# Concurrent Reading and Writing

Leslie Lamport
Massachusetts Computer Associates

**The problem of sharing data among asynchronous processes is considered. It is assumed that only one process at a time can modify the data, but concurrent reading and writing is permitted. Two general theorems are proved, and some algorithms are presented to illustrate their use. These include a solution to the general problem in which a read is repeated if it might have obtained an incorrect result, and two techniques for transmitting messages between processes. These solutions do not assume any synchronizing mechanism other than data which can be written by one process and read by other processes.**

**Key Words and Phrases: asynchronous multiprocessing, multiprocess synchronization, readers/writers problem, shared data**

**CR Categories: 4.32, 5.24**



This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Range Measurements Laboratory under contract number FO8606-74-0068.

Author's address: Massachusetts Computer Associates, Inc., 26 Princess Street, Wakefield, MA 01880.

## Introduction

We consider the problem of the concurrent reading and writing of a common data item by separate processes: the *readers/writers problem*. We assume that the hardware solves the problem for an atomic unit of data. However, the data item may be composed of several atomic units. For example, suppose the atomic unit of data is a decimal digit and the data item is a three-digit number. If one process is reading the number while another process is changing it from 99 to 100, then the read could obtain the value 199—whereas it presumably wants to obtain either 99 or 100. In practice, the atomic unit of data might be an individual memory byte or a single disk track.

Previous solutions [1, 3] have involved mutual exclusions: all other processes are denied access to the data item while one process is modifying it. They seem to have been motivated by the case of a fairly large data file. Like most multiprocess algorithms, they assumed an a priori solution to the problem of concurrent access to the program variables (or semaphores)—presumably implemented by the hardware and/or opening system.

We are motivated by systems in which the processes may be running on separate computers. True concurrent execution is then possible, and achieving mutual exclusion may require considerable overhead. In addition to the question of overhead, there are two reasons for studying algorithms which do not involve mutual exclusions: (1) Mutual exclusion requires that a writer wait until all current read operations are completed. This may be undesirable if the writer has higher priority than the readers. (2) The concurrent reading and writing may be needed to implement mutual exclusion.

We therefore consider the problem of concurrent reading and writing without introducing mutual exclusion. We will assume that there are certain basic units of data, called *digits*, whose reading and writing are indivisible, atomic operations; i.e. we assume that the hardware automatically sequences concurrent operations to a single digit. However, a digit might contain just a single bit of data. A future paper will consider the case in which truly concurrent reading and writing is possible even at the level of the individual digit.

We only consider the case in which no two processes may try to write the same data concurrently. Mutual exclusion of writers seems unavoidable, and some other algorithm (such as the one in [5]) must be used to enforce this mutual exclusion if several processes can modify the same data.

We prove two general theorems, and then describe several sample applications. These include a simple

solution to the general readers/writer problem in which a read is repeated if it might have obtained an incorrect result, and two algorithms for sending messages from one process to another.

## General Theorems

Let $\mathbf{v}$ denote a data item composed of one or more digits. We assume that two different processes cannot concurrently modify $\mathbf{v}$. Let $\mathbf{v}^{[0]}$ denote the initial value of $\mathbf{v}$, and let $\mathbf{v}^{[1]}$, $\mathbf{v}^{[2]}$, . . . denote the successive values assumed by $\mathbf{v}$; i.e. each operation which writes $\mathbf{v}$ begins with $\mathbf{v}$ equal to $\mathbf{v}^{[i]}$, for some $i \geq 0$, and ends with $\mathbf{v}$ equal to $\mathbf{v}^{[i+1]}$. For convenience, we assume that $\mathbf{v}^{[0]}$ is written by some initial operation which precedes all read operations.

We write $\mathbf{v} = \mathbf{v}_1 \ldots \mathbf{v}_m$ to denote that the data item $\mathbf{v}$ is composed of the data items $\mathbf{v}_j$, and that each $\mathbf{v}_j$ is only written as part of a write of $\mathbf{v}$.[1] For convenience, we assume that a read (write) operation of $\mathbf{v}$ involves reading (writing) each $\mathbf{v}_j$. This implies that $\mathbf{v}^{[i]} = \mathbf{v}_1^{[i]} \ldots \mathbf{v}_m^{[i]}$ for all $i \geq 0$. If a particular read (write) operation to $\mathbf{v}$ does not require reading (writing) $\mathbf{v}_j$, then we will just pretend that a read (write) of $\mathbf{v}_j$ is performed; e.g. if the write of $\mathbf{v}^{[i]}$ does not involve writing $\mathbf{v}_j$, then we simply pretend that a write of $\mathbf{v}_j$ was performed which left its value unchanged.

