Efficient and Practical Constructions of 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)

**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.
Abstract

Over the past decade, LL/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 or RLL/RSC (e.g. POWER4, MIPS, SPARC, IA-64). To bridge this gap, this paper presents two efficient wait-free algorithms for implementing 64-bit LL/SC objects from 64-bit CAS or RLL/RSC objects.

Our first algorithm is practical: it has a small, constant time complexity (of 4 for LL and 5 for SC) and a space overhead of only 4 words per process. This algorithm uses unbounded sequence numbers. For theoretical interest, we also present a more complex bounded algorithm that still guarantees constant time complexity and $O(1)$ space overhead per process.

The LL/SC primitive is free of the well-known ABA problem that afflicts CAS. By efficiently implementing LL/SC words from CAS words, this work presents an efficient general solution to the ABA problem.
1 Introduction

Over the past decade, LL/SC have emerged as the most suitable synchronization instructions for the design of lock-free algorithms. In fact, most lock-free algorithms designed in recent years are based on LL/SC [1, 4, 6, 9, 13, 14, 18, 19, 21]. These algorithms assume that LL/SC instructions satisfy the semantics described in Figure 1. However, current architectures do not support LL/SC instructions with these semantics; instead, they support either compare&swap, also known as CAS (e.g., UltraSPARC [10], and Itanium [7]) or restricted versions of LL/SC (e.g., POWER4 [8], MIPS [23], and Alpha [22] processors). Although the restrictions on LL/SC vary from one architecture to another, Moir [17] noted that the LL/SC instructions supported by real machines, henceforth referred to as RLL/RSC, satisfy the restricted semantics stated in Figure 2.

Since CAS suffers from the well-known ABA problem [5] and RLL/RSC impose severe restrictions on their use [17], 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 a wait-free algorithm that implements LL/SC memory words (i.e., 64-bit 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.

Of the existing algorithms for the above problem, the most efficient one is due to Moir [17]. His algorithm runs in constant time and has no space overhead. However, it can only implement small (e.g., 16 to 24 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 version number along with the object’s value in the same memory word. Since version number takes up about 40 bits, only 16 to 24 bits are left for the value field. Our approach is to store the value and the version number in separate memory words, thus enabling values to be as big as 64 bits. This separation of value and version number, of course, makes it hard to ensure atomicity of concurrent operations, but our algorithms meet this challenge. In the following, we state our results and their significance, and compare them with existing work.

1.1 An unbounded algorithm

Our first result is a wait-free algorithm that implements a 64-bit LL/SC object from a 64-bit CAS object and registers. The algorithm is efficient in both time and space. The time complexity of LL and SC operations are small constants (4 and 5, respectively). In terms of space, the algorithm uses a single CAS object and, additionally, four registers per process.

This result shows, for the first time, a practical way of simulating a 64-bit LL/SC object using CAS, incurring only a small constant space overhead per process and a small constant factor slowdown.

Using the technique introduced by Moir [17], we can replace CAS with RLL/RSC, thereby obtaining an implementa-

- $LL(O)$ returns $O$’s value.
- $SC(O, v)$ by process $p$ “succeeds” if and only if no process performed a successful SC on $O$ since $p$’s latest $LL$ on $O$. If SC succeeds, it changes $O$’s value to $v$ and returns $true$. Otherwise, $O$’s value remains unchanged and SC returns $false$.
- $VL(O)$ returns $true$ if and only if no process performed a successful SC on $O$ since $p$’s latest $LL$ on $O$.
- $CAS(X, u, v)$ 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 1: Definitions of operations LL/SC/VL and CAS

- RLL/RSC are similar to $LL$ and $SC$, with two differences [17]: (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 2: Definition of operations RLL/RSC

tion of a 64-bit LL/SC object from RLL/RSC.

This result is significant for the ABA-problem [5], which arises in algorithms that use CAS. The ABA-problem has been known for 20 years, and is briefly described as follows. Suppose that the steps of processes $p$ and $q$ are interleaved in the following manner: $p$ reads value $u$ in variable $X$, $q$ changes $X$ to $v$ and then back to $u$, and then $p$ performs $CAS(X, u, *)$. In this scenario, $p$’s CAS succeeds even though $X$ was modified after $p$’s read. This problem would not occur if $p$’s read and CAS operations were replaced by $LL$ and $SC$, respectively. Thus, by efficiently implementing a 64-bit LL/SC object from 64-bit CAS, our algorithm presents a general and permanent solution to the ABA-problem. In contrast, past solutions to the ABA-problem either were specific to the particular context [24, 25, 26] or solved the problem by storing a sequence number along with the value, which limits the range of values that can be stored [16, 17, 20].

1.2 A bounded algorithm

The previous algorithm maintains a sequence number that grows without bound. Although the use of unbounded sequence numbers is not a concern in practice, from a theoretical standpoint it is desirable to design an algorithm whose variables take on only bounded values that fit into real memory words. Our second algorithm achieves this property while still guaranteeing constant time complexity and constant space overhead per process. We obtain this algorithm by composing three reductions, two of which are novel. One reduction,
which we believe is significant in its own right, is stated next.

1.3 Reducing LL/SC to Weak LL/SC

Anderson and Moir [2] introduced a weak version of LL, denoted WLL, which is described as follows. A WLL operation by a process \( p \) is not obligated to return the object’s value if \( p \)’s subsequent SC operation is sure to fail; in this case, the WLL may simply return the identity of a process whose successful SC took effect during the execution of that WLL. This weaker LL operation is good enough for some applications [2], but not for all. For example, the closed objects construction [6], the construction of \( f \)-arrays and snapshots [13], the abortable mutual exclusion algorithm [14], and some universal constructions [1, 9, 18, 19, 21] require the standard LL operation.

We discovered a surprisingly close connection between LL/SC and WLL/SC: a WLL/SC object can be transformed into an LL/SC object, incurring only a small overhead in time and space. Specifically, we present a wait-free algorithm that, for any \( m \) \( > \) 0, transforms an \( m \)-bit WLL/SC object into an \( m \)-bit LL/SC object; the algorithm has a constant time complexity (of 5 for LL and 1 for SC), and a space overhead of only one \( m \)-bit register per process.

1.4 Previous work

The earliest wait-free algorithm for implementing an LL/SC object from CAS objects is due toIsraeli and Rappoport [11]. Their algorithm makes an unrealistic assumption that \( N \) bits can be stored in a single memory word, where \( N \) is the maximum number of processes for which the algorithm is designed. Furthermore, both LL and SC have a worst case time complexity of \( O(N) \). The more efficient known algorithms are due to Anderson and Moir, and were presented in Figure 1 of [3], Figure 2 of [2], and Figures 4 and 7 of [17]. In the following, we refer to these four algorithms by the names AM1, AM2, M1 and M2, respectively. As we now explain, these algorithms either implement smaller (than 64-bit) LL/SC objects or incur excessive space overhead.

Algorithms AM1 and M2 use only bounded registers, but have two drawbacks: (1) the space overhead is \( O(N) \) per process, and (2) they store a tag along with the value in the same \( 64 \)-bit memory word, thereby limiting the range of values that can be stored in the LL/SC object. Specifically, AM1 and M2 use tags of \( 1 + 3 \log N \) bits and \( 1 + 2 \log N \) bits, respectively. Therefore, assuming \( N = 1000 \), they can only implement a 33-bit and a 43-bit LL/SC object, respectively.

Algorithm M1 is extremely efficient: it has constant time complexity and incurs no space overhead. However, this algorithm stores an unbounded tag along with the value in the same memory word. Assuming 48 bits are used up for the tag, only 16 bits are left for the value.

By composing any of these small LL/SC object implementations with the multi-word LL/SC construction of AM2, one can implement a 64-bit LL/SC variable. The resulting implementations, however, have a space overhead of \( O(N) \) per process (more precisely, at least \( 16N \) shared variables are needed per process, in contrast to 4 variables per process that our algorithm requires).

Finally, a recent algorithm by Luchangco, Moir and Shavit [15] implements a 63-bit LL/SC object, but it is only non-blocking and not wait-free.

2 A practical 64-bit LL/SC implementation

Figure 3 presents our first algorithm for implementing a 64-bit LL/SC object. We begin by providing an intuitive description of how this algorithm works.

2.1 How the algorithm works

The algorithm implements a 64-bit LL/SC object \( O \). Central to the implementation is the variable \( X \) that supports CAS and read operations. In addition there are four atomic registers at each process \( p \) — \( \text{val}_p[0], \text{val}_p[1], \text{oldval}_p \) and \( \text{oldseq}_p \)—that are written to only by \( p \) but may be read by any process. The meanings of these variables are described as follows.

The algorithm associates a tag with every successful SC operation on \( O \). A tag consists of a process id and a sequence number. Specifically, the tag associated with a successful SC operation is \([p, k]\) if it is the \(k\)th successful SC operation by process \( p \). The variable \( X \) always contains the tag corresponding to the latest successful SC.

Suppose that the current value of \( X \) is \([p, k]\) (the last successful SC was performed by \( p \) and \( p \) performed \( k \) successful SC operations so far). Our algorithm ensures that the value written by the \(k\)th successful SC by \( p \) is in \( \text{val}_p[0] \) if \( k \) is even, or in \( \text{val}_p[1] \) if \( k \) is odd; i.e., the value is made available in \( \text{val}_p[k \mod 2] \). The registers \( \text{oldval}_p \) and \( \text{oldseq}_p \) hold an older value and its sequence number, respectively. Specifically, if \( p \) has so far performed \( k \) successful SC operations, \( \text{oldseq}_p \) and \( \text{oldval}_p \) contain, respectively, the number \( k - 1 \) and the value written by the \((k - 1)\)th successful SC by \( p \).

In addition to the shared variables just described, each process \( p \) has two local variables, \( \text{seq}_p \) and \( \text{tag}_p \), described as follows. The value of \( \text{seq}_p \) is the sequence number of \( p \)’s next SC operation: If \( p \) has performed \( k \) successful SC operations so far, \( \text{seq}_p \) has the value \( k + 1 \). (Thus, sequence numbers in our algorithm are local: \( p \)’s sequence number is based on the number of successful SCs performed by \( p \), not by the system as a whole.) The value of \( \text{tag}_p \) is the value of \( X \) read by \( p \) in its latest LL operation.

Given this representation, the variables are initialized as follows. Let \( \text{vinit} \) denote the desired initial value of the implemented object \( O \). We pretend that process 0 performed an “initializing SC” to write the value \( \text{vinit} \). Accordingly, \( X \) is initialized to \([0,1],\text{val}_0[1] \) to \( \text{vinit}, \text{oldseq}_0 \) to 0, and \( \text{seq}_0 \) to 2. For each process \( p \neq 0 \), \( \text{seq}_p \) is initialized to 1. All other variables are arbitrarily initialized.

We now explain the procedure \( SC(p, O, v) \) that describes how process \( p \) performs an SC operation on \( O \) to attempt to
Types
valuetype = 64-bit number
seqnumtype = (64 − log N)-bit number
tagtype = record pid: 0..N − 1; seqnum: seqnumtype end

Shared variables
X: tagtype (X supports read and CAS operations)

For each p ∈ {0, . . . , N − 1}, we have four single-writer, multi-reader registers:
val_p[0], val_p[1], oldval_p: valuetype
oldseq_p: seqnumtype

Local persistent variables at each p ∈ {0, . . . , N − 1}
tag_p: tagtype
seq_p: seqnumtype

Initialization
X = [0, 1]
val0[1] = vinit, the desired initial value of O
oldseq0 = 0
seq0 = 2
For each p ∈ {1, . . . , N − 1} seq_p = 1

procedure LL(p, O) returns valuetype
1: tag_p = X
   Let [q, k] = [tag_p.pid, tag_p.seqnum]
2: v = val_q[k mod 2]
3: k′ = oldseq_q
4: if (k′ = k − 2) ∨ (k′ = k − 1) return v
5: v′ = oldval_q
6: return v′

procedure SC(p, O, v) returns boolean
7: val_p[seq_p mod 2] = v
8: if CAS(X, tag_p, [p, seq_p])
9: oldval_p = val_p[(seq_p − 1) mod 2]
10: oldseq_p = seq_p − 1
11: seq_p = seq_p + 1
12: return true
13: else return false

procedure VL(p, O) returns boolean
14: return X = tag_p

Figure 3: An unbounded implementation of the 64-bit LL/SC variable O using a 64-bit CAS variable and 64-bit registers

change O’s value to v. First, p makes available the value v in val_p[0] if the sequence number is even, or in val_p[1] if the sequence number is odd (Line 7). Next, p tries to make its SC operation take effect by swinging the value in X from the tag that p had witnessed in its latest LL operation to the tag corresponding to its current SC operation (Line 8). If the CAS operation fails, it follows that some other process performed a successful SC after p’s latest LL. In this case, 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. To remain faithful to the previously described meanings of the variables oldval_p and oldseq_p, p writes in oldval_p the value written by p’s earlier successful SC (Line 9) and writes in oldseq_p the sequence number of that SC (Line 10). (Since the sequence number for p’s current successful SC is seq_p, it follows that the sequence number for p’s earlier successful SC is seq_p − 1 and the value written by that SC is in val_p[(seq_p − 1) mod 2]; this justifies the code on Lines 9 and 10.) Next, p increments its sequence number (Line 11) and signals successful completion of the SC by returning true (Line 12).

We now turn to the procedure LL(p, O) that describes how process p performs an LL operation on O. In the following, let SC_q,i denote the i-th successful SC by process q and v_q,i denote the value written in O by SC_q,i. First, p reads X to obtain the tag [q, k] corresponding to the latest successful SC operation, SC_q,k (Line 1). Notice that, at the instant when p performs Line 1, the variable val_q[k mod 2] holds the value v_q,k. Notice further that val_q[k mod 2] is not modified until q initiates an SC operation with seq_q = k + 2. In particular, the value of val_q[k mod 2] is guaranteed to be v_q,k until q completes SC_q,k+1.

In an attempt to learn v_q,k, p reads val_q[k mod 2] (Line 2). By the observation in the previous paragraph, if p is not too slow and executes Line 2 before q completes SC_q,k+1, the value v read on Line 2 will indeed be v_q,k. Otherwise the value v cannot be trusted. To resolve this ambiguity, p must determine if q has completed SC_q,k+1 yet. To make this determination, p reads the sequence number k’ in oldseq_q (Line 3). If k’ < k, it follows that SC_q,k+1 has not yet completed even if it had been already initiated (because, by Line 10, SC_q,k+1 writes k into oldseq_q). It follows that the value v obtained on Line 2 is v_q,k. So, p terminates the LL operation, returning v (Line 4).

