(\text{Bank}[k \mod 2N]\) returns a value different than \(b_k\). Therefore, \(p\) executes the SC at Line 13 of \(SC_p\). Notice that this SC operation also happens during \((t_k, t')\). Since no write into \(\text{Bank}[k \mod 2N]\) happens during \((t_k, t')\), it means that \(p\)’s SC on \(\text{Bank}[k \mod 2N]\) at Line 13 of \(SC_p\) succeeds and writes \(b_k\) into \(\text{Bank}[k \mod 2N]\). That is a contradiction to the fact that no write into \(\text{Bank}[k \mod 2N]\) happens during \((t_k, t')\). \(\square\)

Claim 6 During \((t_k, t')\), no write into \(\text{Bank}[i]\) is performed, for all \(i \in \{0, 1, \ldots, 2N - 1\} - \{k \mod 2N\}.

**Proof.** Suppose not. Then, some process \(q\) writes into \(\text{Bank}[i]\) during \((t_k, t')\), for some \(i \in \{0, 1, \ldots, 2N - 1\} - \{k \mod 2N\}\). By Claim 2, \(q\) must have performed its latest LL operation on \(x\) during \((t_k, t')\) as well. This LL on \(x\) must therefore return the value \((b_k, k)\), which means that \(q\) writes into \(\text{Bank}[k]\), which is a contradiction. \(\square\)

We now prove the invariant (I1). Let \(M(t)\) be the collection of values of \(m_0(t), m_1(t), \ldots, m_{N-1}(t)\). Let \(B(t)\) be the collection of values of \(b_0(t), b_1(t), \ldots, b_{2N-1}(t)\). Notice that if \(PC'(p) \equiv \{1 - 8, 11, 12, 14, 17, 18, 21, 22\}\), then \(p\)’s step does not impact any of the values in \(M(t)\) or \(B(t)\), and hence the invariant holds at time \(t'\) as well.

If \(PC'(p) = 9\), we examine two possibilities: either \(\text{Help}'[p] \equiv (0, \ldots, 2)\) or not. In the first case, \(p\)’s step doesn’t impact any of the values in \(M(t)\) or \(B(t)\), and hence the invariant holds at time \(t'\). In the second case, \(p\)’s SC at Line 9 succeeds, and writes \((0, \text{mybuf}_{p})\) into \(\text{Help}[p]\). Hence, we have \(m_p(t') = m_p(t)\), which means that \(M(t)\) and \(B(t)\) remain the same and the invariant holds at time \(t'\).

If \(PC'(p) = 10\), then, by Lemma 2, \(\text{Help}'[p] \equiv (0, f)\), for some value \(f\). Then, we have \((1, m_p(t) = f, 2, \text{mybuf}_{p}' = f)\), and \((3, m_p(t') = \text{mybuf}_{p}'\)). Therefore, we have \(m_p(t') = m_p(t)\), which means that the invariant holds at time \(t'\).

If \(PC'(p) = 13\), we examine two possibilities: either \(p\)’s SC fails, or it doesn’t. In the first case, \(p\)’s step doesn’t change any of the values in \(M(t)\) or \(B(t)\), and hence the invariant holds at time \(t'\). In the second case, \(p\)’s write into \(\text{Bank}[k]\) at time \(t'\) does not impact \(b_k(t)\) (i.e., we have \(b_k(t') = b_k(t) = a\)), which means that the invariant holds at time \(t'\).

If \(PC'(p) = 15\), we examine two possibilities: either \(p\)’s SC fails or it doesn’t. In the first case, \(p\)’s step doesn’t change any of the values in \(M(t)\) or \(B(t)\), and hence the invariant holds at time \(t'\). In the second case, let \(SC_p\) be the SC operation that \(p\) is currently executing. Let \(q\) be the process whose \(\text{Help}\) variable process \(p\) writes to at Line 15 of \(SC_p\). Then, by Lemma 2, we know that (1) \(PC'(q) = PC'(q) \equiv \{2, 3, \ldots, 9\}\), (2) \(\text{Help}'[q] = (1, \text{mybuf}_{q})\), and (3) \(\text{Help}'[q] = (0, \text{mybuf}_{p})\). Since \(\text{Help}[q]\) doesn’t change during the LL operation (\(\text{Op}_{q}\)) at Line 14 of \(SC_p\) and the SC operation (\(\text{Op}_{q}\)) at Line 15 of \(SC_p\), it means that \(d'(p) = \text{mybuf}_{q}\). Since \(m_p(t') = d'(p) = \text{mybuf}_{q}\) and \(m_q(t') = \text{mybuf}_{p}\), it follows that \(m_p(t') = m_q(t)\) and \(m_q(t') = m_p(t)\), which means that the invariant holds at time \(t'\).

