Efficient Wait-Free Implementation of Atomic Multi-Word Buffer

Rachel B. Ringel
Dartmouth College

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.dartmouth.edu/senior_theses/35

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Abstract
This paper proposes algorithms to create atomic multi-word buffers that support a single writer and multiple readers. The first algorithm given uses multi-writer, multi-reader variables whereas the second algorithm uses single-writer, multi-reader variables as a base. Both of the algorithms require $O(nm)$ space and run in $O(m)$ time for a logical read or a logical write, where $m$ is the number of words in the buffer and $n$ is the number of reading processes.

1 Introduction

Hardware-supported atomic read and write operations apply only to single words of memory. If a parallel application needs a large ($m$-word) shared object with atomic read and write operations, such an object needs to be implemented in software. The simplest way to implement an $m$-word atomic buffer $B$ is to use a lock so that only one process can manipulate the data in the buffer at once. The major disadvantage to this approach is that it can be slow; if a single process is stalled or crashes while performing a logical read or write, all processes that wish to access the data need to wait for the stalled or crashed process to finish.

Our paper gives an implementation for a buffer $B$ that supports a single writer and $n \geq 2$ readers in a wait-free manner; that is to say that any logical read or write operation can be completed in a finite number of steps, regardless of the status of other processes. We first offer an algorithm that uses multi-writer, multi-reader variables, then an algorithm that is based on single-writer, multi-reader variables. Our implementations run in $O(m)$ time for both a logical read and a logical write and require $O(nm)$ space. The best known comparable implementation [Peterson83] runs in $O(nm)$ time for a logical write and $O(m)$ for a logical read and requires $O(nm)$ space.
2 Related Work

As stated in the previous section, Peterson also presents a construction for a single-writer, multi-reader, m-word atomic buffer that uses $O(nm)$ space. Although a logical read runs in $O(m)$ time, this implementation runs in $O(nm)$ time for a logical write operation [Peterson83].

Singh, Anderson, and Gouda offer a way to construct a single-writer, multi-reader m-word register from single-writer, single-reader multi-word registers [SAG94]. Li, Tromp, and Vitányi give an algorithm to create an atomic, multi-word, multi-user variable from single-writer, single-reader single-word variables [LTV96]. Haldar and Vidyasankar create single-writer, multi-reader single-word variables from multi-reader regular variables [HV95]. Vidyasankar constructs a single-writer, multi-reader atomic register from a single-writer, multi-reader boolean atomic register and single-writer, multi-reader regular registers [Vidyasankar89]. Israeli and Shaham create multi-writer, multi-reader atomic registers from single-writer, single-reader atomic registers [IS92].

3 The Main Algorithm

Figure 2 describes our algorithm for implementing an m-word atomic buffer $B$ that supports a single writer and $n \geq 2$ readers (named $Reader_0, \ldots, Reader_{n-1}$), using multiple-writer, multiple-reader variables. We refer to read and write operations on $B$ as logical read and logical write operations, respectively. The procedure write($v$) describes how the writer executes a logical write, where $v$ is an m-word value. The procedure read($i$) describes how $Reader_i$ executes a logical read operation that returns an m-word value. We claim that our algorithm is atomic; each read operation returns the m-word value that was written by the write operation that was most recently linearized before the linearization point of the read.

In the following, we first informally describe what the various variables of the algorithm represent and how the algorithm works. We then prove the algorithm correct.

3.1 The Variables and Their Roles

In our algorithm, we write shared variables in a typewriter font (BUF), and our persistent local variables in an italicized font (index). The roles played by these variables in the algorithm are described as follows.

- $BUF[i,b]$ ($0 \leq i \leq n-1$, $b \in \{0,1\}$): The algorithm uses $2n$ buffers, each consisting of $m$ words. These $2n$ buffers are arranged into $n$ banks, with two buffers per bank: $BUF[i,0]$ and $BUF[i,1]$ are the two buffers in the $i$th bank (see Figure 1). Only the writer may write into each of the buffers, but any of the $n$ readers may read any of the buffers. We require only that the buffers to be safe: a read of $BUF[i,b]$ is guaranteed to return the latest value written in $BUF[i,b]$ only if the read does not overlap with any write operation on $BUF[i,b]$.