If k’ ≥ k, q must have completed SC_q,k+1, its (k + 1)th successful SC. It follows that the value in oldval_q is v_q,k

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D

p

T

D
or a later value (more precisely, the value in oldval_q is v_{q,i} for some i \geq k). Therefore, the value in oldval_q is not too stale for p's LL to return. Accordingly, p reads the value v' of oldval_q (Line 5) and returns it (Line 6). Although v' is a recent enough value of O for p's LL to legitimately return, it is important to note that v' is not the current value of O. This is because the algorithm moves a value into oldval_q only after it is no longer the current value. Since the value v' that p's LL returns on Line 6 is not the current value, p's subsequent SC must fail (by the specification of LL/SC). Our algorithm satisfies this requirement because, when p's subsequent SC performs Line 8, the CAS operation fails since tag_p is [q, k] and the value of X is not [q, k] anymore (the value of X is not [q, k] because, by the first sentence of this paragraph, q has completed its (k + 1)th successful SC). This completes the description of how LL is implemented.

The VL operation by p is simple to implement: p returns true if and only if the tag in X has not changed since p's latest LL operation (Line 14).

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

Theorem 1 The wait-free algorithm in Figure 3 implements a linearizable 64-bit LL/SC object from a single 64-bit CAS object and an additional four registers per process. The time complexity of LL, SC and VL operations are 4, 5 and 1, respectively.

2.2 Remarks

2.2.1 Sequence number wrap-around

The 64-bit variable X stores in it a process number and a sequence number. Even if there are as many as 16000 processes sharing the implementation, only 14 bits are needed for storing the process number, leaving 50 bits for the sequence number. In our algorithm, for seq_q, to wrap around, p must perform 2^{50} successful SC operations. If p performs a million successful SC operations each second, it takes 32 years for seq_q to wrap around! Wrapping around is therefore not a practical concern.

2.2.2 Using RLL/RSC instead of CAS

Using Moir's idea [17], it is straightforward to replace the CAS instruction (on Line 8) in our algorithm with RLL/RSC instructions. Specifically, use the following code fragment in the place of Line 8. The repeat-until loop handles spurious RSC failures and terminates in a single iteration if there are no such failures.

```python
flag = false
repeat
    if RLL(X) \neq tag_p go to L
    flag = RSC(X, [p, seq_p])
until flag
L: if (flag)
```

2.2.3 Implementing read, write

It is straightforward to extend our algorithm in Figure 3 to implement read and write operations in addition to LL, SC and VL operations. Specifically, the implementation of Write(p, O, v) is the same as the implementation of SC(p, O, v) with the following changes: replace the CAS on Line 8 with write(X, [p, seq_p]) and remove Lines 12 and 13. The implementation of Read(p, O) is the same as the code for LL, except that on Line 1 the value of X is read into a local variable, different from tag_p (so that the read operation doesn't affect the success of the subsequent SC). The code for LL, SC and VL operations remains the same as in Figure 3.

Elsewhere we showed that incorporating the write operation into other known constructions of LL/SC variables is not algorithmically easy; it affects the code and the running time of LL and SC operations [12]. Thus, it is an interesting feature of our algorithm that it can be extended effortlessly to support the write operation.

3 Designing a bounded 64-bit LL/SC implementation

For the rest of this paper, the goal is to design a bounded algorithm that implements a 64-bit LL/SC object using 64-bit CAS objects and 64-bit registers. We achieve this goal in four steps:

1. Implement a 64-bit LL/SC object from a 64-bit WLL/SC object and 64-bit registers.
2. Implement a 64-bit WLL/SC object from a (1-bit, pid)-LL/SC object (which will be described later).
3. Implement a (1-bit, pid)-LL/SC object from a 64-bit CAS object and registers.
4. This step is trivial: simply compose the implementations from the above steps. This composition results in an implementation of a 64-bit LL/SC object from 64-bit CAS objects and 64-bit registers.

Interestingly, as we will show in the next three sections, the implementations in the first three steps have O(1) time complexity and O(1) space overhead per process. As a result, the 64-bit LL/SC implementation obtained in the fourth step also has O(1) time complexity and O(1) space overhead per process, as desired.

4 Implementing 64-bit LL/SC from 64-bit WLL/SC

Recall that a WLL operation, unlike LL, is not always required to return the value of the object: if the subsequent SC operation is sure to fail, the WLL may simply return the identity of a process whose successful SC took effect during the execution of that WLL. Thus, the return value of WLL is of the form [flag, v], where either (i) flag = success and v is the...
value of the object \( O \), or (ii) \( \text{flag} = \text{failure} \) and \( v \) is the \( i \)d of a process whose SC took effect during the WLL.

Figure 4 describes the algorithm that implements a 64-bit LL/SC object \( O \). The algorithm uses a single 64-bit WLL/SC variable \( X \) and, for each process \( p \), a single 64-bit atomic register \( \text{lastVal}_p \). In the following, we make an important observation and then describe the intuition underlying the algorithm.

### 4.1 Two obligations of LL

In any implementation, there are two conditions that an LL operation must satisfy to ensure correctness. Our algorithm 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 SCs take effect during the execution of this procedure, changing \( O \)'s value from \( v \) to \( v_1 \), \( v_1 \) to \( v_2 \), \ldots, \( v_{k-1} \) to \( v_k \). Then, any of \( v, v_1, \ldots, v_k \) 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 \( v, v_1, \ldots, v_{k-1} \): assuming that other processes do not perform successful SCs 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 includes exactly one SC operation on the variable \( X \) (Line 11) and the former succeeds if and only if the latter succeeds. Therefore, we can restate the two obligations on \( p \)'s LL procedure as follows: (\( O1 \)) It must return a valid value \( u \), and (\( O2 \)) If other processes do not perform successful SCs after \( p \)'s LL, \( p \)'s subsequent SC (or VL) on \( X \) must succeed if and only if the return value \( u \) is current.

### 4.2 How the algorithm works

The algorithm in Figure 4 is based on two key ideas: (A1) the current value of \( O \) is held in \( X \), and (A2) whenever a process \( p \) performs LL on \( O \) and obtains a value \( v \), it writes \( v \) immediately in \( \text{lastVal}_p \) unless \( p \) is certain that its subsequent SC on \( O \) will fail. With this in mind, consider the procedure \( \text{LL}(p, O) \) that \( p \) executes to perform an LL operation on \( O \). First, \( p \) tries to obtain \( O \)'s current value by performing a WLL on \( X \) (Line 1). There are two possibilities: either WLL returns the current value \( v \) in \( X \), or it fails, returning the id \( v \) of a process that performed a successful SC during the WLL. In the first case, \( p \) writes \( v \) in \( \text{lastVal}_p \) (to ensure A2) and then returns \( v \) (Lines 3 and 4). In the second case, let \( t \) be the instant during \( p \)'s WLL when process \( v \) performs a successful SC, and \( v' \) be \( O \)'s value immediately prior to \( t \) (that is, just before \( v \)'s successful SC). Then, \( v' \) is a valid value for \( p \)'s LL procedure to return. Furthermore, by A2, \( \text{lastVal}_p \) contains \( v' \) at time \( t \). So, when \( p \) reads \( \text{lastVal}_p \) and obtains \( \text{val} \) (Line 5), it knows that \( \text{val} \) must be either \( v' \) or some later value of \( O \). This means that \( \text{val} \) is a valid value for \( p \)'s LL procedure to return. However, \( p \) cannot return \( \text{val} \) yet because its subsequent SC is sure to fail (due to the failure of WLL in Line 1) and, therefore, \( p \) must ensure that \( \text{val} \) is not the latest value of \( O \). So, \( p \) performs another WLL (Line 6). If this WLL succeeds and returns \( v \), then as before \( p \) writes \( v \) in \( \text{lastVal}_p \) and returns \( v \) (Lines 8 and 9). Otherwise, \( p \) knows that some

---

### Figure 4: Implementation of the 64-bit LL/SC variable \( O \) using a 64-bit WLL/SC variable and 64-bit registers
current value of \( \mathcal{O} \). Furthermore, \( p \) knows that its subsequent SC will fail (due to the failure of WLL in Line 6). Thus, returning \( \text{val} \) fulfills both Obligations O1 and O2, justifying Line 10.

The VL operation by \( p \) is simple to implement: \( p \) returns true if and only if the VL on \( X \) returns true (Line 12).

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

**Theorem 2** The wait-free algorithm in Figure 4 implements a linearizable 64-bit LL/SC object from a single 64-bit WLL/SC object and one additional 64-bit register per process. The time complexity of LL, SC and VL operations are 4, 1 and 1, respectively.

## 5 Implementing 64-bit WLL/SC from (1-bit, pid)-LL/SC

A (1-bit, pid)-LL/SC object is the same as a 1-bit LL/SC object except that its LL operation, which we call BitPidLL, returns not only the 1-bit value written by the latest successful SC, but also the name of the process that performed that SC. Figure 5 presents a wait-free algorithm for implementing a 64-bit WLL/SC object from a (1-bit, pid)-LL/SC object and 64-bit registers. This algorithm is nearly identical to Anderson and Moir’s algorithm [2] that implements a multi-word WLL/SC object from a single word CAS object and atomic registers. In the following, we describe the intuition underlying the algorithm.

### 5.1 How the algorithm works

Let \( \mathcal{O} \) denote the 64-bit WLL/SC object implemented by the algorithm. Our implementation uses two registers per process \( p — \text{val}_p[0] \) and \( \text{val}_p[1] \) — which only \( p \) may write into, but any process may read. One of the two registers holds the value written into \( \mathcal{O} \) by \( p \)’s latest successful SC; the other register is available for use in \( p \)’s next SC operation ( \( p \)’s local variable \( \text{index}_p \) stores the index of the available register). Thus, over the \( N \) processes, there are a total of \( 2N \) \( \text{val} \) registers. Exactly which one of these contains the current value of \( \mathcal{O} \) (i.e., the value written by the latest successful SC) is revealed by the (1-bit, pid)-LL/SC object \( X \). Specifically, if \( [b, q] \) is the value of \( X \), then our algorithm ensures that \( \text{val}_q[b] \) contains the current value of \( \mathcal{O} \).

We now explain how process \( p \) performs an SC \((X) \) operation on \( \mathcal{O} \). First, \( p \) writes the value \( v \) into its available register (Line 6). Next, \( p \) tries to make its SC operation take effect by “pointing” \( X \) to this location (Line 7). If this effort fails, it means that some process performed a successful SC since \( p \)’s latest WLL. In this case, \( p \) terminates its SC operation returning false (Line 7). Otherwise, \( p \)’s SC operation has succeeded. So, \( \text{val}_p[\text{index}_p] \) now holds the value written by \( p \)’s latest successful SC. Therefore, to remain faithful to the representation described above, the index of the register available for \( p \)’s next SC operation is updated to be \( 1 - \text{index}_p \) (Line 8).

Finally, \( p \) returns true to reflect the success of its SC operation (Line 9).

We now turn to the procedure \( \text{WLL}(p, \mathcal{O}) \) that describes how process \( p \) performs a WLL operation on \( \mathcal{O} \). First, \( p \) performs an BitPidLL operation on \( X \) to obtain a value \( [b, q] \) (Line 1). By our representation, at the instant when \( p \) performs Line 1, \( \text{val}_q[b] \) holds the current value \( v \) of \( \mathcal{O} \). So, in an attempt to learn the value \( v \), \( p \) reads \( \text{val}_q[b] \) (Line 2). Then, it validates \( X \). If the validate succeeds, \( p \) is certain that the value read in Line 2 is indeed \( v \) and so, it returns \( v \) and signals success (Line 3). Otherwise, some process must have performed a successful SC after \( p \) had executed Line 1. Then, by the definition of WLL, \( p \) is not obligated to return a value; instead, it can signal failure and return the id of a process that performed a successful SC during \( p \)’s WLL. Such an id can be obtained simply by reading \( X \). So, \( p \) reads \( X \) and returns the id obtained, also signaling failure (Lines 4 and 5).

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

**Theorem 3** The wait-free algorithm in Figure 5 implements a 64-bit WLL/SC object from a single (1-bit, pid)-LL/SC object and an additional two 64-bit registers per process. The time complexity of LL, SC and VL operations are 4, 2 and 1, respectively.

## 6 Implementing (1-bit, pid)-LL/SC from 64-bit CAS

Figure 6 presents a wait-free algorithm for implementing a (1-bit, pid)-LL/SC object. This algorithm uses a procedure called \text{select}. As we will explain, the algorithm works correctly provided that \text{select} satisfies a certain property. This algorithm is inspired by, and is nearly identical to, Anderson and Moir’s algorithm in Figure 1 of [3]. The implementation of \text{select}, however, is novel and is crucial to obtaining constant space overhead per process.

Below we provide an intuitive description of how the algorithm works. Later we present two different implementations of the \text{select} procedure that offer different tradeoffs.

### 6.1 How the algorithm works

Let \( \mathcal{O} \) denote the (1-bit, pid)-LL/SC object implemented by the algorithm. The variables used in the implementation are described as follows.

- The variable \( X \) supports \text{read} and \text{CAS} operations, and contains a value of the form \([\text{seq}, \text{pid}, \text{val}] \), where \text{seq} is a sequence number, \text{pid} is a process id, and \text{val} is a 1-bit value. The first two entries (namely, the sequence number and the process id) constitute the tag, and the last two entries (namely, the process id and a 1-bit value) constitute the value of \( \mathcal{O} \).
- The variable \( A \) is an array, with one entry per process. A process \( p \) announces in \( A[p] \) the tag that it reads from \( X \) in its latest BitPidLL operation.
The variable \(s_{eq, p}\) is process \(p\)'s local variable. It holds the value of the next sequence number that \(p\) can use in a tag.

We now explain the procedure \(\text{BitPid}_{\text{LL}}(p, O)\) that describes how process \(p\) performs a \(\text{BitPid}_{\text{LL}}\) operation on \(O\). First, \(p\) reads \(X\) to obtain the current tag and value (Line 1). Next, \(p\) announces this tag in the array \(A\) (Line 2). Then, \(p\) reads \(X\) again (Line 3). There are two cases, based on whether the return values of the two reads are the same or not. If they are not the same, we linearize \(\text{BitPid}_{\text{LL}}\) at the instant when \(p\) performs the first read, and let \(p\) return that value at Line 4. In this case, since the value of \(O\) has changed after \(p\)'s LL operation, we must ensure that \(p\)'s subsequent SC operation will fail. This condition is indeed ensured by Line 5 of the algorithm. In the other case where the reads on Lines 1 and 3 return the same value, we linearize \(\text{BitPid}_{\text{LL}}\) at the instant when \(p\) performs the second read and let the LL operation return that value (at Line 4).

The implementation of the SC operation assumes that the \texttt{select} procedure satisfies the following property:

\textbf{Property 1} Let \(OP\) and \(OP'\) be any two consecutive \(\text{BitPid}_{\text{LL}}\) operations by process \(p\). If \(p\) reads \([s, q, v]\) from \(X\) in both Lines 1 and 3 of \(OP\), then process \(q\) does not write \([s, q, \ast]\) into \(X\) after \(p\) executes Line 3 of \(OP\) and before it invokes \(OP'\).