If a data item $\mathbf{v}$ is not a single digit, then reading and writing $\mathbf{v}$ may involve several separate operations. A read of $\mathbf{v}$ which is performed concurrently with one or more writes to $\mathbf{v}$ may obtain a value different from any of the versions $\mathbf{v}^{[i]}$. The value obtained may contain "traces" of several different versions. If a read obtains traces of versions $\mathbf{v}^{[i_1]}, \ldots, \mathbf{v}^{[i_m]}$, then we say that it obtained a value of $\mathbf{v}^{[k,l]}$ where $k = minimum(i_1, \ldots, i_m)$ and $l = maximum(i_1, \ldots, i_m)$, so $0 \leq k \leq l$. If $k = l$, then $\mathbf{v}^{[k,l]} = \mathbf{v}^{[k]}$ and the read obtained a consistent version of $\mathbf{v}$.

As an example, suppose $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$, where the $\mathbf{d}_j$ are digits. Since reading and writing a single digit are assumed to be atomic operations, a read of $\mathbf{v}$ obtains a value $\mathbf{d}_1^{[i_1]} \ldots \mathbf{d}_m^{[i_m]}$. The value $\mathbf{d}_j^{[i_j]}$ is part of the version $\mathbf{v}^{[i_j]}$ of $\mathbf{v}$, so the read obtained a trace of that version. Hence the read obtained a value $\mathbf{v}^{[k,l]}$ where $k = minimum(i_1, \ldots, i_m)$ and $l = maximum(i_1, \ldots, i_m)$. If $k = l$, then the read obtained the consistent version $\mathbf{d}_1^{[k]} \ldots \mathbf{d}_m^{[k]} = \mathbf{v}^{[k]}$. Note that it is possible for the read to obtain a consistent version even if $k \neq l$. For example, if $\mathbf{d}_1^{[5]} = \mathbf{d}_1^{[6]}$, then a read could obtain the value $\mathbf{v}^{[5,6]} = \mathbf{d}_1^{[5]} \mathbf{d}_2^{[6]} \ldots \mathbf{d}_m^{[6]} = \mathbf{v}^{[6]}$.

For a more complicated data item $\mathbf{v}$, such as a list structure with variable pointers, different versions of $\mathbf{v}$ may consist of different sets of digits. A read operation performed while $\mathbf{v}$ is being written could read digits which were never even part of $\mathbf{v}$. It is not obvious how to define what it means in general for a read to obtain traces of version $\mathbf{v}^{[i]}$. However, to solve the readers/writer problem for $\mathbf{v}$, it suffices to insure that a read does not obtain traces of two different versions of $\mathbf{v}$. We therefore need only a necessary condition for a read to obtain traces of version $\mathbf{v}^{[i]}$. We will use the following.

If a read of $\mathbf{v}$ obtains traces of version $\mathbf{v}^{[i]}$, then:

(i) The beginning of the read preceded the end of the write of $\mathbf{v}^{[i+1]}$.

(ii) The end of the read followed the beginning of the write of $\mathbf{v}^{[i]}$.

It is easy to show that this condition is satisfied in the case $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$ considered above. The reader should convince himself that it is a reasonable assumption in general. (In fact, by properly defining "preceded" and "followed," this condition could be used to define what it means for $\mathbf{v}$ to obtain traces of version $\mathbf{v}^{[i]}$.)

Combining this condition with our definition of $\mathbf{v}^{[k,l]}$ yields the following.

*Premise*. If a read of $\mathbf{v}$ obtained the value $\mathbf{v}^{[k,l]}$, then:

(i) The beginning of the read preceded the end of the write of $\mathbf{v}^{[k+1]}$.

(ii) The end of the read followed the beginning of the write of $\mathbf{v}^{[l]}$.

This premise can be proved when $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$ for digits $\mathbf{d}_j$. It will be taken as an axiom for other types of data. Our results will be based upon this premise and the assumption that a value $\mathbf{v}^{[k,l]}$ is a correct version of $\mathbf{v}$ if $k = l$.