- $X$, a pair of integers: In the algorithm, each logical write operation writes into a single buffer. The variable $X$ indicates the buffer into which the last logical write operation wrote. Thus, if $X$ has $(i,b)$, then $BUF[i,b]$ holds the value written by the latest logical write.

- $last$, an array of integers and $index$, an integer: Another feature of the algorithm is that successive logical write operations write into buffers in successive banks in round-robin order.
Thus, the writer visits each bank once every $n$ logical write operations. If a logical write operation writes into $\text{BUF}[i, b]$, it stores the value $i$ in $\text{index}$ and the value $b$ in $\text{last}[i]$. Thus, when the writer visits bank $i$, $\text{last}[i]$ denotes the buffer of bank $i$ that was most recently written ($n$ logical write operations ago).

- **setaside**, an array of integers: If the writer is very fast relative to a reader, then any reading of a buffer by the reader might face interference from the writer (the next subsection describes such a scenario). In this case, the reader needs the writer’s help, which is provided through the setaside array. Specifically, after writing into a buffer $\text{BUF}[i, b]$, the writer checks if $\text{Reader}_i$ needs help; if it does, the writer sets $\text{BUF}[i, b]$ aside for $\text{Reader}_i$. Until $\text{Reader}_i$ signals that it no longer needs $\text{BUF}[i, b]$, this buffer won’t be touched by the writer (when the writer writes into bank $i$, it will write into $\text{BUF}[i, b]$). We define $b$ to be 1 if $b = 0$, 0 if $b = 1$.

At any point during the execution of the algorithm, the values of the variables will reflect the state of the algorithm. $X$ always points to the buffer that was written in the last logical write. $\text{last}[i]$ always indicates which of $\text{BUF}[i, 0]$ and $\text{BUF}[i, 1]$ was most recently written into.

- If $\text{Reader}_i$ is currently not performing a read operation, then $\text{setaside}[i]$ can hold any value $v \in \{-1, 0, 1\}$.

- If $\text{Reader}_i$ is performing a read and there is no chance that the buffer being read has been corrupted, then $\text{setaside}[i] = -1$, and $\text{Reader}_i$ is reading from the buffer indicated by $X$.

- If $\text{Reader}_i$ is reading from the buffer indicated by $X$ and there is a chance that that buffer has been corrupted, then $\text{setaside}[i] = v$, where $v \in \{0, 1\}$. When $\text{Reader}_i$ finishes reading that buffer, it will begin reading (and eventually return the value from) $\text{BUF}[i, v]$.

### 3.2 Explanation of the Algorithm

$\text{Reader}_i$ performs a logical read operation as follows: The reader sets $\text{setaside}[i]$ to $-1$, to indicate that it no longer needs any buffer that had been set aside for it (Line 1). It then reads a value $(j, b)$ from $X$ (Line 2), thereby learning that $\text{BUF}[j, b]$ holds the value of the latest logical write. To learn this value, the reader reads $\text{BUF}[j, b]$ (Line 3). The reader cannot rely upon the value read at Line 3, however, if that read overlaps with a writing of $\text{BUF}[j, b]$ (which is possible if the writer is very fast, as demonstrated in the next subsection). To detect such a possible overlap, the reader reads $\text{setaside}[i]$ (Line 4). If this value $a$ is $-1$, then the reader is certain that there is no overlap, and hence returns the value read at Line 3. Otherwise, $\text{BUF}[i, a]$ has been set aside by the writer and contains a legitimate value to return, as we will explain in the next subsection. So the reader returns the value in $\text{BUF}[i, a]$ (Line 6).

The writer performs a logical write operation as follows: To ensure that successive logical write operations write into successive banks, the writer increments $\text{index}$ to point to the bank where the writing should be performed (Line 7). There are two buffers at this bank, namely $\text{BUF}[\text{index}, 0]$ and $\text{BUF}[\text{index}, 1]$, and the writer uses the following rule to decide which of these two to write into. If
Figure 1: The data structure used to construct the \( m \)-word atomic buffer. Consider the \( i \)th bank to consist of \( \text{BUF}[i, 0] \) and \( \text{BUF}[i, 1] \). In the figure, \( X = [j, 0] \), meaning that the last logical write wrote into \( \text{BUF}[j, 0] \) (the shaded buffer). Since \( \text{setaside}[i] = -1 \), then \( \text{Reader}_i \) must have begun a logical read since the writer last wrote into one of the buffers in the \( i \)th bank. The next time the writer has \( \text{index} = i \), the writer will write into \( \text{BUF}[i, 1] \) because \( \text{setaside}[i] = 1 \) and \( \text{last}[i] = 0 \) (as specified in section 3.2.2).