If \(PC'(p) = 16\), then by inductive hypothesis we have \(m_p(t) = d'(p)\). Furthermore, at time \(t'\), we have \(\text{mybuf}_{p}' = d'(p)\) and \(m_p(t') = \text{mybuf}_{p}'\). Therefore, we have \(m_p(t') = m_p(t)\), which means that the invariant holds at time \(t'\).

If \(PC'(p) = 19\), we examine two possibilities: either \(p\)’s SC fails or it doesn’t. In the first case, \(p\)’s step doesn’t change any of the values in \(M(t)\) or \(B(t)\), and hence the invariant holds at time \(t'\). In the second case, let \(SC_p\) be the SC operation that \(p\) is currently executing. Then, by invariant (I2), we have (1) \(e'(p) = \text{Bank}[i'(k + 1) \mod 2N]\), and (2) \(\text{Bank}[i \mod 2N] = a\). Furthermore, by inductive hypothesis we have \((1, m_p(t) = \text{mybuf}_{p}'\), \((2, b_k(t) = a\), and \((3, \text{mybuf}_{p}' = a\).
Lemma 3 Let $E$ be any finite execution history of the algorithm in Figure 2. Let $p$ be some process, and $SC_p$ some successful SC operation by $p$ in $E$. Let $v$ be the value that $SC_p$ writes in $O$. Let $(b, i)$ be the value that $p$ writes into $X$ at Line 19 of $SC_p$. Then, $BUF[b]$ holds the value $v$ until $X$ changes at least 2N times.

Proof. Notice that, by the algorithm, the only places where $BUF[b]$ can be modified is either at Line 11 of some LL operation, or at Line 17 of some SC operation. Let $t$ be the time when $p$ writes a into $BUF[b]$ at Line 17 of $SC_p$. Let $t'$ be the time when $p$ writes (a, i) into $X$ at Line 19 of $SC_p$. Let $t''$ be the first time after $a$ that $X$ changes. Let $t'''$ be the 2Nth time after $a$ that $X$ changes. Then, by invariant (I1), no process $q$ can be at Line 11 or 17 with $mybuf_q = b$ during $(t, t')$. Similarly, no process $q$ can be at Line 11 or 17 with $mybuf_q = b$ during $(t'', t''')$. By invariant (I2), we have $Bank[i] = b$ at time $t''$. Hence, by invariant (I1), no process $q$ can be at Line 11 or 17 with $mybuf_q = b$ during $(t'', t''')$. Consequently, no process writes into $BUF[b]$ during $(t, t''')$, which proves the lemma.

Lemma 4 Let $E$ be any finite execution history of the algorithm in Figure 2. Let $p$ be some process, and $LL_p$ some LL operation by $p$ in $E$. Let $t$ be the time when $p$ executes Line 2 of $LL_p$, and $t'$ the time when $p$ executes Line 4 of $LL_p$. If the condition at Line 4 of $LL_p$ fails (i.e., $LL(HEL[p]) \neq (0, b)$), then $X$ changes at most $2N - 1$ times during $(t, t')$.