We now describe how process \(p\) performs \(SC(v)\) on \(O\). In the following, let \(OP\) denote \(p\)'s latest execution of the \(\text{BitPid}_{\text{LL}}\) operation on \(O\). First, \(p\) compares the two values that it read from \(X\) during \(OP\) (Line 5). If these values are different, as already explained, \(p\)'s SC operation must fail, and Line 5 ensures this outcome. To understand Line 6, we make an observation that follows from Property 1: the value of \(X\) is still \(old_p\) if and only if no process wrote into \(X\) after the point where \(p\)'s latest \(\text{BitPid}_{\text{LL}}\) operation took effect (at Line 3 of \(OP\)). It follows that \(p\)'s current SC operation should succeed if and only if the CAS on Line 6 succeeds. Accordingly, if the CAS fails, \(p\) terminates the SC operation returning \texttt{false} (Line 6). On the other hand, if the CAS succeeds, \(p\) obtains a new sequence number to be used in \(p\)'s next SC operation (Line 7), and completes the SC operation returning \texttt{true} (Line 8).

The implementation of the VL operation (Line 9) has the same justification as the SC operation. Finally, the implementation of the \(\text{BitPid}_{\text{read}}\) operation (Lines 10 and 11) is immediate from our representation.

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

\textbf{Theorem 4} If the \texttt{select} procedure is implemented to satisfy Property 1, then the wait-free algorithm in Figure 6 implements a linearizable (1-bit, pid)-LL/SC object from 64-bit CAS objects and 64-bit registers. If \(t\) is the time complexity of \texttt{select}, then the time complexity of \(\text{BitPid}_{\text{LL}}, SC, VL\) and \(\text{BitPid}_{\text{read}}\) operations are \(3, 1 + t, 1 + t, 1\), respectively. If \(s\) is the per-process space overhead of \texttt{select}, then the per-process space overhead of the algorithm is \(1 + s\).
6.2 Why \( X \) is read twice

To execute a BitPid\_LL operation, notice that a process \( p \) reads \( X \), announces the tag obtained in \( A[p] \), and reads \( X \) again (Lines 1–3). As we will see in the next section, this double reading of \( X \), with the tag announced between the reads, is crucial to our ability to implement the \texttt{select} procedure. Intuitively, the usefulness of the code sequence on Lines 1–3 is explained as follows. Suppose that \( p \) reads the same tag \( t \) at Lines 1 and 3. When subsequently executing an SC operation, \( p \) determines the success or failure of its SC based on whether the tag in \( X \) is still \( t \) or not. Clearly, such a strategy goes wrong if \( X \) has been modified several times (between \( p \)'s LL and SC) and the tag \( t \) has simply reappeared in \( X \) because of reuse of that tag. Fortunately, this undesirable scenario is preventable because \( p \) publishes the tag \( t \) in \( A[p] \) (at Line 2) even before it reads that tag at Line 3, where \( p \)'s LL operation takes effect. So, we can prevent the undesirable scenario by requiring processes not to reuse the tags published in the array \( A \) (this requirement will be enforced by the implementation of \texttt{select}).

6.3 An implementation of \texttt{select}

In this section, we design an algorithm that implements the \texttt{select} procedure. This design is challenging because it must guarantee several properties: the \texttt{select} procedure must satisfy Property 1, be wait-free, and have constant time complexity and constant per-process space overhead. The algorithm of this section, presented in Figure 7, guarantees all of these properties, but only works for at most \( 2^{15} = 32,768 \) processes. A more complex algorithm, presented in the next section, can handle a maximum of \( 2^{19} = 524,288 \) processes.

To explain our algorithms, we introduce the notion of a sequence number being unsafe for a process.

Let \( s \) be any sequence number, \( q \) be any process, and \( t \) be any point in time. We say \( s \) is unsafe for \( q \) at time \( t \) if the following scenario is possible:

\textbf{Scenario:} At some \( t' > t \), \( q \)'s call to \texttt{select} returns \( s \) for the first time after \( t \). The subsequent writing of \([s, q, \star]\) in \( X \) by \( q \) (at Line 6 of the algorithm in Figure 6) causes Property 1 to be violated.

A sequence number \( s \) is safe for \( q \) at time \( t \) if \( s \) is not unsafe for \( q \) at \( t \). Notice that if \( s \) is safe for \( q \) at time \( t \), it remains safe until \( q \) writes \([s, q, \star]\) in \( X \) for the first time after \( t \) (which happens only after \( q \)'s call to \texttt{select} returns \( s \) for the first time after \( t \)). An interval is safe for \( q \) at time \( t \) if every sequence number in the interval is safe for \( q \) at time \( t \).

In both of our algorithms, the main idea is as follows. At all times, each process \( p \) maintains a current safe interval of a certain size \( \Delta \); initially, this interval is \([0, \Delta]\). Each call to \texttt{select} by \( p \) returns a sequence number from \( p \)'s current safe interval. By the time all numbers in \( p \)'s current safe interval are returned (which won’t happen until \( p \) calls \texttt{select} \( \Delta \))

---

Figure 6: A bounded implementation of the 1-bit “pid” LL/SC variable using a 64-bit CAS object and 64-bit registers, based on Anderson and Moir’s algorithm [3]
times), \( p \) determines a new safe interval of size \( \Delta \) and makes that interval its current safe interval. Since \( p \)'s current safe interval is not exhausted until \( p \) calls \texttt{select} \( \Delta \) times, our algorithms use a lazy approach to finding the next safe interval: the work involved in identifying the next safe interval is distributed evenly over the \( \Delta \) calls to \texttt{select}. Together with an appropriate choice of \( \Delta \), this strategy helps achieve constant time complexity for \texttt{select}.

The manner in which the next safe interval is determined is different in our two algorithms. The main idea in the first algorithm is as follows. Let \( (k, k + \Delta) \) be \( p \)'s current safe interval. Then, \((k + \Delta, k + 2\Delta)\) is the first interval that \( p \) tests for safety. If there is evidence that this interval is not safe, then the next \( \Delta \)-sized interval, namely, \([k + 2\Delta, k + 3\Delta]\) is tested for safety. The above steps are repeated until a safe interval is found. It remains to be explained how \( p \) tests whether a particular interval \( I \) is safe. To perform this test, \( p \) reads each element \( a \) of array \( \Lambda \) (recall that an element of \( \Lambda \) contains both a process id and a sequence number). If \( a = [s, p, \star] \), then it is possible that some process read \([s, p, \star]\) at Lines 1 and 3 of its latest BitPidLL operation, thereby making \( s \) potentially unsafe for \( p \). Therefore, in our algorithm, \( p \) deems the interval \( I \) to be safe if and only if it reads no element \( a \) such that \([a.pid = p \land a.seq \in I]\). To ensure \( O(1) \) time complexity for the \texttt{select} procedure, \( p \) reads \( \Lambda \) in a lazy manner: it reads only one element of \( \Lambda \) in each invocation of \texttt{select}.

In the following, we explain how the above high level ideas are implemented in our algorithm and why these ideas work.

### 6.3.1 How the algorithm works

Our selection algorithm is presented in Figure 7. Let TestInt\(_p\) denote the interval that \( p \) is currently testing for safety. The algorithm uses three persistent local variables, described as follows:

- \( \texttt{val}_p \) is a sequence number from \( p \)'s current safe interval which was returned by \( p \)'s most recent invocation of \texttt{select}.
- \( \texttt{nextStart}_p \) is the start of the interval TestInt\(_p\). Thus, TestInt\(_p\) is the interval \([\texttt{nextStart}_p, \texttt{nextStart}_p + \Delta]\).
- \( \texttt{procNum}_p \) indicates how far the test of safety of TestInt\(_p\) has progressed. Specifically, if \( \texttt{procNum}_p = k \), it means that the array entries belonging to processes \( 0, 1, \ldots, k - 1 \) (namely, \( \Lambda[0], \Lambda[1], \ldots, \Lambda[k - 1] \)) have not presented any evidence that TestInt\(_p\) is unsafe.

The algorithm works as follows. First, \( p \) reads the next element \( a \) of the array \( \Lambda \) (Line 10). If the process id in \( a \) is \( p \) and the sequence number in \( a \) belongs to the interval TestInt\(_p\), then the interval is potentially unsafe. Therefore, if the condition on Line 11 holds, \( p \) abandons the interval TestInt\(_p\) as unsafe. At this point, the \( \Delta \)-sized interval immediately following TestInt\(_p\) becomes the new interval to be tested for safety. To this end, \( p \) updates \( \texttt{nextStart}_p \) to the beginning of this interval (Line 12) and resets \( \texttt{procNum}_p \) to 0 (Lines 13). On the other hand, if the condition on Line 11 does not hold, it means that

![Figure 7: A simple selection algorithm](image)

the element at the position \( \texttt{procNum}_p \) of array \( \Lambda \) (namely, \( a \)) presents no evidence that TestInt\(_p\) is unsafe. To reflect this fact, \( p \) increments \( \texttt{procNum}_p \) (Line 14).

At Line 15, if \( \texttt{procNum}_p \) is \( N \), it follows from the meaning of \( \texttt{procNum}_p \) that \( p \) read the entire array \( \Lambda \) and has not found any evidence that the interval TestInt\(_p\) is unsafe. In this case, \( p \) performs the following actions. It switches to TestInt\(_p\) as its current safe interval and let \texttt{select} return the first sequence number in this interval (Lines 17 and 20). The \( \Delta \)-sized interval immediately following this new current safe interval becomes the new interval to be tested for safety. To this end, \( p \) updates \( \texttt{nextStart}_p \) to the beginning of this interval (Line 18) and resets \( \texttt{procNum}_p \) to 0 (Lines 19).

At Line 15, if \( \texttt{procNum}_p \) is not yet \( N \), \( p \) is not sure yet that TestInt\(_p\) is a safe interval. Therefore, it keeps the current safe interval as it is and simply returns the next value from that interval (Lines 16 and 20).

Notice that after \( p \) adopts an interval \( I \) to be its current safe interval at some time \( t \), \( p \)'s calls to \texttt{select} return successive sequence numbers starting from the first number in \( I \). Therefore, if \( p \) makes at most \( k \leq \Delta \) calls to \texttt{select} before adopting a new interval \( I' \) as its current safe interval, then all numbers returned (by the \( k \) calls to \texttt{select}) are from \( I \) and no number is returned more than once. Since \( I \) was safe for \( p \) at time \( t \), it follows that the numbers returned by the \( k \) calls to \texttt{select} do not lead to a violation of Property 1. By the
above discussion, the correctness of the algorithm rests on the following two claims:

- After a process $p$ adopts an interval $I$ to be its current safe interval, $p$ makes at most $\Delta$ calls to select before adopting a new interval $I'$ as its current safe interval.
- At the time that $p$ adopts $I'$ to be its current safe interval, $I'$ is indeed safe for $p$.

The above claims are justified in the next two subsections.

6.3.2 A new interval is identified quickly

Suppose that at time $t$ a process $p$ adopts an interval $I$ to be its current safe interval. Let $t' > t$ be the earliest time when $p$ adopts a new interval $I'$ as its current safe interval. Then, we claim:

Claim A: During the time interval $[t, t')$, process $p$ makes at most $\Delta$ calls to select (which return distinct sequence numbers from the interval $I$). Furthermore, $I$ and $I'$ are disjoint.

To prove the claim, we make a crucial subclaim which states that, when a process $p$ searches for a next safe interval, no process $q$ can cause $p$ to abandon more than two intervals as unsafe.

Subclaim: If $I_1, I_2, \ldots, I_m$ are the successive intervals that $p$ tests for safety during $[t, t')$, then at most two of $I_1, I_2, \ldots, I_m$ are abandoned by $p$ as unsafe on the basis of the values read in $\mathbb{A}[q]$, for any $q$.

First we argue that the subclaim implies the claim. By the subclaim, during $[t, t')$, $p$ abandons at most $2N$ intervals as unsafe. Notice that, in the worst case, $p$ invokes select $N$ times before abandoning any interval as unsafe (the worst case arises if none of $\mathbb{A}[0], \mathbb{A}[1], \ldots, \mathbb{A}[N-2]$ provides any evidence of unsafety, and $\mathbb{A}[N-1]$ does). It follows from the above two facts that, during $[t, t')$, $p$ invokes select at most $2N \cdot N$ times before it begins testing an interval that is sure to pass the test. Since the testing of this final interval occurs over $N$ calls to select, it follows that, during $[t, t')$, $p$ invokes select at most $2N^2 + N = (2N + 1)N = \Delta$ times before it identifies the next safe interval $I'$. Hence, we have the first part of the claim.

For the second part, notice that (1) by the subclaim, $p$ abandons at most $2N$ intervals as unsafe, and so $m \leq 2N$, and (2) $I'$ is the interval that $p$ tests for safety after abandoning $I_m$ as unsafe. Furthermore, by the algorithm, each interval in $I, I_1, I_2, \ldots, I_m, I'$ is of size $\Delta$ and begins immediately after the previous one ends. Since $M = (2N + 2)\Delta$ and since we perform the arithmetic modulo $M$, it follows that all of $I, I_1, I_2, \ldots, I_m, I'$ are disjoint intervals. Hence, we have the second part of the claim.

Next we prove the above subclaim. By the algorithm, the testing of $I_1$ for safety begins only after $p$ writes the first sequence number from $I$ in the variable $X$. Let $\tau \in [t, t')$ be the time when this writing happens. For a contradiction, suppose that the subclaim is false and $t'$ is the earliest time when the subclaim is violated. More precisely, let $t'$ be the earliest time in $[t, t')$ such that, for some $q \in \{0, 1, \ldots, N-1\}$, $p$ abandons three intervals as unsafe on the basis of the values that it read in $\mathbb{A}[q]$. Let $I_j, I_k$, and $I_l$ denote these intervals, and let $s_j \in I_j, s_k \in I_k$, and $s_l \in I_l$ be the sequence numbers that $p$ read in $\mathbb{A}[q]$ which caused $p$ to abandon the three intervals. We make a number of observations:

(O1). In the time interval $[t, t')$, $p$ abandons at most $2N + 1$ intervals as unsafe.

Proof: This observation is immediate from the definition of $t'$.

(O2). In the time interval $[t, t')$, $p$ calls select at most $(2N + 1)N$ times.

Proof: This observation follows from Observation O1 and an earlier observation that, in the worst case, $p$ invokes select $N$ times before abandoning any interval as unsafe.

(O3). In the time interval $[t, t')$, all of $p$’s calls to select return distinct sequence numbers from $I$.

Proof: Notice that after $p$ adopts $I$ as its current safe interval at time $t$, $p$’s calls to select return successive sequence numbers starting from the first number in $I$. Since, by Observation O2, $p$ makes at most $\Delta = (2N + 1)N$ calls to select during $[t, t')$, all numbers returned by these calls are distinct numbers from $I$.