To provide intuition for how the algorithm works, we illustrate how the helping mechanism works (Subsection 3.2.1) and why the writer avoids \( \text{last}[\text{index}] \) at Lines 9–10 (Subsection 3.2.2).

### 3.2.1 The Helping Mechanism

To illustrate how the helping mechanism works, consider the execution of a logical read operation \( R \) by \( \text{Reader}_i \). Let \( (j, b) \) be the value read from \( X \) at Line 2. From this the reader infers that the latest logical write, call it \( W \), wrote into \( \text{BUF}[j, b] \). The reader proceeds to read \( \text{BUF}[j, b] \) (Line 3). If the reader is slow relative to the writer, the reader might obtain a corrupted value from \( \text{BUF}[j, b] \) at Line 3, as the following scenario demonstrates.

While the reader is at Line 3, suppose that the writer performs \( n \) logical write operations, which we denote as \( W_{j+1}, W_{j+2}, \ldots, W_{n-1}, W_0, W_1, \ldots, W_{j-1}, W_j \). Notice that, for all \( k \), \( W_k \) writes in a buffer at bank \( k \) (because \( W \), which immediately precedes \( W_{j+1} \), wrote in \( \text{BUF}[j, b] \) and successive logical write operations write in buffers at successive banks). In particular, \( W_j \) writes into either \( \text{BUF}[j, 0] \) or \( \text{BUF}[j, 1] \). To understand the worst case, suppose that \( W_j \) writes into \( \text{BUF}[j, b] \), the buffer that \( \text{Reader}_i \) is still reading at Line 3. Then, \( \text{Reader}_i \) obtains a corrupted value from \( \text{BUF}[j, b] \).
Types
integer
valuetype = Any type

Shared Variables
setaside: array of integers
X: a pair of integers
n: integer
BUF: 3-D array of valuetype

Initialization
setaside[i] = −1, ∀i
X = (0, 0)
n = number of readers
index = 0
last[i] = 1, ∀i

procedure read
1: setaside[i] = −1
2: x = X
3: read BUF[x]
4: a = setaside[i]
5: if (a ≠ −1)
6: read(BUF[i, a])

procedure write(v)
7: index = (index + 1) mod n
8: a = setaside[index]
9: if (a == −1)
10: a = last[index]
11: write v in BUF[index, a]
12: X = (index, a)
13: last[index] = a
14: if (setaside[index] == −1)
15: setaside[index] =  ̅a

Figure 2: Multireader single-writer atomic multi-word buffer algorithm

However, in this scenario, the reader is sure to have received help from the writer, as we explain next.

When performing $W_i$, the writer reads $−1$ from setaside[i] at Line 14 (because Reader$_i$ wrote $−1$ at Line 1). So, by writing $ā$ in setaside[i] (Line 15), the writer sets aside the buffer BUF[i, ā], which holds the value written by $W_i$. When Reader$_i$ eventually performs Line 4, it reads $ā$ from setaside[i] and proceeds to read and return the value written by $W_i$ in BUF[i, ā]. Since $W_i$ is concurrent with Reader$_i$, it is legitimate for Reader$_i$ to return the value written by $W_i$.

3.2.2 The Avoidance of last[index]

Recall the Avoidance Rule that, when the writer reads $−1$ from setaside[index] at Line 8, the writer must avoid writing into BUF[index, a], where a is the value of last[index]. The following scenario demonstrates the need for this rule.

