Concurrent Reading While Writing^{*}

Gary L. Peterson

Department of Computer Science
The University of Rochester
Rochester, NY 14627

June, 1980

Revised: Feb. 1981

TR 67

Abstract

The problem of asynchronous processes reading shared data while the data is being modified by another process is considered. This problem differs from the standard readers/writers problem in that concurrent reading while writing is allowed. The model used here strongly limits the use and size of the shared variables. If multiple copies of the shared data are allowed, then simple, efficient solutions are found. In general, solutions which are more time efficient by avoiding waiting are seen to require more copies of the shared data. The number of copies used by all algorithms is shown to be the best possible. The main solution demonstrates that any system of processes which uses large distributed variables can be strongly simulated by a system which uses only binary distributed variables.

Key Words and Phrases: concurrent processing, readers/writers, shared data, mutual exclusion.

CR Categories: 4.32, 5.25.

* This research was supported in part by NSF grants Nos. MCS79-02971 and MCS77-02474.

CARLSON LIBRARY

Introduction

Consider a set of $n+1$ processes, with n readers and one writer. The processes have in common a set of shared data, such as a file or some data structure. The writer can modify the shared data, while the readers are to determine its value. If reading and writing are atomic actions, such as in the case of a shared variable, then there is no problem. But suppose the shared data cannot be atomically read or written, because it is composed of a large number of atomic units, such as an array. In this case, an algorithm must be provided which simulates atomic reads or writes so that the writer can modify the data while the readers can obtain a correct, recent value.

The classic readers/writers problem [1,2,3] is solved by using mutual exclusion where necessary to simulate atomic actions. Mutual exclusion necessarily involves waiting, but it also allows solutions to the problem with more than one writer.

Lamport considered the concurrent reading and writing problem [5] where the writer is not allowed to wait, but this allows readers to be locked out. Lamport's solution requires an unbounded number of shared variables. All the solutions given here use a bounded number of small shared variables. In addition to the above versions of the problem, the more general Concurrent Reading While Writing (CRWW) problem is studied. Copying of the shared data is permitted so that waiting time can be reduced and even eliminated. The bounds on the number of copies that need to be maintained are found to be tight for all versions of the problem considered.

One of the typical applications of mutual exclusion algorithms is to solve problems such as readers/writers. Mutual exclusion effectively creates

a serialization, and therefore decreases parallelism. However, if it is possible to solve these problems directly, without resorting to mutual exclusion, then we have gained by making better use of the power of parallel computing.

The underlying model is that of a system of asynchronous parallel processes communicating through shared variables. The shared variables are restricted to be distributed; that is, for each shared variable there is at most one process which can write it, while the others may only read it. Furthermore, the solutions desired must use only binary distributed variables, the simplest possible. This is required since a major property of a solution to the CRWW problem can be considered to be simulating a large distributed variable by a set of smaller variables. The writer of the variable uses the writer's solution and the others the readers' solutions. It is a corollary then of the main solution that a system with large distributed variables can be strongly simulated by a system which uses only binary distributed variables.

The copies of the shared data will reside in a set of buffers. The writer, when it wishes to change the shared data, will write into a subset of the buffers. A reader will obtain a correct, recent value from one of the buffers. The exact nature of a buffer will remain unspecified. Picturing it as an array should be helpful. There are only a few required properties of the buffers. Both reading and writing a buffer takes a finite amount of time. In general, there is no way a process performing a read or a write of a buffer can deduce the activities of another process. In particular, the value a reader obtains does not necessarily convey information as to whether a write

has occurred concurrently. Such information must be determined from the shared variables.

Any solution to the CRWW problem must allow readers to return a correct, recent value. If a read of a buffer occurs during a write to that buffer, the value must be considered possibly invalid and discarded. Thus a reader can only return the contents of a buffer which was obtained when the writer was not altering the buffer. In addition, the buffer contents must be recent. It would be unfair to allow readers to perpetually return an old value. Superficially, a reader must return the value of the most recent write. However, the possibility of reads overlapping with writes must be considered. First, assume that the writer performs an initial write before the readers can begin reading. Any read not overlapping a write must return the value most recently written. During a write, a reader cannot return the new value and a later read (possibly by the same process) return the old value.