Let $\mathbf{v} = \mathbf{v}_1 \ldots \mathbf{v}_m$, where the $\mathbf{v}_j$ need not be digits. We say that a read (write) of $\mathbf{v}$ is performed *from left to right* if for each $j$, the read (write) of $\mathbf{v}_j$ is completed before the read (write) of $\mathbf{v}_{j+1}$ is begun. Reading or writing from right to left is defined in the analogous way. Note that we have said nothing about the order in which the digits of any single $\mathbf{v}_j$ are read or written. We now prove our first theorem.

THEOREM 1. *Let* $\mathbf{v} = \mathbf{v}_1 \ldots \mathbf{v}_m$, *and assume that* $\mathbf{v}$ *is always written from right to left. A read performed from left to right obtains a value* $\mathbf{v}_1^{[k_1,l_1]} \ldots \mathbf{v}_m^{[k_m,l_m]}$ *with* $k_1 \leq l_1 \leq k_2 \leq \ldots \leq k_m \leq l_m$.

PROOF. Since $k_j \leq l_j$, we need only show that $l_j \leq k_{j+1}$ if $1 \leq j < m$. We first show that the following five events must occur in the indicated order:

(1) end writing $\mathbf{v}_{j+1}^{[l_j]}$

(2) begin writing $\mathbf{v}_j^{[l_j]}$

(3) end reading $\mathbf{v}_j^{[k_j,l_j]}$

(4) begin reading $\mathbf{v}_{j+1}^{[k_{j+1},l_{j+1}]}$

(5) end writing $\mathbf{v}_{j+1}^{[k_{j+1}+1]}$.

The order in which items are written implies that (1) precedes (2). Premise (ii) implies that (2) precedes (3). The order in which items are read implies that (3) precedes (4). Premise (i) implies that (4) precedes (5).

[1] We use boldface type to denote a data item such as $\mathbf{v}_j$ which can be concurrently read and written. With this convention, the same number may be denoted by both ordinary and boldface type—e.g. the number $j$ in the expression $\mathbf{v}_j^{[i_j]}$.

We have thus proved that (1) precedes (5). By the definition of the versions $\mathbf{v}^{[i]}_{j+1}$, this implies that $l_j < k_{j+1} + 1$. □

Let $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$ for digits $\mathbf{d}_j$. If the $\mathbf{d}_j$ are integers, then we call $\mathbf{v}$ an *m-digit number*. We define $\mu(\mathbf{v})$ to be the $(m - 1)$-digit number $\mathbf{d}_1 \ldots \mathbf{d}_{m-1}$ composed of the leftmost $m - 1$ digits of $\mathbf{v}$. If $m = 1$, then we define $\mu(\mathbf{v})$ to equal zero. The usual relation $<$ on $m$-digit numbers is defined inductively by $\mathbf{v} < \mathbf{w} = \mathbf{e}_1 \ldots \mathbf{e}_m$ if and only if either (i) $\mu(\mathbf{v}) < \mu(\mathbf{w})$ or (ii) $\mu(\mathbf{v}) = \mu(\mathbf{w})$ and $\mathbf{d}_m < \mathbf{e}_m$. For example, we can represent a time and date by a five-digit number, where (−48) (2) (7) (14) (39) represents 14:39 o'clock on February 7, 48 B.C. The relation $<$ then means *earlier than*.

In order to prove our second theorem, we need the following result.

LEMMA. *Let* $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$ *be an m-digit number, and assume that* $i \le j$ *implies* $\mathbf{v}^{[i]} \le \mathbf{v}^{[j]}$.

(a) *If* $k_1 \le \ldots \le k_m \le k$ *then* $\mathbf{d}_1^{[k_1]} \ldots \mathbf{d}_m^{[k_m]} \le \mathbf{v}^{[k]}$.

(b) *If* $k_1 \ge \ldots \ge k_m \ge k$ *then* $\mathbf{d}_1^{[k_1]} \ldots \mathbf{d}_m^{[k_m]} \ge \mathbf{v}^{[k]}$.

PROOF. (a) The proof is by induction on $m$. If $m = 1$ then the result is trivial. Assume that $m > 1$ and the result holds for $m - 1$. Since $\mathbf{v}^{[i]} \le \mathbf{v}^{[j]}$ implies that $\mu(\mathbf{v}^{[i]}) \le \mu(\mathbf{v}^{[j]})$, we can apply the induction hypothesis to conclude that $\mathbf{d}_1^{[k_1]} \ldots \mathbf{d}_{m-1}^{[k_{m-1}]} \le \mu(\mathbf{v}^{[k_m]}) = \mathbf{d}_1^{[k_m]} \ldots \mathbf{d}_{m-1}^{[k_m]}$. This implies that $\mathbf{d}_1^{[k_1]} \ldots \mathbf{d}_m^{[k_m]} \le \mathbf{v}^{[k_m]}$. The result then follows from the hypothesis that $k_m \le k$ implies $\mathbf{v}^{[k_m]} \le \mathbf{v}^{[k]}$.