(O4). In the time interval $[t, t')$, if $X$ contains a value of the form $[s, p, *]$, then $s \in I$.

Proof: By definition of $t$, $p$ writes in $X$ at time $t$ a value of the form $[s, p, *]$, where $s \in I$. By Observation O3, all values that $p$ subsequently writes in $X$ during $[t, t')$ are from $I$. Hence, we have the observation.

(O5). The intervals $I, I_1, I_2$ and $I_l$ are all disjoint (and, therefore, $s_j, s_k$ and $s_l$ are distinct and are not in $I$).

Proof: Recall that $I_1, I_2, \ldots, I_l$ are the intervals that $p$ abandons during $[t, t')$ as unsafe. By Observation O1, $l \leq 2N + 1$. Furthermore, by the algorithm, each interval in $I, I_1, I_2, \ldots, I_l$ is of size $\Delta$ and begins immediately after the previous one ends. Since $M = (2N + 2)\Delta$ and since we perform the arithmetic modulo $M$, it follows that all of $I, I_1, I_2, \ldots, I_l$ are disjoint intervals. Then, the observation follows from the fact that $I, I_1, I_2, \ldots, I_l$ are members of $\{I, I_1, I_2, \ldots, I_l\}$.

(O6). Recall that $p$ abandons the interval $I_l$ at time $t'$ because it reads at $t'$ the value $[s_l, p, *]$ in $\mathbb{A}[q]$, where $s_l \in I_l$. Let $\sigma'$ be the latest time before $t'$ when $q$ writes $[s_l, p, *]$ in $\mathbb{A}[q]$ (at Line 2). By the algorithm, this writing must be preceded by $q$’s reading of the value $[s_l, p, *]$ from the variable $X$ (Line 1). Let $\sigma$ be the latest time before $\sigma'$ when $q$ reads $[s_l, p, *]$ from $X$. Then, we claim that $\tau < \sigma < \sigma'$. 

Proof: By definition of $s_j, s_k$ and $s_l$, we know that $p$ reads from $\mathbb{A}[q]$ the values $[s_j, p, *], [s_k, p, *]$ and $[s_l, p, *]$ (in that order) in the time interval $[t, t')$. It follows that $q$’s writing of $[s_k, p, *]$ and $[s_l, p, *]$ in $\mathbb{A}[q]$ occur (in that
In our algorithm, the operation \( \triangleleft_M \) is performed modulo \( M = (2N + 2)\Delta \). Hence, the space of all sequence numbers must be at least \( M \). Since we store a sequence number, a process id, and a 1-bit value in the same memory word \( \chi \), the number of bits we have available for a sequence number is \( 63 - \log_2 N \). Hence, \( M \) can be at most \( 2^{63 - \log_2 N} = 2^{63}/N \). Since \( M = (2N + 2)\Delta \) and \( \Delta = (2N + 1)N \), the above constraint translates into \( (2N + 2)(2N + 1)N^2 \leq 2^{63} \). It is easy to verify that for \( N = 2^{15} = 32,768 \) this inequality holds. Our algorithm is therefore correct if the number of processes that execute it is less than 32,768. We believe that this restriction is not of practical concern. Furthermore, our second selection algorithm in Section 6.4 reduces this restriction to \( N = 2^{19} = 524,288 \), at the expense of performing one additional CAS per select operation.

### 6.4 An alternative selection algorithm

In this section, we present an algorithm that supports a larger number of processes than our previous selection algorithm. More specifically, the new algorithm can handle a maximum of \( 2^{19} = 524,288 \) processes, whereas the previous algorithm works for a maximum of \( 2^{15} = 32,768 \) processes.

The main idea of the algorithm is the same as in the first algorithm: at all times, each process \( p \) maintains a current safe interval of size \( \Delta \). Each call to select by \( p \) returns a sequence number from \( p \)'s current safe interval. By the time all numbers in \( p \)'s current safe interval are returned (which won't happen until \( p \) calls select \( \Delta \) times), \( p \) determines a new safe interval of size \( \Delta \) and makes that interval its current safe interval.

The way the next safe interval is located is different from the first algorithm. In the first algorithm, process \( p \) searched...
Types
sequenctype = (63 - log N)-bit number
intervaltype = record start, end: sequenctype end
Local persistent variables at each p ∈ {0, ..., N - 1}
I_p: intervaltype;
val_p: sequenctype
passNum_p: 0 .. lg (N + 1);
procNum_p: 0 .. N - 1
Constants
\[ \Delta = N \log (N + 1) + 1 \]
\[ M = (N + 2) \Delta \]
Initialization
passNum_p = 0;
val_p = 0
procNum_p = 0;
I_p = [\Delta, (N + 2) \Delta]
procedure select(p) returns sequenctype
10: if (passNum_p = 0)
11: a = \lambda[procNum_p]
12: if (a.pid = p) \land (a.seq = A[procNum_p], a, [a.seq, a.pid, 1])
13: if (procNum_p < N - 1)
14: procNum_p++
15: else procNum_p = 0
16: passNum_p++
17: val_p = val_p \oplus_M 1
18: else a = \lambda[procNum_p]
19: if ((a.pid = p) \land (a.val = 1) \land (a.seq \in I_p))
20: Increase the counter of the half of I_p that contains a.seq;
21: if (procNum_p < N - 1)
22: procNum_p++
23: val_p = val_p \oplus_M 1
24: else Set I_p to be the half of I_p with a smaller counter; Reset counters;
25: procNum_p = 0
26: if (passNum_p < \log (N + 1))
27: val_p = val_p \oplus_M 1
28: passNum_p++
29: else passNum_p = 0
30: val_p = I_p.start
31: I_p = [I_p.end, I_p.end \oplus_M (N + 1) \Delta]
32: return val_p

Figure 9: Another selection algorithm

for the next safe interval in a linear fashion: first, p tested whether the interval right next to the current safe interval was safe; if there was evidence that this interval was not safe, p selected the next \( \Delta \)-sized interval to test for safety, and repeated this process until a safe interval was found. In the new algorithm, process p employs a more efficient strategy for locating the next safe interval, based on binary search.

Our algorithm consists of two stages—the marking stage, and the search stage. During the marking stage, process p reads each entry a in the array \( \lambda \). If \( a = [s, p, \ast] \), then it is possible that some process read \( [s, p, \ast] \) at Lines 1 and 3 of its latest BitPid.LL operation, thereby making s potentially unsafe for p. In that case, p puts a mark on a to indicate that it contains a sequence number that is potentially unsafe for it. If, on the other hand, \( a \neq [s, p, \ast] \), then p leaves a unchanged.

After the marking stage completes, p initializes \( I_p \) to some large interval of size \((N + 1)\Delta\), and begins the search stage. The search stage consists of many iterations or passes, each of which takes place over many invocations of select. In each pass, the interval \( I_p \) is halved. Ultimately, after all the passes, \( I_p \) is reduced to a size of \( \Delta \). At that point, p regards the interval \( I_p \) safe, and starts using it as its current safe interval. Below we explain this stage in more detail.

Let \( C = [k, k + \Delta] \) be p’s current safe interval, and let \( I_p = [k + \Delta, k + (N + 2) \Delta] \) be the interval immediately after \( C \). The search stage consists of \( \log (N + 1) \) passes, each of which takes place over \( p \text{’s} \) \( N \) consecutive invocations of select. Within each pass, p performs the following two steps:

- Counting phase: p goes through all the marked entries in the array \( \lambda \), and counts how many sequence numbers fall within the first half, and how many fall within the second half of \( I_p \).
- Halving step: p discards the half of \( I_p \) with a higher count, and sets \( I_p \) to be the remaining half.

Without loss of generality, we assume that \((N+1)\) is a power of two. Then, since after each pass the size of \( I_p \) halves, at the end of all \( \log (N + 1) \) passes the size of \( I_p \) becomes \( \Delta \). Further, p regards this interval safe, and starts using it as its current safe interval.

We now intuitively explain why the above method yields a safe interval. First, observe that the number of marked entries in \( \lambda \) that contain a sequence number from \( I_p \) halves after each pass (since we discard the half of \( I_p \) with a higher count). Next, observe that initially there are at most \( N \) marked entries (since the size of \( \lambda \) is \( N \)). By the above two observations, it follows that after \( \log (N + 1) \) passes, no marked entry in \( \lambda \) contains a sequence number in \( I_p \). Hence, at the end of \( \log (N + 1) \) passes, \( I_p \) is indeed safe for p.

In the following, we explain how the above high level ideas are implemented in our algorithm and why these ideas work.

6.4.1 How the algorithm works

We present our selection algorithm in Figure 9. The algorithm uses four persistent local variables, described as follows:
• $I_p$ is the interval which $p$ halves in the search stage of the algorithm.

• $val_p$ is a sequence number from $p$’s current safe interval which was returned by $p$’s most recent invocation of $\text{select}$.

• $passNum_p$ represents the current pass of process $p$’s algorithm. If we consider the marking stage to be a pass zero, then the $passNum_p$ variable takes values from the range $[0 \ldots \lg(N + 1)]$.

• $procNum_p$ indicates how far process $p$’s reading of the entries in $\Lambda$ has progressed. Specifically, if $procNum_p = k$, it means that the array entries belonging to processes $0, 1, \ldots, k - 1$ (namely, $\Lambda[0], \Lambda[1], \ldots, \Lambda[k - 1]$) have so far been read.

The algorithm works as follows. First, $p$ reads the variable $passNum_p$ to determine which pass of the algorithm it is currently executing (Line 10). If the value of $passNum_p$ is zero, it means that $p$ is still in the marking stage. So, $p$ reads the next element $a$ of the array $\Lambda$ (Line 11). If the process id in $a$ is $p$, $p$ puts a mark on the entry in $\Lambda$ it just read (Line 12). Otherwise, it leaves the entry unchanged. Next, $p$ checks whether it has gone through all the entries in $\Lambda$ (i.e., whether it has reached the end of the marking stage), by reading $procNum_p$ (Line 13).

If not, $p$ simply increments $procNum_p$ (Line 14), and returns the next value from the current safe interval (Lines 17 and 32). Otherwise, $p$ has reached the end of the marking stage, and so it resets $procNum_p$ to zero (Line 15), and increments the $passNum_p$ variable (Line 16). Finally, $p$ returns the next value from the current safe interval (Lines 17 and 32).

On the other hand, if the value of $passNum_p$ is not zero, it means that $p$ is in the search stage. So, $p$ reads the next element $a$ of the array $\Lambda$ (Line 18). If the process id in $a$ is $p$ and $a$ has the mark, then the sequence number in $a$ is potentially unsafe for $p$ and hence should be counted (Line 19).

So, $p$ tests whether the sequence number in $a$ belongs to the first or the second half of $I_p$, and increments the appropriate counter (Line 20). Next, $p$ checks whether it has counted all the entries in $\Lambda$ (i.e., whether it has reached the end of the counting phase), by reading $procNum_p$ (Line 21). If not, $p$ simply increments $procNum_p$ (Line 22), and returns the next value from the current safe interval (Lines 23 and 32). On the other hand, if $p$ has counted all the entries in $\Lambda$, it has all the information it needs to halve the interval $I_p$ appropriately (i.e., to perform the halving step). To this effect, $p$ discards the half of $I_p$ with a higher count, and resets the counters (Line 24). Since it has reached the end of a pass, $p$ also resets $procNum_p$ to zero (Line 25). Next, $p$ reads $passNum_p$ to determine whether it has performed all of the $(N + 1)$ passes (Line 26). If it hasn’t, $p$ simply increments $passNum_p$ and returns the next value from the current safe interval (Lines 27, 28, and 32).

On the other hand, if $p$ has reached the end of all the passes, it means that the interval $I_p$ contains $p$’s next safe interval. So, $p$ selects $I_p$ as its current safe interval (Line 30) and resets variables $passNum_p$ and $I_p$ to begin searching for the next safe interval (Lines 29 and 31). Finally, $p$ returns the next (i.e., the first) value from its new current safe interval (Line 32).

Notice that, similar to our first selection algorithm, the correctness of the above algorithm depends on the following two claims:

• After a process $p$ adopts an interval $I$ to be its current safe interval, $p$ makes at most $\Delta$ calls to $\text{select}$ before adopting a new interval $I’$ as its current safe interval.

• At the time that $p$ adopts $I’$ to be its current safe interval, $I’$ is of size $\Delta$ and is indeed safe for $p$.

We justify the above two claims in the next two subsections.

### 6.4.2 A new interval is identified quickly

Suppose that at time $t$ a process $p$ adopts an interval $I$ to be its current safe interval. Let $t’ > t$ be the earliest time when $p$ adopts a new interval $I’$ as its current safe interval. Then, we claim:

Claim C: During the time interval $[t, t’)$, process $p$ makes at most $\Delta$ calls to $\text{select}$ (which return distinct sequence numbers from the interval $I$). Furthermore, $I$ and $I’$ are disjoint.

The first part of the claim trivially holds since (1) during $[t, t’)$, $p$ executes exactly $\lg(N + 1) + 1$ passes, and (2) in each pass $p$ makes exactly $N$ calls to $\text{select}$. Hence, during the time interval $[t, t’)$, process $p$ makes exactly $N(\lg(N + 1) + 1) = \Delta$ calls to $\text{select}$.

For the second part, notice that soon after $t$, $p$ initializes $I_p$ to be the interval $I’''$, where $I’''$ is the interval of size $(N + 1)\Delta$ immediately after $I$. Furthermore, notice that $I’’$ is the subinterval of $I’''$. Since $M = (N + 2)\Delta$ and since we perform the arithmetic modulo $M$, it follows that the intervals $I$ and $I’’$ are disjoint, and so the intervals $I$ and $I’$ are disjoint as well. Hence, we have the second part of the claim.

### 6.4.3 The new interval is safe

In this section, we argue that (1) the interval $I’$ is of size $\Delta$, and (2) the rule by which the algorithm determines the safety of an interval works correctly. More precisely, let $t$ be the time when process $p$ adopts an interval $I$ to be its current safe interval, and $t’$ be the earliest time after $t$ when $p$ switches its current safe interval from $I$ to a new interval $I’$. Then, we claim:

Claim D: The interval $I’$ is of size $\Delta$, and is safe for process $p$ at time $t’$.

To prove this claim, we make a crucial subclaim which states that, at time $t’$, there are no marked entries in $\Lambda$ holding a sequence number from the interval $I’$.

Subclaim The interval $I’$ is of size $\Delta$, and at time $t’$, no entry in $\Lambda$ is of the form $[s, p, 1]$, where $s \in I’$.