Basic Solutions

Given first are two basic solutions to the problem. The first is quite simple and uses mutual exclusion to give actual atomic reads and writes on a single copy of the shared data. The second solves Lamport's reading and writing problem [5]. It has writer priority, that is, the writer never waits. Since there is only one copy of the shared data, a reader may never be able to obtain a correct value if the writer performs a rapid sequence of writes. Thus the first solution allows any process to wait, but has no lockout, while the second solution has the writer never waiting, but the readers can be locked out.

The first solution requires a mutual exclusion algorithm which uses only binary distributed variables. There are several algorithms which can be implemented with binary distributed variables, e.g., the tournament algorithm in [7] and Katseff's FIFO algorithm [4], but the algorithm in [8] uses fewer shared variables: just four per process. The algorithms for the readers and the writer are outlined below. The shared variables `reading[1..n]` and `writing` (all initially false) are used in addition to the variables used by the mutual exclusion solution. Mutual exclusion algorithms come in two parts: an entry protocol, to get into the critical section; and an exit protocol, when leaving. The procedures `Entryprotocol(i,n+1)` and `Exitprotocol(i,n+1)` execute these algorithms for the i th process in an $n+1$ process system. Note that the solution allows more than one reader to read at a time.

Algorithm for the i th reader:

```
entryprotocol(i,n+1);
wait until not writing;
reading[i] := true;
exitprotocol(i,n+1);
read buffer;
reading[i] := false;
```

Algorithm for the writer:

```
entryprotocol(n+1,n+1);
for j := 1 step 1 until n do
    wait until not reading[j];
writing := true;
exitprotocol(n+1,n+1);
write buffer;
writing := false;
```

It is easy to see that this solution is trivially generalized to the multiple writers case. The fairness properties of the solution depend upon the properties of the mutual exclusion algorithm used. The four bit solution [8] has no well-defined fairness property other than that it guarantees every process eventual access to the resource. Katseff's algorithm is FIFO, but requires $O(n)$ binary distributed variables per process.

This second algorithm does not use copying of the shared data either. The writer uses a flag to signal when it is writing the buffer. A reader can

test the flag before and after reading the buffer and determine if it partially overlapped a write. However, a write may have entirely occurred during the read, so a second switch variable is inverted after each write. This still leaves open the problem of two or more writes occurring during a read. This requires a pair of variables per reader. The reader initially sets them different, with the writer setting them equal between writes. Hence the reader, after its read of the buffer, can determine if it overlapped part of a write, an entire write, or two or more writes, in which cases it repeats. Modifying the algorithm slightly eliminates the switch variable, but increases the likelihood of a reader discarding a correct value. All variables are initially false. Only `reading[1...n]`, `writing[1...n]`, `wflag`, and `switch` are shared; the remainder are local to each process.

Algorithm for the *i*th reader:

```
T1: reading[i] := not writing[i];
T2: sflag := wflag;
   sswitch := switch;
   read buffer;
   if sflag or wflag or switch ≠ sswitch
       then goto T2;
   if reading[i] = writing[i] then goto T1;
```

Algorithm for the writer:

```
wflag := true;
write buffer;
switch := not switch;
wflag := false;
for j := 1 step 1 until n do
    if reading[j] ≠ writing[j] then
        writing[j] := reading[j];
```

It is clear that at least one buffer is necessary to solve the problem at all, and therefore the above solutions lead to the following trivial theorem.

Theorem. One buffer is necessary and sufficient to solve the CRWW problem where (1) any process is allowed to wait or (2) the writer is wait-free and readers may be locked out.

Main Solution

The next algorithm solves the CRWW problem with neither the readers nor the writer ever waiting. The algorithm takes advantage of the ability to make copies of the shared data. It requires that the shared data be kept in up to $n+2$ buffers and it is shown later that this is necessary without knowing something more about the data. The solution also demonstrates that a system with large distributed variables can be strongly simulated by a system with small (binary) distributed variables.