(b) The proof of (b) is obtained from that of (a) by reversing the inequalities. □

THEOREM 2. *Let* $\mathbf{v}$ *be an m-digit number and assume that* $i \le j$ *implies* $\mathbf{v}^{[i]} \le \mathbf{v}^{[j]}$.

(a) *If* $\mathbf{v}$ *is always written from right to left, then a read from left to right obtains a value* $\mathbf{v}^{[k,l]} \le \mathbf{v}^{[l]}$.

(b) *If* $\mathbf{v}$ *is always written from left to right, then a read from right to left obtains a value* $\mathbf{v}^{[k,l]} \ge \mathbf{v}^{[k]}$.

PROOF. (a) Let $\mathbf{v} = \mathbf{d}_1 \ldots \mathbf{d}_m$. Since reading and writing a single digit are atomic operations, reading the digit $\mathbf{d}_j$ can give the value $\mathbf{d}_j^{[k_j,l_j]}$ only if $k_j = l_j$. Theorem 1 then implies that the value $\mathbf{v}^{[k,l]}$ obtained by the read must equal $\mathbf{d}_1^{[k_1]} \ldots \mathbf{d}_m^{[k_m]}$ with $k_1 \le \ldots \le k_m$. Since $l = \mathit{maximum}(k_1, \ldots, k_m)$, the result follows immediately from part (a) of the lemma.

(b) The proof of part (b) is similar, using the "mirror image" of Theorem 1 and part (b) of the lemma. □

## Applications

We now give some algorithms based upon the preceding theorems. They will be described by Algol-like programs, employing some additional notation. We let $:>$ mean *set greater than*, in the same way as $:=$ means *set equal to*. Our algorithms will use $m$-digit numbers. The value of $m$ is unspecified, but it is assumed to be the same for all variables in a single algorithm. The order in which the digits of a multidigit variable are to be read or written is indicated by an arrow over the variable occurrence. Thus execution of the statement

**if** $\vec{\mathbf{x}} = \mathbf{y}$ **then** $\vec{\mathbf{z}} :> \mathbf{z}$ **fi**

reads the variable $\mathbf{x}$ by reading its digits from left to right, reads $\mathbf{y}$ by reading its digits in any order, and tests if the two values obtained are equal. If they are, then it sets $\mathbf{z}$ equal to some undetermined number greater than its previous value, writing the individual digits from right to left. (Our algorithms will not allow a variable to be set by two different processes, so it does not matter how the old value of $\mathbf{z}$ is read.)

We define the statement

**wait until** *condition*

to be equivalent to the following waiting loop:

*L*: **if not** *condition* **then goto** *L* **fi**.

We will also use a

**repeat** *body* **until** *condition*

loop, which has the obvious meaning.

Most of the following algorithms use Theorem 2, so they require variables that can have arbitrarily large values. However, practical considerations will always allow a bound to be placed on these values. For example, a variable whose value equals the current year is theoretically unbounded, but it can be satisfactorily implemented with four decimal digits.

## General Readers/Writer Solution

We first give a simple solution to the general readers/writers problem in the case of a single writer. The basic idea is to let processes read or write at any time. After reading, a process checks to see if it might have obtained an incorrect value, in which case it repeats the operation. The algorithm might be used if either (i) it is undesirable to make the writer wait for a reader to finish reading, or (ii) the probability of having to repeat a read is small enough so that it does not pay to incur the overhead of a solution employing mutual exclusion. Of course, it allows the possibility of a reader looping forever if writing is done often enough.

The reader (of this paper) should convince himself that finding such a solution is a nontrivial problem. For example, a simple "I am writing" flag will not work. Our algorithm maintains two version numbers for the data: **v1** and **v2**. The writer increments **v1** before writing the data item and increments **v2** after writing. The reader reads **v2** before reading the data item and **v1** after reading it. If it finds them equal, then it knows that it read a single version of the data.