We first argue that the above subclaim implies the claim. Suppose that the claim is false and $I’$ is not safe for $p$ at time $t’$. Let $s’, q’, r’, \eta$ and $\eta’$ be as defined in the proof of Claim B in Section 6.3.3. Then, if we use the same argument
we used in the proof of Claim B, we conclude that (1) \( q \) does not write into \( \mathcal{A}[q] \) during the time interval \([\tau, \tau']\), and (2) the latest value that \( q \) writes into \( \mathcal{A}[q] \) prior to time \( \tau \) is \( \{s', p, \ast\} \). Furthermore, as long as the value in \( \mathcal{A}[q] \) stays of the form \([\ast, p, \ast]\), no other process will attempt to put their mark on \( \mathcal{A}[q] \). By the above observations, we conclude that at some time \( \tau' \in [\tau, \tau'] \) during its 0th pass, \( p \) succeeds in putting its mark on \( \mathcal{A}[q] \). Hence, at all times during \([\tau', \tau']\), \( \mathcal{A}[q] \) holds the value \([s', p, 1]\). In particular, at time \( \tau' \), \( \mathcal{A}[q] \) holds the value \([s', p, 1]\), which contradicts the subclaim. Hence, the claim holds.

Next we prove the above subclaim. Let \( t'' \in [\tau, \tau'] \) be the time when \( p \) completes its 0th pass. For all \( k \in [0, 1, \ldots, \lg (N + 1)] \), let

- \( I_k \) denote the value of the interval \( I_p \) at the end of \( p \)'s \( k \)th pass, and
- \( s_k \) denote the number of marked entries in \( \mathcal{A} \) that, at the end of the \( k \)th pass, hold a sequence number from \( I_k \).

(01). The size of the interval \( I_k \) is \((N + 1)/2^k \)\( \Delta \).

**Proof (by induction):** For the base case (i.e., \( k = 0 \)), the observation trivially holds, since at the beginning of the 0th pass \( I_p \) is initialized to be of size \((N + 1)\Delta \). Hence, the size of \( I_0 \) is \((N + 1)\Delta \). The inductive hypothesis states that the size of \( I_j \) is \((N + 1)/2^j \)\( \Delta \), for all \( j \leq k \). We now show that the size of \( I_{k+1} \) is \((N + 1)/2^{k+1} \)\( \Delta \). By the algorithm, \( I_{k+1} \) is a half of \( I_k \). Moreover, we made an assumption earlier that \( N + 1 \) is a power of two. Hence, the size of \( I_{k+1} \) is exactly \(((N + 1)/2^k)/2) \Delta = ((N + 1)/2^{k+1}) \Delta \).

(02). If an entry \( \mathcal{A}[q] \) holds the value \([s, p, 1]\) at time \( \tau \in [t'', \tau] \), then \( \mathcal{A}[q] \) holds the value \([s, p, 1]\) at all times during \([t'', \tau] \).

**Proof:** Suppose not. Then, at some time \( \tau'' \in [t'', \tau] \), \( \mathcal{A}[q] \) does not hold the value \([s, p, 1]\). Therefore, at some point during \([\tau', \tau] \), the value \([s, p, 1]\) is written into \( \mathcal{A}[q] \). Since by the time \( \tau' \), process \( p \) has already done marking all the entries in \( \mathcal{A} \), some process other than \( p \) must have written \([s, p, 1]\) into \( \mathcal{A}[q] \), which is impossible. Hence, the observation holds.

(03). The value of \( s_k \) is at most \((N + 1)/2^k - 1\).

**Proof (by induction):** For the base case (i.e., \( k = 0 \)), the observation trivially holds, since \( \mathcal{A} \) can hold at most \( N = (N + 1) - 1 \) entries. Hence, the value of \( s_0 \) is at most \( N \). The inductive hypothesis states that the value of \( s_j \) is at most \((N + 1)/2^j - 1\), for all \( j \leq k \). We now show that the value of \( s_{k+1} \) is at most \((N + 1)/2^{k+1} - 1\). Since \( s_k \) is the number of entries in \( \mathcal{A} \) that, at the end of the \( k \)th pass, hold a sequence number from \( I_k \), it follows by Observation 02 that \( p \) can count at most \( s_k \) sequence numbers during the \((k + 1)\)st pass. Moreover, since \( I_{k+1} \) is a half of \( I_k \) with a smaller count, it follows that at most \([s_k/2]\) of the counted sequence numbers fall within \( I_{k+1} \). Hence, by Observation 02, at the end of the \((k + 1)\)st pass, at most \([s_k/2]\) marked entries in \( \mathcal{A} \) hold a sequence number from \( I_{k+1} \). Therefore, we have 
\[ s_{k+1} = [s_k/2] = [(N + 1)/2^{k+1} - 1]. \]
Since \( N + 1 \) is a power of two, it means that 
\[ s_{k+1} = (N + 1)/2^{k+1} - 1. \]
Hence, the observation holds.

By the above observations, at the end of the \( \lg (N + 1) \)st pass, we know that (1) the size of \( I_p \) is \( \Delta \) (by Observation 01), and (2) the number of marked entries in \( \mathcal{A} \) that hold a sequence number from \( I' \) is zero (by Observation 03). Hence, we have the subclaim.

Based on the above discussion, we have the following lemma. Its proof is given in the Appendix F.

**Lemma 2** The implementation of select in Figure 9, satisfies Property 1. The time complexity of the implementation is \( 2 \), and the per-process space overhead is zero.

### 6.4.4 A remark about sequence numbers

In our algorithm, the operation \( \oplus_M \) is performed modulo \( M = (N + 2)\Delta \). Hence, the space of all sequence numbers must be at least \( M \). Since we store a sequence number, a process id, and a 1-bit value in the same memory word \( X \), the number of bits we have available for a sequence number is \( 63 - \lg N \). Hence, \( M \) can be at most \( 2^{63 - \lg N} = 2^{63}/N \). Since \( M = (N + 2)\Delta \) and \( \Delta = N(\lg (N + 1) + 1) \), the above constraint translates into \((N + 2)N^2(\lg (N + 1) + 1) \leq 2^{63} \). It is easy to verify that for \( N \leq 2^{19} = 524,288 \), this inequality holds. Our algorithm is therefore correct if the number of processes that execute it is less than 524,288. This limit is large enough that it is not of any practical concern.

### 7 Conclusions and Future Work

We have shown two implementations of a 64-bit LL/SC object from 64-bit CAS/read or RLL/RSC objects. Both implementations have constant time complexity, and use only a constant amount of space per process. The first algorithm is efficient and practical, but uses unbounded sequence numbers. The second algorithm is more complex, but overcomes this limitation.

Although the space requirements of our implementations are negligible when a single LL/SC variable is implemented, our present algorithms do not extend well when the number of LL/SC variables to be supported is large. In particular, in order to implement \( M \) LL/SC variables using CAS or RLL/RSC instructions, our algorithms require \( O(M) \) space per process (which amounts to \( O(NM) \) space among all \( N \) processes). Future research will explore the possibility of implementing \( M \) LL/SC variables using \( O(N + M) \) space among all \( N \) processes.
References


A Proof of the algorithm in Figure 3

In the following, let SC_{p,i} denote the i-th successful SC by process p and v_{p,i} denote the value written in O by SC_{p,i}. The operations are linearized according to the following rules. We linearize each SC operation at Line 8 and each VL at Line 14.
Let $OP$ be any execution of the LL operation by $p$. Let $[q, k]$ be the value that $p$ reads in Line 1 of $OP$. The linearization point of $OP$ is determined by two cases. If $OP$ returns $v_{q,k}$, then we linearize $OP$ at Line 1. Otherwise, we show that, for some $i > k$, $SC_{q,i}$ takes effect during $OP$ and $OP$ returns $v_{q,i}$. In this case, we linearize $OP$ just after $SC_{q,i}$ takes effect.

We begin by making the following observation.

**Observation 1** The value of $seq_p$ is increased by 1 in every successful SC operation by $p$ (Line 11). Unsuccessful SC’s by $p$ do not change the value of $seq_p$.

**Claim 1** At the beginning of $SC_{p,i}$, $seq_p$ holds the value $i$.

**Proof.** Prior to $SC_{p,i}$, exactly $i - 1$ successful SC operations are performed by $p$. According to Observation 1, variable $seq_p$ was therefore incremented exactly $i - 1$ times prior to $SC_{p,i}$. Since $seq_p$ is initialized to 1, it follows that at the beginning of $SC_{p,i}$, $seq_p$ holds the value $i$.\[\square\]

**Claim 2** From the moment $p$ performs Line 7 of $SC_{p,i}$, until the time $p$ completes the $SC_{p,i+1}$ operation, $val_p[i \mod 2]$ holds the value $v_{p,i}$.

**Proof.** According to Claim 1, at the beginning of $SC_{p,i}$, $seq_p$ holds the value $i$. Therefore, $p$ writes $v_{p,i}$ into $val_p[i \mod 2]$ in Line 7 of $SC_{p,i}$. Since $p$ increments $seq_p$ in Line 11 of $SC_{p,i}$, $v_{p,i}$ will not be overwritten in $val_p[i \mod 2]$ until $seq_p$ reaches the value $i + 2$. By Observation 1, $seq_p$ will not reach $i + 2$ until $p$ executes Line 11 of $SC_{p,i+1}$. Therefore, variable $val_p[i \mod 2]$ holds the value $v_{p,i}$ from the moment $p$ preforms Line 7 of $SC_{p,i}$ until the time $p$ completes $SC_{p,i+1}$.\[\square\]

**Claim 3** The values that process $p$ writes into $oldval_p$ and $oldseq_p$ in Lines 9 and 10 of $SC_{p,i}$ are $v_{p,i-1}$ and $i - 1$, respectively.

**Proof.** According to Claim 1, at the beginning of $SC_{p,i}$, $seq_p$ holds the value $i$. Therefore, $p$ writes $i - 1$ into $oldseq_p$ in Line 10 of $SC_{p,i}$. By Claim 2, $val_p[(i - 1) \mod 2]$ holds the value $v_{p,i-1}$ at all times during $SC_{p,i}$. As a result, $p$ writes $v_{p,i-1}$ into $oldval_p$ in Line 9 of $SC_{p,i}$.\[\square\]

**Claim 4** Let $OP$ be an LL operation by process $p$, and $[q, k]$ the value that $p$ reads in Line 1 of $OP$. If $OP$ terminates in Line 4, then it returns the value $v_{q,k}$.

**Proof.** Let $T$ be the time interval starting from the moment $q$ performs Line 7 of $SC_{q,k}$ until $q$ completes $SC_{q,k+1}$. According to Claim 2, variable $val_q[k \mod 2]$ holds the value $v_{q,k}$ at all times during $T$. Our goal is to show that $p$ executes Line 2 of $OP$ during $T$, and therefore reads $v_{q,k}$ from $val_q[k \mod 2]$ in Line 2. At the moment $p$ reads $[q, k]$ from $X$ in Line 1 of $OP$, $q$ must have already executed Line 7 of $SC_{q,k}$, and not yet executed Line 8 of $SC_{q,k+1}$. Hence, $p$ executes Line 1 during $T$. From the fact that $OP$ terminates in Line 4, it follows that $p$ satisfied the condition in Line 4. Therefore, the value that $p$ reads in Line 3 of $OP$ is either $k - 2$ or $k - 1$. Hence, by Claim 3, it follows that when $p$ performs Line 3, $q$ did not yet complete $SC_{q,k+1}$. So, $p$ executes Line 3 during $T$. Since $p$ performed both Lines 1 and 3 during $T$, it follows that $p$ performs Line 2 during $T$ as well. Therefore, $p$ reads $v_{q,k}$ from $val_q[k \mod 2]$ in Line 2, and the value that $OP$ returns is $v_{q,k}$.\[\square\]

**Lemma 3 (Correctness of LL)** Let $OP$ be any LL operation and $OP'$ be the latest successful SC operation that precedes OP by the linearization order defined earlier. Then, $OP$ returns the value written by $OP'$.

**Proof.** Let $p$ be the process executing $OP$. Let $[q, k]$ be the value that $p$ reads in Line 1 of $OP$. We examine the following two cases: (i) $OP$ returned $v_{q,k}$, and (ii) $OP$ returned $v' \neq v_{q,k}$.

In the first case, by our linearization, $OP$ is linearized at Line 1. From the fact that $p$ reads $[q, k]$ in Line 1 of $OP$, it is clear that $SC_{q,k}$ is the latest SC operation to perform Line 8 prior to $LP$. Since we linearize all SC operations at Line 8, it follows that $OP' = SC_{q,k}$, and therefore the lemma is trivially true. In the second case, by Claim 4, it follows that $OP$ didn’t terminate in Line 4. Hence, $OP$ must have terminated in Line 6, returning $v'$. Our goal is to show that there exists $i > k$, such that (1) $SC_{q,i}$ executes Line 8 during $OP$, and (2) $v' = v_{q,i}$. Then, by our linearization, $OP' = SC_{q,i}$, which implies the lemma.

If $OP$ terminated in Line 6, then the condition in Line 4 of $OP$ didn’t hold. Then, $p$ reads a value $i \geq k$ in Line 3 of $OP$. Hence, by Claim 3, $q$ completes Line 10 of $SC_{q,k+1}$ before $p$ performs Line 3 of $OP$. Consequently, $q$ completes Line 9 of $SC_{q,k+1}$ before $p$ performed Line 5 of $OP$. As a result, the value $v'$ that $p$ reads in Line 5 was written by $q$ in either $SC_{q,k+1}$ or a later SC operation by $q$. Since $v' \neq v_{q,k}$, $v'$ was written by $SC_{q,i+1}$, for some $i > k$. Then, by Claim 3, $v' = v_{q,i}$.

We now show that $q$ executes Line 8 of $SC_{q,i}$ during $OP$. At the moment that $p$ reads $[q, k]$ in Line 1 of $OP$, $q$ did not yet execute Line 8 of $SC_{q,i}$ (since $i > k$). Also, by the argument above, at the moment when $p$ reads $v' = v_{q,i}$ in Line 5 of $OP$, $q$ already executed Line 9 of $SC_{q,i+1}$, and therefore already executed Line 8 of $SC_{q,i}$. Hence, $q$ must have executed Line 8 of $SC_{q,i}$ while $p$ was between Lines 1 and 5 of $OP$, which means that $q$ executed Line 8 of $SC_{q,i}$ during $OP$. As discussed above, this observation implies that $OP$ returns the correct value.\[\square\]

**Claim 5** Let $OP$ be an LL operation by process $p$, and $[q, k]$ the value that $p$ reads from $X$ in Line 1 of $OP$. If $X$ does not change for the remainder of $OP$, then $OP$ terminates in Line 4.