The second solution is the basis for the algorithm. The flag, switch, and paired reading/writing variables are used again to protect one buffer, called buff1. This buffer is the main one readers use to obtain the value of the shared data when the writer is not changing it. A second buffer, called buff2, is kept as a kind of backup. For each reader i , there is a copy buffer, called copybuff[i], that is used by the writer to give the reader a reserved copy of the data during a concurrent read while writing. The reading/writing pairs of variables are also used to signal when a copy has been made for a reader. The basic sequence for the writer is to write in buff1, make a copy for each concurrent reader, and then write in buff2. The readers' sequence is to read buff1, then buff2. Afterwards it checks to see if a copy was made, and if so it returns that value. If no copy was made and if the value from buff1 is determined to be correct, it returns that value. Otherwise it returns the value from buff2.

The correctness of the algorithm centers on the fact that in order for the writer to overlap a process reading both buff1 and buff2 it would have to make a copy for the reader between writing buff1 and buff2. Hence, it is easy

to see that the readers are assured of obtaining a correct value. Also, the value they return will be recent. If a reader returns a value from a copy buffer, then the writer was concurrently changing the shared data, and the value must be recent. Any value obtained from buff1 must be recent since it is the first buffer changed. The only problem would be if a reader obtained a new value from buff1 or a copy buffer and then later returned the old value from buff2. But the reader could return the value from buff2 only if the value from buff1 was invalid. This would mean the writer had begun a second write and by that time the value from buff2 must be the new one and not the old one.

The algorithms for the readers and the writer are given below. The shared variables and their initial values are the same as before; all other variables are local to each process.

Algorithm for the ith reader:

```

reading[i] := not writing[i];
sflag := wflag;
sswitch := switch;
read buff1;
sflag2 := wflag;
sswitch2 := switch;
read buff2;
bad1 := sswitch ≠ sswitch2
      or sflag or sflag2
if reading[i] = writing[i] then
    return copybuff[i] value;
if bad1 then
    return buff2 value;
else return buff1 value;

```

Algorithm for the writer:

```

wflag := true;
write buff1;
switch := not switch;
wflag := false;
for j := 1 step 1 until n do
    if reading[j] ≠ writing[j] then
        begin
            write copybuff[j];
            writing[j] := reading[j];
        end;
write buff2;

```

One of the most notable deficiencies of the above algorithm is that the writer, during a single pass, may have to make n copies for the readers. If the size of a buffer is large, this could be quite expensive. The simple

solution is to make only one copy for all concurrent readers using a binary flag system to inform them which copy is the most recent. A garbage collection scheme must be employed so that the writer can detect which copies have been read and can be re-used. Several simple garbage collection algorithms may be used, as long as they can be implemented using binary distributed variables. Note that still no more than n copy buffers are required since no reader ever has two copies made for it.

If one has a system with large distributed variables, then it is possible to strongly simulate a system with small (binary) distributed variables using the above solution. Every shared variable is replaced by a sequence of variables representing its bits. Each set of variables is then maintained as a single set of shared data using the main algorithm. The writer of the shared variable uses the writer's algorithm, and the others the readers' algorithm. Note that it is not necessary to have $n+2$ buffers for each shared variable of a process, but rather 2 copies of each plus n copy buffers large enough to hold the largest. A reader uses flags to first signal which variable it is going to read, and if a write of that variable occurs concurrently, the writer uses one of the copy buffers to make a copy.

That the main solution cannot be improved by reducing the number of buffers is seen from the following theorem.

Theorem. $n+2$ buffers are necessary and sufficient to solve the CRWW problem where no readers nor the writer ever wait.