We let **v1** and **v2** be multidigit common variables, and assume that initially **v1** = **v2**. The algorithms for reading and writing are given below. There may be any number of readers, each executing its own copy of the readers' algorithm. The writing algorithm may only be executed by one writer at a time.

| *writer* | *reader* |
|---|---|
| $\overrightarrow{\mathbf{v1}} :> \mathbf{v1}$; | **repeat** *temp* := $\overrightarrow{\mathbf{v2}}$; |
| write; | read |
| $\overleftarrow{\mathbf{v2}} := \mathbf{v1}$ | **until** $\overleftarrow{\mathbf{v1}}$ = *temp* |

We now prove that the algorithm is correct. Let **D** denote the data item which is being read and written, and let $\mathbf{v2}^{[k_1,l_1]}$, $\mathbf{D}^{[k_2,l_2]}$, $\mathbf{v1}^{[k_3,l_3]}$ denote the values of **v2**, **D**, **v1** read by a reader during a single iteration of the reading loop. We must show that if the reader decides not to read again, then this read of **D** obtained a correct value; i.e. we must show that if $\mathbf{v2}^{[k_1,l_1]} = \mathbf{v1}^{[k_3,l_3]}$, then $k_2 = l_2$.

Applying Theorem 2 to the reading and writing of **v1** and **v2**, we obtain

$$\mathbf{v2}^{[k_1,l_1]} \leq \mathbf{v2}^{[l_1]} \quad \text{and} \quad \mathbf{v1}^{[k_3]} \leq \mathbf{v1}^{[k_3,l_3]}. \tag{1}$$

Applying Theorem 1 to **v2 D v1**, we see that

$$k_1 \leq l_1 \leq k_2 \leq l_2 \leq k_3 \leq l_3. \tag{2}$$

Since $\mathbf{v2}^{[0]} = \mathbf{v1}^{[0]}$, examination of the writing algorithm shows that $l_1 \leq k_3$ implies $\mathbf{v2}^{[l_1]} \leq \mathbf{v1}^{[k_3]}$, and equality holds if and only if $l_1 = k_3$. Combining this inequality with (1), we obtain

$$\mathbf{v2}^{[k_1,l_1]} \leq \mathbf{v2}^{[l_1]} \leq \mathbf{v1}^{[k_3]} \leq \mathbf{v1}^{[k_3,l_3]}.$$

Hence, $\mathbf{v2}^{[k_1,l_1]} = \mathbf{v1}^{[k_3,l_3]}$ implies that $\mathbf{v2}^{[l_1]} = \mathbf{v1}^{[k_3]}$, which in turn implies that $l_1 = k_3$. By (2), this implies that $k_2 = l_2$, completing the proof of correctness.

Note that the converse is not true. We could have $k_2 = l_2$ even though $\mathbf{v2}^{[k_1,l_1]} \neq \mathbf{v1}^{[k_3,l_3]}$; i.e. a reader could decide to read again even though it actually obtained a correct version of **D**.

If reading **D** is an expensive operation, then the reader's *temp* := $\overrightarrow{\mathbf{v2}}$ statement should be changed to

**repeat** *temp* := $\overrightarrow{\mathbf{v2}}$ **until** $\overleftarrow{\mathbf{v1}}$ = *temp*.

This keeps a reader from performing a read operation if the writer has already begun writing.

Suppose we know that at most $P$ write operations can occur during a single iteration of the reading loop, and a single digit can assume $P + 1$ distinct values. Then we can let **v1** and **v2** be single-digit variables which cycle through $P + 1$ or more values rather than assuming a (theoretically) unbounded number of different values. However, if **D** is a data file kept in secondary storage, then it is likely to have a version number (or creation date) associated with it anyway. The algorithm just requires maintaining an extra copy of this version number.

A method similar to our algorithm was introduced in [4]. However, it uses a single version number and assumes that reading is inhibited while writing is in progress. A referee has pointed out that similar applications also appear in [7] and [8].

## Message Buffer

We now consider the problem of transmitting messages from one process to another. Assume that there is a sender process which transmits a sequence of messages to a receiver process. The sender deposits the messages one at a time in a buffer, and the receiver reads them from the buffer one at a time. Assume a buffer $B$ which can hold $P$ messages in locations $B[0]$, . . . , $B[P - 1]$. If the buffer is empty, then the receiver must wait for the next message to arrive. If the buffer is full, then the sender must wait until the receiver empties a buffer position by reading the message in it. This is also known as the producer/consumer problem [2].