Proof. Suppose not. Then, the condition at Line 4 of $LL_p$ fails, and $X$ changes 2N or more times during $(t, t')$. Let $t''' \in (t, t')$ be the 2Nth time after $a$ that $X$ changes. Let $(b, i)$ be the value that $p$ reads from $X$ at time $t'''$. Since the condition at Line 4 of $LL_p$ fails, it means that $HEL[p]$ holds the value $(1, a)$ at all times during $(t, t')$, for some $a$. Furthermore, by Lemma 1, there exist two successful SC operations $SC_1$ and $SC_2$ on $X$ such that (1) $SC_1$ writes the value of the form $(\_a, s)$ into $X$ at some time $t_1 \in (t, t''')$, for some $s$ mod $2N = p$, (2) $SC_2$ writes the value of the form $(\_a, (s + 1) \mod 2N)$ into $X$ at some time $t_2 \in (t_1, t''')$, and (3) $SC_2$ is the first SC operation to write into $X$ after $t_1$. Let $p_2$ be the process executing $SC_2$, $LL_2$ the latest LL operation on $X$ by $p_2$ prior to $SC_2$, and $SC_{p_2}$ the SC operation on $O$ by $p_2$ during which $SC_2$ is executed. Then, $LL_2$ is executed during $(t_1, t_2)$, and returns the value of the form $(\_a, s)$. Hence, at Line 14 of $SC_{p_2}$, $p_2$ performs an LL operation on $HEL[p]$. Since $HEL[p]$ holds the value $(1, a)$ at all times during $(t, t')$, $p_2$'s LL on $HEL[p]$ must return the value $(1, a)$. Furthermore, since $SC_2$ succeeds, the VL operation at Line 14 of $SC_{p_2}$ succeeds as well. Therefore, $p_2$ executes the SC operation at Line 15 of $SC_{p_2}$. Since $HEL[p]$ doesn't change during $(t', t''')$, it also doesn't change between the time $p_2$ performs the LL of $HEL[p]$ at Line 14 of $SC_{p_2}$, and the time $p_2$ performs the SC of $HEL[p]$ at Line 15 of $SC_{p_2}$. Consequently, $p_2$'s SC at Line 15 succeeds, writing the value of the form $(0, \_a)$ into $HEL[p]$, which is a contradiction to the fact that $HEL[p]$ doesn't change during $(t, t')$.

Lemma 6 Let $E$ be any finite execution history of the algorithm in Figure 2. Let $p$ be some process, and $LL_p$ some LL operation by $p$ in $E$. Let $t$ be the time when $p$ executes Line 2 of $LL_p$, and $t'$ the time when $p$ executes Line 4 of $LL_p$. If the condition at Line 4 of $LL_p$ fails (i.e., $LL(HEL[p]) \neq (0, b)$), then the value that $p$ writes into retval at Line 3 of $LL_p$ is the value of $O$ at time $t$.

Proof. Let $(b, i)$ be the value that $p$ reads from $X$ at time $t$. Let $SC_q$ be the SC operation on $O$ that wrote that value into $X$, and $q$ the process that executed $SC_q$. Let $t'' < t$ be the time during $SC_q$ when $q$ wrote $(b, i)$ into $X$, and $v$ the value that $SC_q$ writes in $O$. Then, by Lemma 3, $BUF[b]$ will hold the value $v$ until $X$ changes at least 2N times after $t''$. Since $X$ doesn't change during $(t'', t)$, it means that $BUF[b]$ will hold the value $v$ until $X$ changes at least 2N times after $t$. Furthermore, by Lemma 4, $X$ can change at most $2N - 1$ times during $(t, t')$. Therefore, $BUF[b]$ holds the value $v$ at all times during $(t, t')$, and hence the value that $p$ writes into retval at Line 3 of $LL_p$ is the value of $O$ at time $t$. 

Proof. Let $(b, i)$ be the value that $p$ reads from $X$ at time $t$. Let $SC_q$ be the SC operation on $O$ that wrote that value into $X$, and $q$ the process that executed $SC_q$. Let $t'' < t$ be the time during $SC_q$ when $q$ wrote $(b, i)$ into $X$, and $v$ the value that $SC_q$ writes in $O$. Then, by Lemma 3, $BUF[b]$ will hold the value $v$ until $X$ changes at least 2N times after $t''$. Since $X$ doesn't change during $(t'', t)$, it means that $BUF[b]$ will hold the value $v$ until $X$ changes at least 2N times after $t$. Since $X$ doesn't change during $(t'', t)$, it means that $BUF[b]$ will hold the value $v$ at all times during $(t, t')$. Therefore, $BUF[b]$ holds the value $v$ at all times during $(t, t')$, and hence
the value that \( p \) writes into \( \text{retval} \) at Line 6 of \( LL_p \) is the value of \( O \) at time \( t \).