Suppose that the latest logical write operation $W$ writes into BUF[i, a] and sets last[i] to a. Then, Reader$_i$ initiates a logical read, writes $−1$ in setaside[i], reads (i, a) from X, and begins to read BUF[i, a] at Line 3. While Reader$_i$ is at Line 3, suppose that the writer performs enough logical writes for index to move from i to $i − 1$. Furthermore, suppose that the writer initiates another logical write operation $W’$. During $W’$, the writer increments index to $i$ (Line 7) and reads $−1$ from setaside[i] (because Reader$_i$ wrote $−1$ at Line 1). If the writer does not respect the Avoidance Rule, it could overwrite BUF[i, a], thereby causing Reader$_i$ (who is still reading BUF[i, a] at Line 3) to obtain a corrupted value. Now, if Reader$_i$ completes Line 3, it finds setaside[i] = $−1$ at Line 4 and returns the corrupted value. Thus, it is crucial that the writer respects the Avoidance Rule.
3.3 Proof of the Algorithm

The point where an operation is considered to have successfully completed is called the Linearization Point (LP) of the operation. In the algorithm given in Figure 2, the LP of read is Line 2 (where X is read) if the condition in Line 5 is false, otherwise the LP for read is Line 15 (where setaside[i] is assigned a new value by some call to write). The LP for write is Line 12, when X is assigned a new value.

Let the notation LP(x) denote the moment in time where operation x is linearized. For any read or write operation O, let O[i] be the time that O executes Line i. Note that all lines of the algorithm are atomic operations with the exception of Lines 3, 6, and 11

Lemma 1 For any read operation R by Reader, if the value of setaside[i] read by R at Line 4 is not −1, then there is a unique write operation W that writes into setaside[i] between R[1] and R[4].

Proof. Since there is a single writer, calls to write cannot overlap. Let us assume that W is the first write operation where R[1] < W[14] and index = i. Then W will find that setaside[i] = −1, and will execute Line 15, thereby setting setaside[i] to 0 or 1. Since for all subsequent writes, W’, where R[1] < W’[14] < R[4] and index = i, W’ will find that setaside[i] ≠ −1, then no W’ will execute Line 15, making it impossible for any W’ to change the value of setaside[i].

Lemma 2 The linearization point of a read or a write operation, O, occurs at some point during the execution of O.

Proof. For a write operation W, we defined LP(W) to be W[12]. So the lemma trivially holds for any write operation W.

For a read operation R, we defined LP(R) to be either:
(a) R[2], if at R[5] setaside[i] = −1, or
(b) W[15], if at R[5] setaside[i] ≠ −1, where W is the write operation that changes the value of setaside[i] between R[1] and R[4].

Case (a) holds trivially. Case (b) must hold since we have shown in the proof of Lemma 1 that there is a single W that changes setaside[i] between R[1] and R[4].

Lemma 3 Let R be any read operation by Readeri,
W be the latest write operation linearized before R,
a’ be the value R reads at R[4],
BUF[j, a] be the buffer into which W writes.

Then, R will read BUF[j, a], W finishes writing into BUF[j, a] before R starts reading it, and no write operation W’ that is linearized after W will write into BUF[j, a] until R has finished reading it.

Proof. There are two cases to consider based on the value of a’.

Claim 1 If a’ = −1, then
(2) R reads BUF[j, a],
If j equals i, then
There is at most one write, \( W' \), which is later than \( W \) with index = \( i \) that starts before \( R[4] \). If it exists, then \( W' \) starts after \( R[1] \) and does not write into \( \text{BUF} [i, a] \).

If \( j \neq i \), then

(4) If \( W' \) is a later write than \( W \) with index = \( j \), then \( W'[11] \) does not start until \( R[3] \) completes.

Proof.


(2) We defined \( W \) to be the latest write operation linearized before \( R \), and since \( \text{LP}(R) \) is \( R[2] \), where \( x \) is read, then \( W \) was the last logical write to change the value of \( x \) before \( R \) read it. \( W \) wrote into \( \text{BUF}[j, a] \), thus at \( W[12] \), \( W \) set \( x \) to \( (j, a) \). Therefore \( R \) would have read \( \text{BUF}[j, a] \) at \( R[3] \). Since \( a' = -1 \), Line 6 was not executed by \( R \), so the value in \( \text{BUF}[j, a] \) would be returned by \( \text{Reader}_i \).

(3) If more than one write with \( \text{index} = i \) started after \( W[12] \) but before \( R[4] \), then at least one such write would have to terminate. Let us call the first such write to terminate \( X \), and let \( W' \) be the last write with \( \text{index} = i \) to start between \( W[12] \) and \( R[4] \). Remembering that any \( W' \) would have to be linearized after \( \text{LP}(R) = R[2] \), and that since there is only a single writer, writes are executed sequentially, then,

\[
\]

We also know that \( X \) would find \( \text{setaside}[i] = -1 \), so \( X \) would execute Line 15, and consequently we would find that

\[
\]

However, this implies that \( X \) would change the value of \( \text{setaside}[i] \) to 0 or 1. This contradicts the conditions of the claim. Therefore there can only be one write, \( W' \), with \( \text{index} = i \) such that \( W[12] < W'[7] < R[4] \).