The following solution uses multidigit variables **ms** and **mr** to hold the total number of messages sent and received, respectively. We assume that they are both initialized to zero.

*sender*

**wait until ms** < $\overrightarrow{\mathbf{mr}}$ + $P$;
put message in $B$[**ms** *mod* $P$];
$\overleftarrow{\mathbf{ms}}$ := **ms** + 1

*receiver*

**wait until mr** < $\overrightarrow{\mathbf{ms}}$;
read message in $B$[**mr** *mod* $P$];
$\overleftarrow{\mathbf{mr}}$ := **mr** + 1

The algorithm is quite straightforward, and its correctness is clear if the values of **mr** read by the sender and of **ms** read by the receiver are always correct. It is also easy to see that the algorithm is still correct if the values obtained by these reads are always less than or equal to the correct values, and part (a) of Theorem 2 guarantees that this is true. (Note that the sender always reads the correct value of **ms** and the receiver always reads the correct value of **mr**.)

If a digit can assume $2P$ distinct values, then this solution can be modified to make **mr** and **ms** single digit variables. The resulting algorithm, and a proof of its correctness, can be found in [6].

## Mailbox

In the preceding algorithm, a message which has been sent is not destroyed until it has been received. This is undesirable if a process may want to cancel an unreceived message. For example, suppose the message is "I want to write file $X$." The sender would like to cancel this message when it has finished writing. Using the preceding algorithm, it could simply transmit the message "I no longer want to write file $X$." However, this is unsatisfactory because the sender may have to wait for the receiver to empty the message buffer. The receiver should not have to look at that message buffer unless it wishes to use file $X$.

For such cases we want a *mailbox* that holds one message which can be written and rewritten by the sender, and read by the receiver. We are not concerned

with insuring that the receiver reads a correct version of the mailbox. If that is necessary, it can be done with any solution to the reader/writer problem, such as the one given above. Instead we will only consider the problem of letting the sender know that the current message in the mailbox has been received. For example, knowing that an "I want to write file $X$" message was received may allow a process to safely write file $X$.

This problem is easily solved by implementing mutual exclusion of accesses to the mailbox. However, mutual exclusion can be avoided by using the following technique. The sender writes a unique message number in the variable **msg.no**, and the receiver puts the number of the message it has just read into the variable **msg.rd**. More precisely, the following algorithms are used for sending and receiving. We assume that **msg.no** and **msg.rd** are multidigit numbers which are initially equal to one another.

| *sender* | *receiver* |
|---|---|
| put message in mailbox; | *temp* := $\overrightarrow{\textbf{msg.no}}$; |
| $\overleftarrow{\textbf{msg.no}}$ :> **msg.no** | read message in mailbox; |
| | $\overleftarrow{\textbf{msg.rd}}$ := *temp* |

To find out if the most recent message has been received, the sender performs the following test:

**if msg.no** = $\overrightarrow{\textbf{msg.rd}}$ **then** most recent message was received **fi**.

To prove the correctness of this test, let **mailbox** denote the mailbox and assume that the receiver reads the message in **mailbox** by executing the statement *msg* := **mailbox**. Let $\textbf{mailbox}^{[l]}$ be the current value of the mailbox, and let $\textbf{msg.rd}^{[p,\,q]}$ be the value of **msg.rd** obtained by the sender when executing the testing statement. We must show that if $\textbf{msg.rd}^{[p,\,q]} = \textbf{msg.no}^{[l]}$, then the current value of *msg* is $\textbf{mailbox}^{[l]}$. It suffices to prove that $msg^{[q]} = \textbf{mailbox}^{[l]}$, since the fact that the sender read $\textbf{msg.rd}^{[p,\,q]}$ implies by premise (ii) that the receiver has already written $msg^{[q]}$. If the $q$th version of *msg* already contained the current version of **mailbox**, then any subsequent versions of *msg* must also contain this version.