\( \square \)

\textbf{Lemma 7} Let \( E \) be any finite execution history of the algorithm in Figure 2. Let \( p \) be some process, and \( LL_p \) some \( LL \) operation by \( p \) in \( E \). Let \( t \) be the time when \( p \) executes Line 1 of \( LL_p \), and \( t' \) the time when \( p \) executes Line 4 of \( LL_p \). If the condition at Line 4 of \( LL_p \) succeeds (i.e., \( LL(\text{Help}[p]) \equiv (0, b) \)), then (1) there exists exactly one \( SC \) operation \( SC_q \) on \( O \) that writes into \( \text{Help}[p] \) during \( (t, t') \), and (2) the \( VL \) operation on \( X \) at Line 14 of \( SC_q \) is executed during \( (t, t') \).

\textbf{Proof.} Since the condition at Line 4 of \( LL_p \) succeeds, it means that some \( SC \) operation \( SC_q \) writes the value of the form \( (0, \_\_\) into \( \text{Help}[p] \) during \( (t, t') \). By Lemma 2, \( SC_q \) is the only \( SC \) operation that writes into \( \text{Help}[p] \) during \( (t, t') \). Let \( t'' \in (t, t') \) be the time when \( SC_q \) writes into \( \text{Help}[p] \).

Let \( q \) be the process executing \( SC_q \). Since \( q \) writes into \( \text{Help}[p] \) at time \( t'' \), it means that \( \text{Help}[p] \) does not change between \( q \)'s \( LL \) at Line 14 of \( SC_q \) and \( t'' \). Therefore, \( q \)'s \( LL \) at Line 14 of \( SC_q \) occurs during the time interval \( (t, t'') \). Consequently, \( q \)'s \( VL \) at Line 14 of \( SC_q \) occurs during the time interval \( (t, t'') \) as well.

\textbf{Lemma 8} Let \( E \) be any finite execution history of the algorithm in Figure 2. Let \( p \) be some process, and \( LL_p \) some \( LL \) operation by \( p \) in \( E \). Let \( t \) be the time when \( p \) executes Line 1 of \( LL_p \), and \( t' \) the time when \( p \) executes Line 4 of \( LL_p \). If the condition at Line 7 of \( LL_p \) succeeds (i.e., \( VL(X) \) returns false), let \( SC_q \) be the \( SC \) operation on \( O \) that writes into \( \text{Help}[p] \) during \( (t, t') \), and let \( t'' \in (t, t') \) be the time when the \( VL \) operation on \( X \) at Line 14 of \( SC_q \) is performed. Then, the value that \( LL_p \) returns is the value of \( O \) at time \( t'' \).

\textbf{Proof.} Let \( q \) be the process executing \( SC_q \). Let \( LL_q \) be \( q \)'s latest \( LL \) operation on \( O \) before \( SC_q \). Since the \( VL \) operation on \( X \) at Line 14 of \( SC_q \) succeeds, it means that either the condition at Line 7 of \( LL_q \) failed, or that Line 7 of \( LL_q \) was never executed. In the first case, let \( t_q \) be the time when \( q \) executes Line 5 of \( LL_q \). In the second case, let \( t_q \) be the time when \( q \) executes Line 2 of \( LL_q \). In either case, by Lemmas 5 and 6, \( LL_q \) returns the value of \( O \) at time \( t_q \). Let \( v \) be the value returned by \( LL_q \). Since the \( VL \) operation on \( X \) at Line 14 of \( SC_q \) succeeds, it means that \( v \) is the value of \( O \) at time \( t'' \) as well.

Let \( t'_q \) be the time just before \( q \) starts executing Line 11 of \( LL_q \). Let \( t''_q \) be the time when \( q \) executes the \( SC \) operation on \( \text{Help}[p] \) at Line 15 of \( SC_q \). Let \( b \) be the value of \( mybuf_q \) at time \( t'_q \). Notice that, by the algorithm, the only places where \( \text{BUF}[b] \) can be modified is either at Line 11 of some \( LL \) operation, or at Line 17 of some \( SC \) operation. By invariant (I1), we know that during \( (t'_q, t''_q) \), no process \( r \neq q \) can be at Line 11 or 17 with \( mybuf_r \equiv b \). Therefore, \( \text{BUF}[b] \) holds the value \( v \) at all times during \( (t'_q, t''_q) \). Since \( mybuf_q \) doesn’t change during \( (t'_q, t''_q) \) as well, it means that \( q \) writes \( (0, b) \) into \( \text{Help}[p] \) at time \( t''_q \in (t, t') \). Since, by Lemma 2, no other process writes into \( \text{Help}[p] \) during \( (t, t') \), it means that \( p \) reads \( b \) at Line 4 of \( LL_p \) (at time \( t'') \). Let \( t''' \) be the time when \( p \) executes Line 7 of \( LL_p \). Then, by invariant (II), we know that during \( (t''', t'') \) no process \( r \) can be at Line 11 or 17 with \( mybuf_r \equiv b \). Therefore, \( \text{BUF}[b] \) holds the value \( v \) at all times during \( (t'''', t'') \). So, at Line 6 of \( LL_p \), \( p \) writes into \( \text{retval} \) the value \( v \), which is the value of \( O \) at time \( t''' \).