Since there must be 2 or more readers, then in order for \( W'[7] < R[1] \), then some write \( X \) for the \( \text{Reader}_x \), \( x \neq i \), must have completed and thus,

\[
\]

and this contradicts the definition of \( W \) as the latest write to be linearized before \( R \). So we can conclude that \( W' \) starts after \( R[1] \) if such a \( W' \) exists.

We know that \( W' \) won’t write into \( \text{BUF}[i, a] \) as follows: since \( R[1] < W'[7] \), then \( W'[8] \) finds \( \text{setaside}[i] = -1 \), and thus writes into \( \text{last}[i] \), which must have last been set by \( W \) to \( a \). Therefore \( W' \) will write into \( \text{BUF}[i, a] \).

(4) Let \( X \) be the earliest write linearized after \( W \) where \( \text{index} \) is \( i \), and \( W' \) be the earliest write linearized after \( W \) where \( \text{index} = j \). Since \( \text{write} \) will write into all \( \{\text{BUF}[k, \_] \mid k \neq j, 0 \leq k < \text{number of readers}\} \) before writing into \( \text{BUF}[j, \_] \) again, then \( \text{LP}(X) < \text{LP}(W') \). Thus,

\[
\]

Since at \( R[4], a' = -1 \), and \( X[15] \) would change the value of \( \text{setaside}[i] \) to something other than...
1, we know that
which shows us that \( R[3] < W'[11], \) and thereby that no write can possibly overwrite the data being read by \( R \) until \( R \) has finished reading that data.

Claim 2 If \( a' \neq -1, \)

1. \( a' = a, j = i, \) and \( W \) changed the value of \text{setaside}[i]
2. \( W \) writes into \text{BUF}[i, a'] before \( R[4], \) and
3. No write operation writes into \text{BUF}[i, a'] between \( R[4] \) and \( R[6]. \)

(1) We proved in Lemma 1 that in the time from \( R[1] \) to \( R[4], \) only a single write could change the value of \text{setaside}[i] (if this write operation is \( X, \) then \( R[1] < X[15] < R[4]. \) Since \( a' \neq -1, \)

\[ \text{LP}(R) = X[15]. \]
Recalling that \( \text{LP}(X) = X[12] \) and that all writes must be executed sequentially, we know \( X \) must have been the latest write operation to linearize before \( R, \) so \( W = X. \) Consequently, we know that \( W \) changed the value of \text{setaside}[i] to be the buffer in bank \( i \) that it wrote, so \( a' \) must equal \( a. \) Moreover, since \( W \) is able to change the value of \text{setaside}[i], index must equal \( i, \) and thus thereby \( j \) must equal \( i. \)

(2) Since \( a' \neq -1 \) at \( R[4], \) then \( W[15] < R[4]. \) Consequently,

(3) Since \( W[15] < R[4], \) we know that \( W \) set \( X = (i, a) \) at \( W[12], \) \text{last}[i] = \text{a} \) at \( W[13], \) and \text{setaside}[i] = \text{a} \) at \( W[15]. \) Thus, any write operation that occurs from \( R[4] \) to \( R[6] \) will find that \text{setaside}[i] \neq -1 \) at Line 9 and will thereby write into \text{BUF}[i, \text{last}[i]] = \text{BUF}[i, a] \) at Line 11.

Clearly, the proof for Lemma 3 follows from Claims 1 and 2.

Theorem 1 The shared register algorithm in Figure 2 is linearizable.

Proof. The theorem follows immediately from Lemmas 1, 2, and 3.

4 Algorithm Using Single-Writer Variables

In Figure 3 we propose an algorithm for a multi-reader single-writer \( m \)-word buffer for 2 or more readers that uses only single-writer, multi-reader variables, whereas the algorithm in Figure 2 uses multi-writer, multi-reader variables. This algorithm differs from the one given by Figure 2 in that the array \text{setside} is replaced by

1. An array \( R \) of integers, where \( R[i] \) is written solely by \text{Reader}_i
2. An array \( W \) of pairs of integers, where \( W \) is written solely by the writer. Let \( W[i].a \) denote the first integer of the pair at the \( i \)th index, and \( W[i].b \) the second integer of the pair.