The receiving algorithm implies that $\textbf{msg.rd}^{[q]} = \textbf{msg.no}^{[k_1,\,l_1]}$ and $msg^{[q]} = \textbf{mailbox}^{[k_2,\,l_2]}$ for some $k_i$, $l_i$. We must show that $\textbf{msg.rd}^{[p,\,q]} = \textbf{msg.no}^{[l]}$ implies that $k_2 = l_2 = l$. Applying Theorem 1 to the pair of data items **msg.no**, **mailbox**, we have

$$k_1 \le l_1 \le k_2 \le l_2 \le l, \tag{3}$$

where the last inequality follows from the fact that $\textbf{mailbox}^{[l]}$ is the current value of **mailbox**. From part (a) of Theorem 2 and the fact that $l_1 \le l$, we obtain

$$\textbf{msg.rd}^{[p,\,q]} \le \textbf{msg.rd}^{[q]} = \textbf{msg.no}^{[k_1,\,l_1]} \le \textbf{msg.no}^{[l_1]} \le \textbf{msg.no}^{[l]}.$$

Therefore $\textbf{msg.rd}^{[p,\,q]} = \textbf{msg.no}^{[l]}$ implies that $\textbf{msg.no}^{[l_1]} = \textbf{msg.no}^{[l]}$, which in turn implies that $l_1 = l$. By (3), this shows that $k_2 = l_2 = l$, which completes the proof.

Note that our algorithm gives the sender a sufficient condition for the current message to have been read, but not a necessary one. The receiver could have read the current message and the prior message number, in which case the sender will not discover that the current message has been read until the receiver reads it again. This limits the applicability of this algorithm.

As an example of how such a mailbox can be used, we give a new solution to a generalized readers/writer problem (with a single writer), in which a read (write) operation consists of reading (writing) from some set of files. We assume that this set is chosen before the operation begins. We will insure that a file is not written while it is being read, so this is a mutual exclusion approach. A read operation may be performed concurrently with a write operation if it does not use any of the files being written.

Our solution gives the writer highest priority, so a reader must wait if it wants to use a file which the writer is waiting for. This allows the possibility of a reader waiting forever if writing is done very frequently.

The solution uses a single mailbox which is written by the writer and read by all the readers. The mailbox contains a set of file names. The element "$X$" in the mailbox represents an "I want to write file $X$" message. The variables **msg.rd** and *temp* of our algorithm become arrays. We also use an array **r[1:N]** of sets of file names, where $N$ is the number of readers. We let **mailbox** and each **r[i]** be initially equal to the empty set, which is denoted by $\phi$.

The following are the algorithms for the writer and for reader number $i$. Note that each program variable is written by only one process.

*writer*

**mailbox** := set of names of files to be written;
$\overleftarrow{\textbf{msg.no}}$ :> **msg.no**;
**for** $j$ := 1 **step** 1 **until** $N$ **do**
  **wait until msg.no** = $\overrightarrow{\textbf{msg.rd[j]}}$ **or r[j]** $\cap$ **mailbox** = $\phi$ **od**;
write;
**mailbox** := $\phi$

*i*th *reader*

**r[i]** := set of names of files to be read;
**repeat** *temp*[*i*] := $\overrightarrow{\textbf{msg.no}}$;
  *msg*[*i*] := **mailbox**;
  $\overleftarrow{\textbf{msg.rd[i]}}$ := *temp*[*i*]
**until r[i]** $\cap$ *msg*[*i*] = $\phi$;
read;
**r[i]** := $\phi$

We first prove that a file cannot be read while it is being written. To do this, we assume that the $i$th reader is reading and the writer is writing, and show that this implies **r[i]** $\cap$ **mailbox** = $\phi$. Consider the $i$th iteration of the writer's **for** loop before it began writing. While executing the **wait until** statement, the writer must have found (a) **msg.no** = **msg.rd[i]**, or (b) **r[i]** $\cap$ **mailbox** = $\phi$. We consider these two cases separately.

If the writer found (a) to be true, then the correctness of our mailbox algorithm implies that the $i$th reader must have read the current value of **mailbox**.

Hence, the reader must have used this current value when it last evaluated the **until** condition of its **repeat** statement, at which time it found **r[i]** $\cap$ **mailbox** = $\phi$. This proves the desired result.

Next, assume that the writer found (b) to be true. The writer either (i) did or (ii) did not use the current version of **r[i]** when it found **r[i]** $\cap$ **mailbox** = $\phi$. In case (i), the desired result is immediate. In case (ii), the writer must have read **r[i]** before the reader finished writing its current value. Hence the writer was already in its **for** loop before the reader began to read **mailbox**. This implies that the reader read the current version of **mailbox**, and the result follows as in case (a).

It is easy to see that once the writer has finished writing **mailbox**, it has priority over any reader which then begins executing its algorithm. A reader wishing to read a file whose name is in **mailbox** must then wait until the writer has finished writing. If every read operation must terminate, then the writer will eventually be able to write. However, a reader might have to wait forever.