**Proof.** Let $k'$ be the value that $p$ reads in Line 3 of $OP$. At the moment when $p$ reads $[q, k]$ from $X$ in Line 1 of $OP$, $q$ must have already executed Line 8 of $SC_{q,k}$. Hence, $q$ must have already executed Line 10 of $SC_{q,k-1}$. Then, by Claim 3,
it follows that $k' \geq k - 2$. Furthermore, since $X$ does not change for the remainder of $OP$, $q$ does not execute Line 8 of $SC_{q,k+1}$ during $OP$. Consequently, $q$ does not execute Line 10 of $SC_{q,k+1}$, or any later SC operation, during $OP$. Therefore, by Claim 3, $k' \leq k - 1$. Then, by this and the previous observation, we have $k - 2 \leq k' \leq k - 1$. Therefore, $p$ terminates in Line 4.

Lemma 4 (Correctness of SC) Let $OP$ be any SC operation by process $p$, and $OP'$ be the latest LL operation by $p$ that precedes $OP$. Then, $OP$ succeeds if and only if, in the linearization order, no successful SC occurs between $OP'$ and $OP$.

Proof. Let $[q, k]$ be the value that $p$ reads in Line 1 of $OP$. If $OP$ returned false, then clearly the CAS in Line 8 of $OP$ failed. Hence, the value in $X$ was different than $[q, k]$. We examine the following two cases: (i) $OP'$ returned $v_{q,k}$, and (ii) $OP'$ returned a value different than $v_{q,k}$. In the first case, by our linearization, $OP'$ is linearized at Line 1. From the fact that $X$ was different than $[q, k]$ in Line 8 of $OP$, it follows that some successful SC operation $OP''$ executed Line 8 between the time $p$ executed Line 1 of $OP'$ and the time $p$ executed Line 8 of $OP$. Since all SC operations are linearized at Line 8, it follows that $OP''$ is linearized between $OP'$ and $OP$, and $OP$ is therefore correct to return false. In the second case, by Claim 4, and the proof of Lemma 3, we have the following:

- $OP'$ terminates in Line 6,
- the value $v'$ that $p$ reads in Line 5 of $OP'$ was written by $q$ in Line 9 of $SC_{q,i+1}$, for some $i > k$,
- $v' = v_{q,i}$,
- $SC_{q,i}$ executes Line 8 during $OP'$.

Furthermore, by our linearization, $OP'$ is linearized just after the LP of $SC_{q,i}$. Since $q$ executes Line 9 of $SC_{q,i+1}$ before $p$ executes Line 5 of $OP'$, it follows that $q$ executes Line 8 of $SC_{q,i+1}$ before $p$ executes Line 5 of $OP'$. Moreover, since $SC_{q,i}$ executed Line 8 during $OP'$, and $SC_{q,i+1}$ happened after $SC_{q,i}$, $SC_{q,i+1}$ must have also executed Line 8 during $OP$. Then, by our linearization, $OP'$ is linearized before $SC_{q,i+1}$, which is in turn linearized before $OP$. Hence, $OP$ was correct to return false.

If $OP$ returned true, then clearly the CAS in Line 8 of $OP$ succeeded. Hence, the value in $X$ was equal to $[q, k]$. Then, from the moment $p$ reads $[q, k]$ in Line 1 of $OP'$, until $p$ executes Line 8 of $OP$, $X$ does not change. Hence, by Claims 5 and 4, $OP'$ returned $v_{q,k}$. Then, by our linearization, $OP'$ is linearized at Line 1, and no successful SC is linearized between $OP'$ and $OP$. Hence, $OP$ was correct to return true.

Lemma 5 (Correctness of VL) Let $OP$ be any VL operation by a process $p$, and $OP'$ be the latest LL operation by $p$ that precedes $OP$. Then, $OP$ returns true if and only if, in the linearization order, no successful SC occurs between $OP'$ and $OP$.

Proof. Similar to the proof of Lemma 4.

Theorem 1 follows from the above three lemmas.

B Proof of the algorithm in Figure 4

The operations are linearized according to the following rules. We linearize each SC operation at Line 11, and each VL at Line 12. Let $OP$ be any execution of the LL operation by $p$. The linearization point of $OP$ is determined by two cases. If either one of $WLL$ calls in Lines 1 or 6 of $OP$ succeeds, we linearize $OP$ to the point at which that $WLL$ happened. Otherwise, we linearize $OP$ as follows. Let $[failure, q]$ be the value returned by $WLL$ call in Line 1 of $OP$. Let $OP'$ be the successful SC operation performed by process $q$ that took effect during the execution of that $WLL$. Let $OP''$ be the latest LL operation performed by process $q$ to update the value of variable lastVal before process $p$ reads it in Line 5 of $OP$. Then, if $OP''$ happened before $OP'$, we linearize $OP$ to the point just before $OP'$ takes effect. Otherwise, we linearize $OP$ to the point just before $OP''$ is linearized, making sure that no successful SC operation is linearized between $OP$ and $OP'$.

We now give a formal specification of the WLL/SC object:

Definition 1 Let $O$ be a WLL/SC object. In every execution history $H$ on object $O$, the following is true:

- each operation $op$ takes effect at some instant $LP(op)$ during its execution interval,
- if an WLL operation $op$ performed by process $p$ returns $[failure, q]$, then there exists a successful SC operation $op'$ performed by process $q$ such that:
  1. $LP(op')$ lies in the execution interval of $op$,
  2. $LP(op) < LP(op')$.
- the responses of all successful WLL operations and all VL and SC operations, when ordered according to their $LP$ times, are consistent with the sequential specifications of LL, SC and VL.

Lemma 6 (Correctness of LL) Let $OP$ be any LL operation and $OP'$ be the latest successful SC operation that precedes $OP$ by the linearization order defined earlier. Then, $OP$ returns the value written by $op'$.

Proof. We examine the following three cases: (i) $OP$ returns in Line 4, (ii) $OP$ returns in Line 9, and (iii) $OP$ returns in Line 10. In the first case, the $WLL$ call in Line 1 of $OP$ succeeds and, by Definition 1, returns the value $v$ written by the latest successful SC on $X$ before Line 1 of $OP$. Since, by our linearization, $OP$ is linearized at Line 1 and all SC operations are linearized at Line 11, $OP$ returns the value written by $op'$. The same argument holds in the second case as well.

In the third case, let $p$ be the process executing $OP$. Since $OP$ returns in Line 10, both $WLL$ calls in $OP$ must have failed. Let $[failure, q]$ be the value returned by the $WLL$ call in Line 1 of $OP$. Then, by Definition 1, there exists a successful SC on $X$ by process $q$ that takes effect at some point during $WLL$. Let $SC_{q,y}$ be the SC operation which made that $SC$ call. Let $LL_{pq}$ be the latest LL by $q$ to update lastVal before $p$ executes Line 5 of $OP$. We examine the following two cases: (1) $LL_{pq}$ was executed before $SC_{q,y}$, and (2) $LL_{pq}$ was executed
after SC
. In the first case, let OP
 be the latest LL by q to precede SC
. Then, we argue that OP
 = LL
. Since the SC
 call in Line 11 of SC
 returned true, one of the WLL calls in OP
 must have been successful. As a result, OP
 updated lastVal
. Furthermore, since OP
 is the latest LL by q that precedes SC
, OP
 is also the latest LL by q to update lastVal
 prior to SC
. On the other hand, since q executes Line 11 of SC
 while p is executing Line 1 of OP
, q executes Line 11 of SC
 before p executed Line 5 of OP
. Since, by supposition, LL
 is executed before SC
, LL
 is therefore the latest LL by q to update lastVal
 prior to SC
. Hence, we have OP
 = LL
. It remains to be shown that if OP
 is the latest successful SC operation that precedes OP by the linearization order, and v is the value that OP
 writes, then OP
 returns v. We make this argument by showing that the value that q reads in the successful WLL call of LL
 is v. Then, since LL
 writes v into lastVal
, and LL
 is the latest LL operation to update lastVal
 prior to Line 5 of OP
, it is clear that OP
 returns v.

According to our linearization, the operations are linearized as follows: SC
 is linearized at Line 11, LL
 is linearized at either Line 1 or Line 6, depending on which of the two WLL operations succeeded, OP
 is linearized at Line 11, and OP
 is linearized just before the LP of SC
. (Notice that the LP of OP
 falls within the execution interval of OP
 because LP of SC
 lies within the execution interval of Line 1 of OP.) Then, by Definition 1, it is clear that the LP of OP
 is not between the LPs of LL
 and SC
 (otherwise, the SC call in Line 11 of SC
 would not have succeeded). Furthermore, since OP
 is the latest successful SC whose LP is before the LP of OP, and since the LP of OP
 is just before the LP of SC
, OP
 is the latest successful SC whose LP is before the LP of SC
 (and based on the previous argument, before the LP of LL
). As a result, q reads v in the successful WLL call of LL
. Based on an earlier argument, OP
 returns v.

Finally, we examine the case where both of the WLL operations of OP fail, and LL
 happens after SC
. We show that if OP
 is the latest successful SC operation that precedes OP
 by the linearization order, and v the value that OP
 writes, then the value that q reads in the successful WLL call of LL
 is v. Then, since LL
 writes v into lastVal
, and LL
 is the latest LL operation to update lastVal
 prior to Line 5 of OP
, it follows that OP
 returns v.

According to our linearization, the operations are linearized as follows: SC
 is linearized at Line 11, LL
 is linearized at either Line 1 or Line 6, depending on which of these two WLL operations succeeded, OP
 is linearized at Line 11, and OP
 is linearized just before the LP of LL
, with no successful SC operations linearized between the LP of OP
 and the LP of LL
. Notice that the LP of OP
 indeed falls within its execution interval since (a) The LP of LL
 happens before Line 5 of OP
, and (b) The LP of LL
 happens after SC
, and therefore after Line 1 of OP
. Hence, the LP of LL
 is within OP, which goes for the LP of OP as well. Since OP
 is the latest successful SC whose LP is before the LP of OP, and since no successful SCs are linearized between the LP of OP and the LP of LL
, OP
 is the latest successful SC whose LP is before the LP of LL
. As a result, q reads v in the successful WLL call of LL
. Based on an earlier argument, OP
 returns v.

**Lemma 7 (Correctness of SC)** Let OP be any SC operation by process p, and OP
 be the latest LL operation by p that precedes OP. Then, OP succeeds if and only if, in the linearization order, no successful SC occurs between OP
 and OP.

**Proof.** If OP returned false, then clearly the SC call in Line 11 of OP failed. We examine the following two cases: (i) At least one WLL call during OP
 succeeded, and (ii) both WLL calls during OP
 failed. In the first case, without loss of generality, we assume that the call in Line 1 of OP
 succeeded. Then, by our linearization, OP
 is linearized at Line 1, and OP
 is linearized at Line 11. Furthermore, by Definition 1, there exists a successful SC on x by some process q that takes effect between the LP of OP
 and the LP of OP
 (otherwise, the SC call in Line 11 of OP would have succeeded). Hence, a successful SC operation is linearized between the LP of OP
 and the LP of OP
.

In the second case, let [failure, q] be the value returned by the WLL call in Line 1 of OP
. Then, by Definition 1, there exists a successful SC on x by process q that takes effect at some point during Line 1 of OP
. Let SC
 be the SC operation which made that SC call. Let LL
 be the latest LL by q to update lastVal
 prior to OP
. We examine the following two cases: (1) LL
 was executed before SC
, and (2) LL
 was executed after SC
. In both of these cases, OP
 is linearized at some point before Line 5. In the first case, it is linearized at some point during the execution of Line 1, and in the second case just prior to a successful WLL call of LL
, which in turn happened before Line 5 of OP
. Since the WLL call in Line 6 of OP
 failed, there exists a successful SC on x that takes effect during that WLL. The SC operation which made that call is therefore linearized between the LP of OP
 and the LP of OP.

If OP returned true, then clearly the SC call in Line 11 of OP succeeded. That means that at least one of the WLL calls in OP
 succeeded. By our linearization, OP
 is linearized at the point at which that successful WLL happened, and OP in Line 11. Furthermore, by Definition 1, no successful SC calls on x happened between the LP of OP
 and the LP of OP. Hence, no successful SC operation is linearized between the LP of OP
 and the LP of OP.

**Lemma 8 (Correctness of VL)** Let OP be any VL operation by a process p, and OP
 be the latest LL operation by p that precedes OP. Then, OP returns true if and only if, in the linearization order, no successful SC occurs between OP
 and OP.

**Proof.** Similar to the proof of Lemma 7.

Theorem 2 follows from the above three lemmas.

**C Proof of the algorithm in Figure 5**

In the following, let SC
 denote the i-th successful SC by process p and v
 denote the value written in O by SC
. The operations are linearized according to the following rules. We linearize each SC operation at Line 7, each VL at Line 10, and each WLL at Line 1.
Claim 6 Let SC_{p,i} be a successful SC operation by process p. If i is odd, then p changes the value of index_p from 1 to 0 in Line 8 of SC_{p,i}. If i is even, then p changes the value of index_p from 0 to 1 in Line 8 of SC_{p,i}. Unsuccessful SC’s by p do not change the value of index_p.

Proof. From the algorithm design, it is clear that unsuccessful SC’s by p do not change the value of index_p. We now prove the first part of the claim, by induction. For the base case, we verify that the claim holds for SC_{p,1}. Since the index_p variable is initialized to 1, and since p changes index_p to 1 = 0 in Line 1 of SC_{p,1}, the claim trivially holds for the base case. The induction hypothesis is that the claim holds for some SC_{p,n}. The induction step is to show that the claim holds for SC_{p,n+1} as well. Suppose that n + 1 is odd (which means that n is even). Then, by IH, p changes the value of index_p from 0 to 1 in Line 8 of SC_{p,n}. Therefore, at the beginning of SC_{p,n+1}, the value of index_p is 1. Furthermore, p changes index_p to 1 = 0 in Line 8 of SC_{p,n+1}. Hence, the claim holds if n + 1 is odd. Suppose that n + 1 is even (which means that n is odd). Then, by IH, p changes the value of index_p from 1 to 0 in Line 8 of SC_{p,n}. Therefore, at the beginning of SC_{p,n+1}, the value of index_p is 0. Furthermore, p changes index_p to 1 = 0 in Line 8 of SC_{p,n+1}.

Corollary 1 At the beginning of SC_{p,i}, index_p holds the value i mod 2.

Claim 7 From the moment p performs Line 6 of SC_{p,i}, until the time p completes the SC_{p,i+1} operation, variable val_p[i mod 2] holds the value v_p,i.

Proof. According to Corollary 1, at the beginning of SC_{p,i}, index_p holds the value i mod 2. Therefore, p writes v_p,i into val_p[i mod 2] in Line 6 of SC_{p,i}. Since, by Claim 6, p changes index_p in Line 11 of SC_{p,i} to 1 = (i mod 2) \neq i mod 2, v_p,i will not be overwritten in val_p[i mod 2] until index_p reaches the value i mod 2 again. By Claim 6, index_p will not reach the value i mod 2 until p changes index_p to 1 = (1 - (i mod 2)) = i mod 2 in Line 11 of SC_{p,i+1}. Therefore, val_p[i mod 2] holds the value v_p,i from the moment p performs Line 6 of SC_{p,i} until the time p completes SC_{p,i+1}.