The algorithm works the same as the algorithm given by Figure 2, with the exception that, to indicate that it is reading a buffer, \text{Reader}_i sets \( R[i] \neq W[i].a, \) and the writer sets \( W[i] = (R[i], x) \) to let the reader know that it has set aside \text{Reader}_i’s buffer \( x. \)
Types
integer
valuetype = Any type

Shared Variables Initialization
R: array of integers
W: array of pairs of integers [a, b]
X: a pair of integers
n: integer
BUF: 3-D array of valuetype

\begin{verbatim}
procedure read(retval)
1: [a, b] = W[i]
2: R[i] = a
3: x = X
4: read BUF[x]
5: [a', b'] = W[i]
6: if (a' == a) 7: read(BUF[i, b'])
8: \end{verbatim}

\begin{verbatim}
procedure write(v)
8: index = (index + 1) mod n)
9: r = R[index]
10: [c, d] = W[index]
11: if r == c
12: buf = d
13: else buf = last[index]
14: write v in BUF[buf]
15: X = [index, buf]
16: last[index] = buf
17: r = R[index]
18: if (c \neq r)
19: W[index] = [r, buf]
\end{verbatim}

Figure 3: Multireader single-writer atomic multi-word buffer algorithm using 1-writer, n-reader variables

4.1 Proof

In the algorithm given in Figure 3, the LP of \texttt{read} is Line 3 (where \texttt{X} is read) if the condition in Line 6 is false, otherwise the LP for \texttt{read} is Line 19 (where \texttt{W[i]} is assigned a new value by some call to \texttt{write}). The LP for \texttt{write} is Line 15, when \texttt{X} is assigned a new value.

Lemma 4 For any read operation \texttt{R} by Reader \texttt{i}, if the value of \texttt{W[i].a} read by \texttt{R} at Line 5 equals \texttt{R[i]}, then there is a unique write operation \texttt{W} that writes into \texttt{W[i]} between \texttt{R[1]} and \texttt{R[5]}.

Proof. Since there is a single writer, calls to \texttt{write} cannot overlap. Let us assume that \texttt{W} is the first write operation where \texttt{R[2]} \textless \texttt{W[18]} and \texttt{index} = \texttt{i}. Then \texttt{W} will find that \texttt{R[i]} \textneq \texttt{W[i].a} at Line 18, and will execute Line 19, thereby setting \texttt{W[i].a} = \texttt{R[i]}. Since for all subsequent writes, \texttt{W'}, where \texttt{R[2]} \textless \texttt{W'[18]} \textless \texttt{R[5]} and \texttt{index} = \texttt{i}, \texttt{W'} will find that \texttt{R[i]} = \texttt{W[i].a} and won’t execute Line 19, making it impossible for any \texttt{W'} to change the value of \texttt{W[i]}.

Lemma 5 The linearization point of a read or a write operation, \texttt{O}, occurs at some point during the execution of \texttt{O}.

Proof. For a write operation \texttt{W}, we defined \texttt{LP(W)} to be \texttt{W[15]}. So the lemma trivially holds for any write operation \texttt{W}.  

9
For a read operation \( R \), we defined \( \text{LP}(R) \) to be either:
(a) \( R[3] \), if at \( R[6], R[i] \neq W[i].a \), or
(b) \( W[19] \), if at \( R[6], R[i] = W[i].a \), where \( W \) is the write operation that changes the value of \( W[i] \) between \( R[2] \) and \( R[5] \).

Case (a) holds trivially. Case (b) must hold since we have shown in the proof of Lemma 4 that there is a single \( W \) that changes \( W[i] \) between \( R[2] \) and \( R[5] \).

**Lemma 6** Let \( R \) be any read operation by Reader \( i \),
\( W \) be the latest write operation linearized before \( R \),
\( \bar{a} \) be the value that \( R \) writes into \( R[i] \) at \( R[2] \),
\( [a', b'] \) be the value \( R \) reads from \( W[i] \) at \( R[5] \),
\( \text{BUF}[j, a] \) be the buffer into which \( W \) writes.