It is interesting to observe that no precaution is taken to insure that a correct value is read when the writer reads **r[j]** or a reader reads **mailbox**. A read of any of these variables which occurs while it is being written is allowed to obtain any arbitrary value. However, without some assumption about concurrent reading and writing of **mailbox**, it is possible for a reader to wait forever even though the writer never writes any of the files it wishes to read. This can be prevented if the value $\textbf{mailbox}^{[k,\,l]}$ obtained by a reader always satisfies the condition $\textbf{mailbox}^{[k,\,l]} \subseteq \textbf{mailbox}^{[k]} \cup \textbf{mailbox}^{[k+1]} \cup \ldots \cup \textbf{mailbox}^{[l]}$. It is not hard to devise ways of reading and writing **mailbox** which satisfy this condition.

## Mutual Exclusion

The "bakery algorithm" described in [5] provides a solution to the mutual exclusion problem for $N$ processes without assuming any hardware implemented mutual exclusion. To enter its critical section, process $i$ sets **number[i]** (which is initially zero) greater than every other **number[j]**. It is allowed to enter when for each nonzero **number[j]**: either **number[i]** $<$ **number[j]**, or **number[i]** = **number[j]** and $i \leq j$.

This algorithm requires a (theoretically) unbounded amount of storage for **number[i]**. We now show how to insure a practical bound on the amount of storage needed. It suffices to insure that the value chosen for **number[i]** is at most one greater than a previously chosen value of **number[j]** for some $j$; e.g. if values are chosen at the rate of one per microsecond, then this guarantees that **number[i]** will remain less than $2^{55}$ for over a century. To do this, we assume that **number[i]** is stored as a multidigit number with non-negative digits. We then modify the algorithm so that:

(1) Process $i$ chooses the (nonzero) value of **number[i]** to be greater than or equal to its previous nonzero value, as well as greater than **number[j]** for all $j \neq i$.

(2) The value of **number[i]** is always written from right to left and read from left to right.

Theorem 2 easily proves that this modification has the desired effect. (One need only consider the nonzero values of **number[i]**, since introducing zero digits can only decrease the value which is read.) Moreover, it is easy to verify that this does not alter the validity of the three assertions proved in [5] which imply the correctness of the original algorithm.

## Conclusion

We proved two theorems and then gave several algorithms to demonstrate their use. The algorithms were actually developed first in order to solve some problems in "theoretical programming." We abstracted the essential aspects of the algorithms to form the theorems. The decreasing cost of hardware has encouraged the development of systems composed of independent computers sharing common data. The problems which arise in designing such systems will often be more complex than the theoretical problems which inspired our algorithms, and they may require different algorithms for their solution. However, they will include problems of concurrent reading and writing of shared data. When multiprogramming a single computer, such problems have traditionally been solved by using mutual exclusion, which is easily implemented with an "inhibit interrupts" operation. Such a simple approach does not work for a true multicomputer system. We hope our theorems will be useful in the problems of sharing data among different computers.

Finally, we wish to point out the common thread that runs through all of our results: writing data elements in one order and reading them in the opposite order. It is this technique which allows our algorithms to work without assuming mutual exclusion of access to shared data. The technique may have more applications than are suggested by our theorems.

Received September 1974; revised September 1976

**References**


1. Brinch Hansen, P. A comparison of two synchronizing concepts. *Acta Informatica 1*, 3 (1972), 190–199.
2. Brinch Hansen, P. Concurrent programming concepts. *Computing Surveys 5*, 4 (Dec. 1973), 223–245.
3. Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers." *Comm. ACM 14*, 10 (Oct. 1971), 667–668.
4. Easton, W.B. Process synchronization without long-term interlock. Proc. Third ACM Symp. on Operating System Principles, Operating Syst. Rev. (ACM) *6*, 1 and 2 (June 1972), 95–100.
5. Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM 17*, 8 (Aug. 1974), 453–455.
6. Lamport, L. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering SE-3*, 2 (Mar. 1977), 125–143.
7. Schaefer, M. Quasi-synchronization of readers and writers in a secure multi-level environment. TM-5407/003, System Development Corp., Santa Monica, Calif., Sept. 1974.
8. White, J.C.C. Design of a secure file management system. MTR-2931, The Mitre Corp., Bedford, Mass., June 1974.