\( \square \)

\textbf{Lemma 9} Let \( E \) be any finite execution history of the algorithm in Figure 2. Let \( p \) be some process, and \( LL_p \) some \( LL \) operation by \( p \) in \( E \). Let \( LP(\text{LL}_p) \) be the linearization point for \( LL_p \). Then, \( LL_p \) returns the value of \( O \) at \( LP(\text{LL}_p) \).

\textbf{Proof.} This lemma follows immediately from Lemmas 5, 6, and 8.

\textbf{Lemma 10} Let \( E \) be any finite execution history of the algorithm in Figure 2. Let \( p \) be some process, and \( SC_p \) some \( SC \) operation by \( p \) in \( E \). Let \( LL_p \) be the latest \( LL \) operation by \( p \) to precede \( SC_p \). Then, \( SC_p \) succeeds if and only if there does not exist some other successful \( SC \) operation \( SC' \) such that \( LP(SC') \in (LP(LL_p), LP(SC_p)) \).

\textbf{Proof.} If \( SC_p \) succeeds, then the \( SC \) operation on \( X \) at Line 19 of \( SC_p \) succeeds. Then, \( LP(LL_p) \) is either at Line 2 of \( LL_p \) or at Line 5 of \( LL_p \). In either case, \( X \) doesn’t change during \( (LP(LL_p), LP(SC_p)) \), and hence no other successful operation is linearized during \( (LP(LL_p), LP(SC_p)) \). If \( SC_p \) fails, we examine three possibilities, based on where the \( LP(LL_p) \) is. If \( LP(LL_p) \) is at Line 2 or Line 5 of \( LL_p \), the fact that \( SC_p \) fails means that \( X \) changes during \( (LP(LL_p), LP(SC_p)) \). Hence, there exists a successful \( SC \) operation \( SC' \) such that \( LP(SC') \in (LP(LL_p), LP(SC_p)) \). If \( LP(LL_p) \) is between Lines 2 and 4 of \( LL_p \) (the third linearization case), then the \( VL \) operation on \( X \) at Line 7 of \( LL_p \) failed, and hence \( X \) changes during \( (LP(LL_p), LP(SC_p)) \). Hence, there exists a successful \( SC \) operation \( SC' \) such that \( LP(SC') \in (LP(LL_p), LP(SC_p)) \).

The proof of the following lemma is identical to the proof of Lemma 10, and is therefore omitted.

\textbf{Lemma 11} Let \( E \) be any finite execution history of the algorithm in Figure 2. Let \( p \) be some process, and \( VL_p \) some \( VL \) operation by \( p \) in \( E \). Let \( LL_p \) be the latest \( LL \) operation by \( p \) to precede \( VL_p \). Then, \( VL_p \) succeeds if and only if there does not exist some successful \( SC \) operation \( SC' \) such that \( LP(SC') \in (LP(LL_p), LP(VL_p)) \).

\textbf{Theorem 1} The \( N \)-process wait-free implementation in Figure 2 of a \( W \)-word \( LL/SC/VL \) variable is linearizable. The time complexity of \( LL, SC \) and \( VL \) operations on \( O \) are \( O(W) \), \( O(W) \) and \( O(1) \), respectively. The implementation requires \( O(NW) \) 64-bit safe registers and \( O(N) \) 64-bit \( LL/SC/VL \) read objects.
Proof. This theorem follows immediately from Lemmas 9, 10, and 11. □

References