Then, \( R \) will read \( \text{BUF}[j, a] \), \( W \) finishes writing into \( \text{BUF}[j, a] \) before \( R \) starts reading it, and no write operation \( W' \) that is linearized after \( W \) will write into \( \text{BUF}[j, a] \) until \( R \) has finished reading it.

**Proof.** There are two cases to consider based on the value of \( a' \).

**Claim 3** If \( a' \neq \bar{a} \), then
1. \( W[14] \) completes before \( R[4] \) starts,
2. \( R \) reads \( \text{BUF}[j, a] \),
3. If \( j \) equals \( i \), then
   3.1. There is at most one write, \( W' \), which is later than \( W \) with index \( i \) that starts before \( R[5] \). If it exists, then
       3.1.1. \( W' \) starts after \( R[2] \) and does not write into \( \text{BUF}[i, a] \).
4. If \( j \neq i \), then
   4.1. If \( W' \) is a later write than \( W \) with index \( j \), then \( W'[14] \) does not start until \( R[4] \) completes.

**Proof.**

2. We defined \( W \) to be the latest write operation linearized before \( R \), and since \( \text{LP}(R) \) is \( R[3] \) (where \( X \) is read) then \( W \) was the last logical write to change the value of \( X \) before \( R \) read it. \( W \) wrote into \( \text{BUF}[j, a] \), thus at \( W[15] \), \( W \) set \( X \) to \( (j, a) \). Therefore \( R \) would have read \( \text{BUF}[j, a] \) at \( R[4] \). Since \( a' \neq \bar{a} \), Line 7 was not executed by \( R \), so the value in \( \text{BUF}[j, a] \) would be returned.

3. If more than one write with \( \text{index} = i \) started after \( W[15] \) but before \( R[5] \), then at least one such write would have to terminate. Let us call the first such write to terminate \( X \), and let \( W' \) be the last write with \( \text{index} = i \) to start between \( W[15] \) and \( R[5] \). Remembering that any \( W' \) would have to be linearized after \( \text{LP}(R) = R[3] \), and that since there is only a single writer, writes are executed sequentially, then,
We also know that X would find \( w[i].a \neq R[i] \), so X would execute Line 19, and consequently we would find that


However, this implies that X would change the value of \( w[i].a \) to equal \( R[i] \). This contradicts the conditions of the claim. Therefore there can only be one write, \( W' \), with index = \( i \) such that \( W[15] < W'[8] < R[5] \).

Since there must be 2 or more readers, then in order for \( W'[8] < R[2] \), then some write X for the Reader_\( x \), \( x \neq i \), must have completed and thus,


and this contradicts the definition of \( W \) as the latest write to be linearized before \( R \). So we can conclude that \( W' \) starts after \( R[2] \) if such a \( W' \) exists.

We know that \( W' \) won’t write into \( BUF[i, a] \) as follows: since \( R[2] < W'[8] \), then \( W'[10] \) finds \( w[i].a \neq R[i] \), and thus writes into \( last[i] \), which must have last been set by \( W \) to \( a \). Therefore \( W' \) will write into \( BUF[i, a] \).

(4) Let \( X \) be the earliest write linearized after \( W \) where index is \( i \), and \( W' \) be the earliest write linearized after \( W \) where index = \( j \). Since \text{write} will write into all \{ \( BUF[k, j] \mid k \neq j, 0 \leq k < \text{number of readers} \} \) before writing into \( BUF[j, j] \) again, then \( LP(X) < LP(W') \). Thus,


Since at \( R[5], w[i].a \neq R[i] \), and \( X[19] \) would change the value of \( w[i].a \) to equal \( R[i] \), we know that \( R[4] < R[5] < X[19] < W'[8] < W'[14] \), which shows us that \( R[4] < W'[14] \), and thereby that no write can possibly overwrite the data being read by \( R \) until \( R \) has finished reading that data.

**Claim 4** If \( a' = \overline{a} \), then

1. \( b' = a, j = i \), and \( W \) changed the value of \( w[i] \)
2. \( W \) writes into \( BUF[i, b'] \) before \( R[5] \), and
3. No write operation writes into \( BUF[i, b'] \) between \( R[5] \) and \( R[7] \).