Lemma 9 (Correctness of WLL) Let OP be any WLL operation and OP’ the latest successful SC operation that precedes OP by the linearization order defined earlier. If OP returns [success, v], then v is the value written by OP’. If OP returns [failure, q], then there exists a successful SC operation OP” by process q such that: (i) LP(OP") lies in the execution interval of OP, and (ii) LP(OP) < LP(OP’).

Proof. If OP returned [success, v], let q be the process that executes OP’. Since OP’ is successful, OP’ = SC_{q,i}, for some i. Thus, we need to show that v = v_{q,i}.

By our linearization, OP is linearized at Line 1, and OP’ in Line 7. Then, since OP’ is the latest successful SC that precedes OP by the linearization order, the SC call in Line 7 of OP’ is the latest successful SC on X before the BitPid_LL call in Line 1 of OP. Furthermore, by Corollary 1, q writes i mod 2 into X in Line 7 of OP’. Therefore, p reads i mod 2, q in Line 1 of OP. Since the VL call in Line 3 of OP succeeded, no process made a successful SC call between the time p executes Line 1 of OP, and the time p executes Line 3 of OP. More specifically, q did not execute the SC call in Line 7 of SC_{p,i+1} between the times p executes Lines 1 and 3 of SC_{p,i+1}. Therefore, by Claim 7, val_q[i mod 2] holds the value v_{q,i} at all times while p is between Lines 1 and 3 of OP. Consequently, p reads v_{q,i} in Line 2 of OP, and returns the correct value in Line 3 of OP.

If OP returned [failure, q], then clearly the VL call in Line 3 of OP failed. This failure must have happened because some other process performed a successful SC on X between the time p executed Line 1 of OP, and the time p executed Line 3 of OP. Consequently, the value [b, q] that p reads in Line 4 of OP was written to X by some process q while p was between Lines 1 and 4 of OP. Therefore, there exists a successful SC operation OP” by q, such that q made the SC call in Line 7 of OP” while p was between Lines 1 and 4 of OP. Since, by our linearization, OP is linearized at Line 1 and OP” is linearized at Line 7, if follows that (1) LP(OP) lies in the execution interval of OP, and (2) LP(OP) < LP(OP”).

Lemma 10 (Correctness of SC) Let OP be any SC operation by process p, and OP’ be the latest LL operation by p that precedes OP. Then, OP succeeds if and only if, in the linearization order, no successful SC occurs between OP’ and OP.

Proof. If OP returned false, then clearly the SC call in Line 7 of OP failed. We examine the following two cases: (i) OP’ is successful, and (ii) OP’ has failed. Observe that in both of the cases there exists a successful SC on X which happened between the time p executed Line 1 of OP’ and the time p executed Line 7 of OP. Since, by our linearization, OP’ is linearized at Line 1 and OP is linearized at Line 7, there exists a successful SC operation that is linearized between the LP of OP’ and the LP of OP.

If OP returned true, then clearly the SC call in Line 7 of OP succeeded. Therefore, no successful SC operation on X happened between the time p executed Line 1 of OP’, and the time p executed Line 7 of OP. Since, by our linearization, OP’ is linearized at Line 1 and OP is linearized at Line 7, no successful SC operations happened between the LP of OP’ and the LP of OP.

Lemma 11 (Correctness of VL) Let OP be any VL operation by a process p, and OP’ be the latest LL operation by p that succeeds OP. Then, OP returns true if and only if, in the linearization order, no successful SC occurs between OP’ and OP.

Proof. Similar to the proof of Lemma 10.

Theorem 3 follows from the above three lemmas.
D Proof of the algorithm in Figure 6

The operations are linearized according to the following rules.
Let OP be any execution of the SC operation by p. The linearization point of OP is determined by two cases. If the condition in Line 5 of OP is true, we linearize OP at any point during Line 5. Otherwise, we linearize OP at Line 6. Let OP' be any execution of the VL operation by p. The linearization point of OP' is determined by two cases. If the first part of the condition in Line 9 (oldp = chk_p) of OP' is true, we linearize OP' at any point during Line 9. Otherwise, we linearize OP' at the point in Line 9 when x is read. Let OP'' be any execution of the LL operation by p. The linearization point of OP'' is determined by two cases. If the values read in Lines 1 and 3 of OP'' are different, we linearize OP'' at Line 1. Otherwise, we linearize OP'' at Line 3.

In the following, we assume that select satisfies Property 1.

Lemma 12 (Correctness of BitPid_LL) Let OP be any BitPid_LL operation and OP' be the latest successful SC operation that precedes OP by the linearization order defined earlier. Then, OP returns the value written by OP', and the process id of the process that executes OP'.

Proof. Let p be the process executing OP, and q the process executing OP'. We examine the following two cases: (i) the values that p reads in Lines 1 and 3 of OP are different, and (ii) the values that p reads in Lines 1 and 3 of OP are the same. In the first case, OP is linearized at Line 1, and OP' is linearized at Line 6 (since OP' is successful). Since OP' is the latest successful SC whose LP is before the LP of OP, we have the following: OP' is the latest SC to write to X before p executes Line 1 of OP. Therefore, OP returns the value written by OP', and the process id q.

In the second case, OP is linearized at Line 3, and OP' is linearized at Line 6 (since OP' is successful). Since OP' is the latest successful SC whose LP is before the LP of OP, we have the following: OP' is the latest SC to write to X before p executes Line 3 of OP. Therefore, p reads the value written by OP' and the process id q in Line 3 of OP. Since the values that p reads in Lines 1 and 3 of OP are the same, OP returns the correct value-pid pair in Line 4 of OP.

Lemma 13 (Correctness of SC) Let OP be any SC operation by process p, and OP' be the latest BitPid_LL operation by p that precedes OP. Then, OP succeeds if and only if, in the linearization order, no successful SC occurs between OP' and OP.

Proof. We examine the following three cases: (i) OP returns false in Line 5, (ii) OP returns false in Line 6, and (iii) OP returns true in Line 8. In the first case, the values that p reads in Lines 1 and 3 of OP' are different. Then, by our linearization, OP' is linearized at Line 1 and OP is linearized at Line 5. Since x changes between the LP of OP' and the LP of OP, there exists an SC operation OP'' that performed a successful CAS in Line 6 between the LP of OP'' and the LP of OP. Furthermore, since OP'' is linearized at Line 6, it follows that the LP of OP'' is between the LP of OP' and the LP of OP.

In the second case, the values that p reads in Lines 1 and 3 of OP' are the same, but the CAS call in Line 6 of OP failed. Then, by our linearization, OP' is linearized at Line 3 and OP is linearized at Line 6. Since x changes between the LP of OP' and the LP of OP, there exists an SC operation OP'' that performed a successful CAS in Line 6 between the LP of OP'' and the LP of OP. Furthermore, since OP'' is linearized at Line 6, it follows that the LP of OP'' is between the LP of OP' and the LP of OP.

In the third case, the values that p reads in Lines 1 and 3 of OP' are the same, and the CAS call in Line 6 of OP succeeds. Then, by our linearization, OP' is linearized at Line 3, and OP is linearized at Line 6. Let [s, q, v] be the value that p reads in Lines 1 and 3 of OP'. Then, q is the latest process to write to X before the LP of OP'. According to Property 1, q did not write [s, q, v] to X between the LP of OP' and the LP of OP. Therefore, since x contains the same value at the LP of OP' as it did at the LP of OP, it follows that no process wrote to X between the LP of OP' and the LP of OP. Consequently, no successful SC operation occurs between the LP of OP' and the LP of OP.

Lemma 14 (Correctness of VL) Let OP be any VL operation by a process p, and OP' be the latest LL operation by p that precedes OP. Then, OP returns true if and only if, in the linearization order, no successful SC occurs between OP' and OP.

Proof. Similar to the proof of Lemma 13.

Theorem 4 follows from the above three lemmas.

E Proof of the algorithm in Figure 7

We show that the implementation of select in Figure 7, satisfies Property 1.

Definition 2 An ‘epoch of p’ is the period of time between the two consecutive executions of Line 17 in select(p), or a period of time between the end of the initialization phase of the algorithm and the first time Line 17 is executed in select(p).

Definition 3 Interval [x, x ⊕ M Δ) is the ‘current interval’ of an epoch if x is the value of the variable nextStart_p at the beginning of that epoch.

Claim 8 Let E be the current epoch of p and t an arbitrary point in time during E. Let E_t be the time interval that spans from the moment E starts until time t. If a condition in Line 11 of select(p) holds true at most 2N times during E_t, then all the sequence numbers that select(p) returns during E_t are unique and belong to the current interval of E.

Proof. We prove this claim in two steps. First, we use the supposition to prove that the total number of sequence numbers returned by select(p) during E_t is at most (2N + 1)N.
Then, we use this fact to prove that all the sequence numbers that \( \text{select}(p) \) returns during \( E_1 \) are unique and belong to the current interval of \( E \).

Let \( t_e \) be the latest time during \( E \) that a sequence number is returned by \( \text{select}(p) \). (If there is no such time, then the claim trivially holds.) Then, since \( t_e \leq t \), it follows by supposition that the condition in Line 11 of \( \text{select}(p) \) was true at most \( 2N \) times prior to \( t_e \). Thus, the value of \( \text{procNum} \) was reset in Line 13 at most \( 2N \) times prior to \( t_e \). Furthermore, since \( t_e \) belongs to \( E \), the value of \( \text{procNum} \) hasn’t yet reached \( N \) at time \( t_e \) (otherwise, the new epoch would have started in Line 17). Since \( \text{procNum} \) has been reset at most \( 2N \) times prior to \( t_e \), the number of times \( \text{procNum} \) was incremented in Line 14 prior to \( t_e \) is at most \((2N + 1)(N - 1)\) (otherwise, \( 2N \) resets wouldn’t be able to prevent \( \text{procNum} \) from reaching the value \( N \)). Therefore, the condition in Line 11 was false at most \((2N + 1)(N - 1)\) times prior to \( t_e \). Hence, the total number of times the condition in Line 11 was tested prior to \( t_e \) is at most \((2N + 1)(N - 1) + 2N = 2N^2 + N - 1\). Then, the total number of sequence numbers returned by \( \text{select}(p) \) during \( E_1 \) is at most \( 2N^2 + N - 1 + 1 = (2N + 1)N \).

Let \( t_1 \) and \( t_2 \) in \( E \) be the times of two successive executions of Line 20 by \( p \). Since \( t_1 \) and \( t_2 \) belong to \( E \), \( p \) did not execute Line 17 during \((t_1, t_2)\). Hence, \( p \) executed Line 16 during \((t_1, t_2)\), incrementing \( \text{val} \) by one. Thus, the value of \( \text{val} \) at time \( t_2 \) is by one greater than the value of \( \text{val} \) at time \( t_1 \). To show that \( \text{val} \) always stays within the current interval of \( E \), observe the following. At the beginning of an epoch, \( \text{val} \) is set to the first value in the current interval. Furthermore, by the argument above, Line 20 gets executed at most \((2N + 1)N\) times during \( E_1 \). Consequently, \( \text{val} \) stays within the current interval at all times during \( E_1 \), and all the sequence numbers that \( \text{select}(p) \) returns during \( E_1 \) are unique and belong to the current interval of \( E \).

\( \square \)

\begin{claim}
During an epoch by process \( p \), the condition in Line 11 can be true at most \( 2N \) times.
\end{claim}

\begin{proof}
Suppose not. Let \( E \) be an epoch by \( p \) during which the condition in Line 11 holds true more than \( 2N \) times. Then, there exists an entry in A for which the condition in Line 11 is true three or more times. Let \( \lambda[q] \) be the first such entry during \( E \). Let \( t_1, t_2 \) and \( t_3 \) in \( E \) be the times the entry \( \lambda[q] \) is read in Line 10 of \( \text{select}(p) \). Let \( a_1, a_2 \), and \( a_3 \) be the values that \( \lambda[q] \) holds at times \( t_1, t_2 \), and \( t_3 \), respectively. Let \( x_1, x_2 \), and \( x_3 \) be the values that \( \text{nextStart} \) holds at times \( t_1, t_2 \), and \( t_3 \), respectively. Then, since \( a_1, a_2 \), and \( a_3 \) satisfy the condition in Line 11 of \( \text{select}(p) \), they must be of the form \( [s_1, p, \ast] \), \( [s_2, p, \ast] \), and \( [s_3, p, \ast] \), respectively, where \( s_1, s_2 \), and \( s_3 \) belong to intervals \( [x_1, x_1 \oplus_M \Delta] \), \( [x_2, x_2 \oplus_M \Delta] \), and \( [x_3, x_3 \oplus_M \Delta] \), respectively. Let \( C \) be the current interval of \( E \). Let \( E_{t_3} \) be the time interval that spans from the moment \( E \) starts until time \( t_3 \). Then, at the beginning of \( E_{t_3} \), \( C \) is set to \( [x, x \oplus_M \Delta] \), and \( \text{nextStart} \) to \( x \oplus_M \Delta \), for some \( x \). Furthermore, each time the condition in Line 11 is true, \( \text{nextStart} \) is incremented by \( \Delta \) in Line 12. Since \( \lambda[q] \) is the first entry for which the condition in Line 11 is true three or more times, it means that during \( E_{t_3} \), the condition in Line 11 could have been true at most \( 2N \) times. Hence, during \( E_{t_3} \), \( \text{nextStart} \) could have been incremented at most \( 2N - 1 \) time, and is therefore at most \( x \oplus_M (2N + 1)\Delta \) at time \( t_3 \). Then, since \( \oplus_M \) is done modulo \( M = (2N + 2)\Delta \), at no point during \( E_{t_3} \) does the interval \( [\text{nextStart}, \text{nextStart} \oplus_M \Delta] \) intersect with \( C \). Therefore, the intervals \([x_1, x_1 \oplus_M \Delta], [x_2, x_2 \oplus_M \Delta], \) and \([x_3, x_3 \oplus_M \Delta] \) are disjoint, and do not intersect with \( C \). Consequently, the sequence numbers \( s_1, s_2 \) and \( s_3 \) are distinct, and do not belong to \( C \).