(1) We proved in Lemma 4 that in the time from \( R[2] \) to \( R[5] \), only a single write could change the value of \( w[i] \) (if this write operation is \( X \), then \( R[2] < X[19] < R[5] \)). Since \( a' = \overline{a} \), \( LP(R) = X[19] \). Recalling that \( LP(X) = X[15] \) and that all writes must be executed sequentially, we know \( X \) must have been the latest write operation to linearize before \( R \), so \( W = X \). Consequently, we know that \( W \) changed the value of \( w[i].b \) to be the buffer in bank \( i \) that it wrote, so \( b' \) must equal \( a \). Moreover, since \( W \) is able to change the value of \( w[i] \), index must equal \( i \), and thus \( j \) must equal \( i \).

2. Since \( a' = \overline{a} \) at \( R[5] \), then \( W[19] < R[5] \). Consequently,

3. Since \( W[19] < R[5] \), we know that \( W \) set \( X = (i, a) \) at \( W[15], last[i] = a \) at \( W[16], \) and \( w[index] = R[i], a \) at \( W[19] \). Thus, any write operation that occurs from \( R[5] \) to \( R[7] \) will find that \( R[i] = w[index] \) at Line 11 and will thereby write into \( BUF[i, last[i]] = BUF[i, \overline{a}] \) at Line 14.

Clearly, the proof for Lemma 6 follows from Claims 3 and 4.
Theorem 2  The shared register algorithm in Figure 3 is linearizable

Proof. The theorem follows immediately from Lemmas 4, 5, and 6.

5 Algorithm with a single reader

Both of the algorithms that we present in this paper work for \( n \) readers where \( n \geq 2 \). We were able to develop a similar algorithm for a single reader and a single writer. However, this algorithm made no asymptotic improvements over the algorithm offered by Peterson when implemented with a single reader [Peterson83].

The structure of this algorithm that we discovered is very similar to those of the algorithms given in Sections 3 and 4. To implement an \( m \)-word atomic buffer for a single reader and a single-writer, we maintain the following shared variables:

- setaside. The reader sets setaside to \(-1\) to notify that it is beginning a logical read operation. If the writer, when executing a logical write, finds that setaside = \(-1\), it sets setaside to \(X\), and guarantees that it will not write into that buffer until the reader no longer has need for it.

- reading. The reader sets reading to equal \(X\) to warn that it might start reading the buffer indexed by \(X\).

- \(X\). As in the other two algorithms, \(X\) is the buffer into which the writer last wrote.

- \(\text{BUF}[i], i \in \{0, 1, 2, 3\}\). Each entry of \(\text{BUF}\) is \(m\) words long, and is written only by the writer.

As in the implementations in Sections 3 and 4, all entries of \(\text{BUF}\) are required to only be safe.

To perform a logical read, the reader first sets setaside to \(-1\) to signal that it is reading. The reader then reads the value of \(X\) and stores it in the local variable \(x\). Then, it sets reading to \(x\) to notify the writer that it might be reading the buffer indicated by reading. At this point the reader checks the value of setaside. If it still equals \(-1\), then it knows that the data there has not been corrupted, so it reads and returns the value at \(\text{BUF}[x]\). However, if setaside \(\neq -1\), then the buffer indicated by setaside has been reserved for the reader, and so the reader reads and returns the value at that buffer.

The writer begins a logical read by checking if the reader is reading. The writer accomplishes this by reading the value of setaside to see if it equals \(-1\). If it does equal \(-1\), then the writer changes setaside to equal \(X\) (the last buffer that the writer wrote), and guarantees not to write into that buffer until the reader is finished with it. The writer then chooses a value \(i\) such that \(i \in \{1, 2, 3, 4\} - \{\text{setaside, reading, } X\}\) and writes into \(\text{BUF}[i]\). To finish, the writer changes the value of \(X\) to equal \(i\).

Since this algorithm is not improvement of existing algorithms, we won’t provide a proof of it’s correctness.

6 Conclusion

We have shown how to simply implement an atomic \(m\)-word buffer for multiple readers and a single writer using both multi-writer, multi-reader variables and single-writer, multi-reader variables.
Both of these algorithms have optimal time complexity; logical reads and logical writes run in $O(m)$ time. Specifically, the $O(m)$ time complexity for a logical write is a significant improvement over the $O(nm)$ time complexity offered by Peterson’s solution, where $n$ is the number of readers [Peterson83]. Principles of distributed computing

References