Since, by an earlier argument, the condition in Line 11 could have been true at most \( 2N \) times during \( E_{t_3} \), then by Claim 8, all the sequence numbers returned by \( \text{select}(p) \) during \( E_{t_3} \) must belong to \( C \). Furthermore, it follows by construction that at all times during \((t_1, t_3)\), the latest sequence number written into \( x \) by \( p \) was returned by \( \text{select}(p) \) during \( E_{t_3} \). Consequently, at all times during \((t_1, t_3)\), if \( x \) holds a value of the form \([s, p, \ast] \), then \( s \) belongs to \( C \). This is a contradiction to the fact that the intervals \([x_1, x_2 \oplus_M \Delta], [x_2, x_3 \oplus_M \Delta] \) and \( C \) are disjoint. Hence, we have the claim.
\( \square \)

\begin{claim}
All sequence numbers returned by select during an epoch are unique and belong to that epoch’s current interval.
\end{claim}

\begin{proof}
Let \( t \) be the time at the very end of the current epoch. Then, by Claims 9 and 8, this claim trivially holds.
\( \square \)

\begin{claim}
Current intervals of two consecutive epochs are disjoint.
\end{claim}

\begin{proof}
Let \( E \) be some epoch, and \( C \) the current interval of \( E \). Then, at the beginning of \( E \), \( C \) is set to \([x, x \oplus_M \Delta]\) and \( \text{nextStart} \) to \( x \oplus_M \Delta \), for some \( x \). Furthermore, each time the condition in Line 11 is true, \( \text{nextStart} \) is incremented by \( \Delta \) in Line 12. Since, by Claim 9, the condition in Line 11 can hold true at most \( 2N \) times during an epoch, \( \text{nextStart} \) can be at most \( x \oplus_M (2N + 1)\Delta \) at the end of \( E \). Therefore, the current interval of the next epoch can at most \([x \oplus_M (2N + 1)\Delta, x \oplus_M (2N + 2)\Delta]\). Since \( \oplus_M \) is done modulo \( M = (2N + 2)\Delta \), intervals \([x \oplus_M (2N + 1)\Delta, x \oplus_M (2N + 2)\Delta]\) and \([x, x \oplus_M \Delta]\) are disjoint. Thus, the current intervals of two consecutive epoch are disjoint.
\( \square \)

\begin{lemma}
The implementation of \text{select} in Figure 7, satisfies Property I. The time complexity of the implementation is 1, and the per-process space overhead is zero.
\end{lemma}

\begin{proof}
Suppose not. Then, there exist two consecutive BitPidLL operations \( \text{op} \) and \( \text{op’} \) by \( p \), and a successful SC operation \( \text{op”} \) by \( q \), such that the following is true: \( p \) reads
Definition 6 Interval C is the `current interval' of an epoch, if C is the value of the variable \( I_p \) at the beginning of that epoch.

We introduce the following notation. Let \( E \) be some epoch. Then, for all \( k \in \{0, 1, \ldots, \lg(N + 1)\} \), let

- \( t^E_k \) denote the value of the interval \( I_p \) at the end of the \( k \)th pass of an epoch \( E \), and
- \( s^E_k \) denote the number of entries in \( \Delta \) that, at the end of the \( k \)th pass of an epoch \( E \), hold the value of the form \([s, p, 1]\), for some \( s \in I^E_k \).

In the rest of the proof, we assume that \((N+1)\) is a power of two.

Claim 12 There are \( \lg(N + 1) + 1 \) passes in an epoch.

Proof. At the beginning of an epoch, the value of the variable \( passNum_p \) is zero (by the definition of an epoch). Furthermore, an epoch ends when the Line 29 is executed the first time during that epoch, which will happens only when the condition in Line 26 is \( \text{false} \), i.e. when \( passNum_p \) reaches the value \( \lg(N + 1) \). Hence, at the end of an epoch, the value of \( passNum_p \) is \( \lg(N + 1) \). Since the variable \( passNum_p \) is incremented only in Lines 16 and 28, and it is incremented only by one, it follows that during an epoch, \( passNum_p \) goes through all the values in the range \([0 \ldots \lg(N + 1)]\). Hence, there are \( \exists \lg(N + 1) + 1 \) passes in an epoch. \( \square \)

Claim 13 In any given pass, process \( p \) invokes \( \text{select} \) exactly \( N \) times.

Proof. At the beginning of any pass, the value of \( procNum_p \) is zero (since the pass begins at Lines 16, 28, and 29, and \( procNum_p \) is set to zero at Line 15 and 25; likewise, \( procNum_p \) is set to zero at initialization time). Furthermore, a pass ends only after the variable \( procNum_p \) reaches the value \( N - 1 \) (since the variable \( passNum_p \) is modified only after the conditions in Lines 13 and 21 are \( \text{false} \)). Hence, during a pass, \( procNum_p \) goes through all the values in the range \([0\ldots N-1]\). Notice that in every invocation of \( \text{select} \) in which a pass doesn’t end, process \( p \) executes Lines 14 and 22 (since it doesn’t execute Lines 16, 28, and 29). Hence, \( p \) increments \( procNum_p \) by one in the first \( N - 1 \) invocations of \( \text{select} \) during the pass, and then ends the pass during the \( N \)th invocation. Hence, in any given pass, process \( p \) invokes \( \text{select} \) exactly \( N \) times. \( \square \)

Claim 14 Let \( E \) be some epoch by process \( p \), and \( t \in E \) the time when \( p \) completes the \( 0 \)th pass of \( E \). Let \( s \) be any sequence number. Then, if some entry \( \Delta[p] \) holds the value \([s, p, 1]\) at time \( t' \in E \), \( t' \geq t \), then \( \Delta[q] \) holds the value \([s, p, 1]\) at all times during \([t, t']\).

Proof. Suppose not. Then, at some time \( t'' \in [t, t'] \), \( \Delta[q] \) does not hold the value \([s, p, 1]\). Therefore, at some point during \([t'', t']\), the value \([s, p, 1]\) is written into \( \Delta[q] \). Since by the time \( t'' \), process \( p \) has already performed the 0th pass...
of $E$, some process other than $p$ must have written $[s, p, 1]$ into $\mathcal{A}[q]$, which is impossible. Hence, we have the claim. □

Claim 15 If $E$ is an epoch by process $p$, then the size of the interval $I_k^E$ is $((N + 1)/2^k)\Delta$, for all $k \in \{0, 1, \ldots, \lg(N + 1)\}$.

Proof. We prove this claim by induction. For the base case (i.e., $k = 0$), the observation trivially holds, since at the beginning of the 0th pass $I_p$ is initialized to be of size $(N + 1)\Delta$. Hence, $I_0^E$ is of size $(N + 1)\Delta$. The inductive hypothesis states that the size of $I_j^E$ is $((N + 1)/2^j)\Delta$, for all $j \leq k$. We now show that the size of $I_{k+1}^E$ is $((N + 1)/2^{k+1})\Delta$. By the algorithm, $I_{k+1}^E$ is a half of $I_k$. Moreover, we made an assumption earlier that $N + 1$ is a power of two. Hence, the size of $I_{k+1}^E$ is exactly $((N + 1)/2^{k+1})\Delta$. □

Claim 16 If $E$ is an epoch by process $p$, then the value of $s_k^E$ is at most $(N + 1)/2^k - 1$, for all $k \in \{0, 1, \ldots, \lg(N + 1)\}$.

Proof. For the base case (i.e., $k = 0$), the observation trivially holds, since $\mathcal{A}$ can hold at most $N$ entries. Hence, the value of $s_0^E$ is at most $N$. The inductive hypothesis states that the value of $s_k^E$ is at most $(N + 1)/2^j - 1$, for all $j \leq k$. We now show that the value of $s_{k+1}^E$ is at most $(N + 1)/2^{k+1} - 1$. Since $s_k$ is the number of entries in $\mathcal{A}$ that, at the end of the $k$th pass, hold a sequence number from $I_k^E$, it follows by Claim 14 that $p$ can count at most $s_k^E$ sequence numbers during the $(k+1)$st pass. Moreover, since $I_{k+1}^E$ is a half of $I_k^E$ with a smaller count, it follows that at most $[s_k^E/2]$ of the counted sequence numbers fall within $I_{k+1}^E$. Hence, by Claim 14, at the end of the $(k+1)$st pass, at most $[s_k^E/2]$ entries in $\mathcal{A}$ are of the form $[s, p, 1], s \in I_k^{E+1}$. Therefore, we have $s_{k+1}^E = [s_k^E/2] = \lceil((N + 1)/2^k - 1)/2\rceil$. Since $N + 1$ is a power of two, it means that $s_{k+1}^E = (N + 1)/2^{k+1} - 1$. Hence, the observation holds. □

Claim 17 Let $E$ be an epoch by process $p$. Let $I'$ be the value of the interval $I_p$ at the end of $E$. Then, $I'$ contains exactly $\Delta$ sequence numbers.

Proof. Observe that, at the end of $E$, we have $I_p = I_{\lg(N + 1)}^E$. Hence, by Claim 15, $I'$ is of the size $((N + 1)/2^{\lg(N + 1)})\Delta = ((N + 1)/(N + 1))\Delta = \Delta$. Hence, we have the claim. □

Claim 18 Let $E$ be an epoch by process $p$. Let $I'$ be the value of the interval $I_p$ at the end of $E$. Then, at the end of $E$, no entry in $\mathcal{A}$ is of the form $[s, p, 1]$, where $s \in I'$.

Proof. Observe that, at the end of $E$, the number of entries of the form $[s, p, 1]$, where $s \in I'$, is $s_{\lg(N + 1)}^E$. By Claim 16, we have $s_{\lg(N + 1)}^E = (N + 1)/2^{\lg(N + 1)} - 1 = (N + 1)/(N + 1) - 1 = 0$. Hence, we have the claim. □

Claim 19 Let $E$ be an epoch by process $p$. Then, all the sequence numbers that $\text{select}(p)$ returns during $E$ are unique and belong to the current interval of $E$.

Proof. Let $C$ be the current interval of $E$. We prove the claim in two steps. First, we prove that the total number of sequence numbers returned by $\text{select}(p)$ during $E$ is at most $N(\lg(N + 1) + 1)$. Then, we use this fact to prove that all the sequence numbers that $\text{select}(p)$ returns during $E$ are unique and belong to $C$.

During each pass of $E$, $\text{select}(p)$ returns exactly $N$ sequence numbers (by Claim 13). Since an epoch consists of $\lg(N + 1) + 1$ passes (by Claim 12), the total number of sequence numbers returned by $\text{select}(p)$ during $E$ is therefore $N(\lg(N + 1) + 1)$. Let $t_1$ and $t_2$ belong to $E$ and $p$ did not execute Line 29 during $(t_1, t_2)$. Hence, $p$ executed either Line 17, Line 23, or Line 27 during $(t_1, t_2)$, incrementing $\text{val}_p$ by one. Thus, the value of $\text{val}_p$ at time $t_2$ is by one greater than the value of $\text{val}_p$ at time $t_1$. To show that $\text{val}_p$ always stays within $C$, observe the following. At the beginning of an epoch, $\text{val}_p$ is set to the first value in $C$. Furthermore, by the argument above, Line 32 gets executed at most $N(\lg(N + 1) + 1)$ times during $E$. Consequently, $\text{val}_p$ stays within $C$ at all times during $E$. Therefore, all the sequence numbers that $\text{select}(p)$ returns during $E$ are unique and belong to $C$. □

Claim 20 Current intervals of two consecutive epochs are disjoint.

Proof. Let $E$ and $E'$ be any two consecutive epochs. Let $C (C')$ be the current interval of $E (E')$. Let $I$ be the value of the interval $I_p$ at the start of $E$. Let $I'$ be the value of the interval $I_p$ after Line 31 of $\text{select}(p)$ is executed during $E$. By Claim 17, $I$ is of the form $[x, x + M, \Delta)$, for some $x$. Therefore, $C = [x, x + M, \Delta)$, and $I' = [x + M, x + M, (N + 2)\Delta)$. Since the operation $+M$ is performed modulo $M = (N + 2)\Delta$, intervals $C$ and $I'$ are disjoint. Furthermore, since $C'$ is a subinterval of $I'$, $C$ and $C'$ are disjoint as well. □

Lemma 2 The implementation of $\text{select}$ in Figure 9, satisfies Property 1. The time complexity of the implementation is 2, and the per-process space overhead is zero.

Proof. Suppose not. Then, there exist two consecutive BitPIdLL operations $\text{op}$ and $\text{op}'$ by $p$, and a successful SC operation $\text{op}''$ by $q$, such that the following is true: $p$ reads $[s, q, v]$ in both Lines 1 and 3 of $\text{op}$, yet process $q$ writes $[s, q, x]$ in Line 6 of $\text{op}''$ before $p$ invokes $\text{op}'$. Let $t$ be the time when $q$ executes Line 6 of $\text{op}''$. Let $E$ be the $q'$s epoch at time $t$. Let $C$ be the current interval of $E$. Since, by Claim 19, all the sequence numbers returned by $\text{select}(q)$ during $E$ are unique and belong to $C$, it follows that all the sequence numbers $q$ writes into $\mathcal{A}$ during $E$ are unique and belong to $C$. Consequently, $s \in C$. 

24
Let $E'$ be the epoch that precedes $E$. (If there is no such epoch then the claim trivially holds, since, by the argument above, all the sequence number that $q$ writes to $X$ during $E$ are unique, and there can not be a process $p$ that has read $[s, q, v]$ in Lines 1 and 3 of OP before $q$ writes it in Line 6 of OP$^0$.) Let $C'$ be the current interval of $E'$. Let $t'$ be the time that $q$ reads $A_T p U$ in Line 11 during $E'$. By Claim 19, all the sequence numbers returned by select($q$) during $E' \cup E$ belong to $C' \cup C$. Furthermore, it follows by construction that at all times during $(t', t)$, the latest sequence number written into $X$ by $q$ was returned by select($q$) during $E' \cup E$. Consequently, at all times during $(t', t)$, if $X$ holds a value of the form $[s', q, s]$, then $s'$ belongs to $C' \cup C$. Since, by Claim 20, $C'$ and $C$ are disjoint, $X$ does not hold the value $[s, q, v]$ during $(t', t)$. Then, $p$ must have read $[s, q, v]$ in Line 3 of OP before $t'$. Consequently, $p$ wrote $[s, q, 0]$ into $A[p]$ in Line 2 of OP before $t'$. Since OP is $p$'s latest BitPidLL at time $t$, no other BitPidLL by $p$ wrote into $A[p]$ after Line 2 of OP and before $t$. Furthermore, no other process $r$ writes $[*, r, 1]$ into $A[p]$ after Line 2 of OP and before $t$. Hence, at all times during $(t', t)$, no process other than $q$ writes into $A[p]$. Furthermore, at time $t'$, $A[p]$ holds the value $[s, q, 0]$. Let $t'' \in E'$ be the time $p$ performs a CAS call in Line 12. Then, since no other process changes $A[p]$ during $(t', t)$, $p$'s CAS at time $t''$ succeeds, and at all times during $(t'', t)$, $A[p]$ holds the value $[s, q, 1]$. Let $I$ be the value of the interval $I_p$ at the end of $E'$. Then, we have $I = C$. Since $s \in C$, it follows that $s \in I$, which is a contradiction to Claim 18. \(\square\)