Proof: The upper bound follows from the main solution while the lower bound is an inductive proof. Consider just one reader and assume two buffers

sufficient. Let the reader begin a read and proceed to where it first begins to read a buffer. Let the writer begin a series of writes. The writer is either able to alter the buffer or it cannot. If it can, then let it do so, and resume the reader. It must discard the value it just read and proceed on to again reading a buffer. Activate the writer again and repeat the process. Eventually the reader will be able to read a buffer without interference from the writer or else lockout is possible. Temporarily halt the reader while it's reading that buffer. Let the writer make a couple of passes so that the value in the first buffer is out of date. The writer during each pass must put the new value in the other buffer; otherwise the reader might resume between writes and be unable to obtain the most recent value. Halt the writer while it is changing the other buffer. Restart the reader and let it finish its current read. On its next read, however, it will find the value in the first buffer is out of date and the value in the other buffer is invalid since the writer is concurrently changing it. Therefore, it must wait until the writer finishes with the other buffer. Thus, three buffers are necessary with one reader. In the general case, assume that $n-1$ readers require $n+1$ buffers, and it will be shown that n readers require more than $n+1$ buffers by generalizing the above argument. Start by selecting a reader and have it proceed as above to a point where it is reading a buffer that cannot be changed by the writer, even with the help of the other readers. The writer makes that value out of date. Consider the $n-1$ remaining readers. They must be able to solve the CRWW problem while the first reader is temporarily halted reading that out-of-date buffer. But they cannot use that buffer since it no longer contains a recent value nor can the writer change it. This leaves them

with only n buffers, too few by the induction hypothesis. Therefore n readers require $n+2$ buffers. []

Variant Solutions

Three other versions of the CRWW problem are considered so as to show how much waiting must be allowed in order to get by with fewer buffers than the main solution. Writer priority (the writer never waits) is still required, but readers are allowed to wait, either for other readers or for the writer. The fourth algorithm of this paper solves the problem where readers may wait for the writer but not for other readers. The next solution allows readers to wait for other readers but not for the writer. Note that this is a weaker condition than the other since there is only one writer but many readers. However, the algorithm uses just three buffers instead of the other's $n+1$. The last solution weakens the problem even further to where readers can wait for any other process, but it uses only two buffers. These bounds are shown to be tight.

The fourth solution is basically the main solution with buff2 taken out. The readers, where before they would return buff2's value, will now have to restart. They will be forced to repeat until the writer either leaves buff1 alone or makes a copy. (The writer cannot change buff1 twice without making copies.) Everything else remains the same.

Algorithm for the i th reader:

```

reading[i] := not writing[i];
top: sflag := wflag;
sswitch := switch;
read buff1;
bad1 := sswitch  $\neq$  switch
      or sflag or wflag;
if reading[i] = writing[i] then
    return copybuff[i] value;
if bad1 then goto top;
else return buff1 value;

```

Algorithm for the writer:

```

wflag := true;
write buff1;
switch := not switch;
wflag := false;
for j := 1 step 1 until n do
    if reading[j]  $\neq$  writing[j] then
        begin
            write copybuff[j];
            writing[j] := reading[j];
        end;

```

One would think that allowing even this small amount of waiting would permit considerably fewer than $n+1$ buffers to be used, but the following theorem proves the algorithm is optimal.

Theorem. $n+1$ buffers are necessary and sufficient to solve the CRWW problem where the writer is waitfree and the readers wait only for the writer.

Proof: The upper bound is the fourth solution while the lower bound is a simple generalization of the previous proof. For the single reader case, one buffer is too few by again causing the reader to advance to a (the) buffer that the writer cannot change. The writer is now unable to leave the newest value of the shared data in the buffer since it cannot interfere with the reader. When the reader resumes, it will not be able to obtain the most recent value on successive attempts. Similarly, for n processes, one causes a reader to reach a buffer that cannot be changed and reduce the number of readers and buffers by one. Therefore $n+1$ buffers are necessary. []

The fifth solution allows readers to wait for other readers, but not for the writer. It uses the mutual exclusion protocols to allow one reader to read at a time, and that reader uses the main solution to obtain the value.

Since there is at most one concurrent reader, the writer needs only one extra copy, for a total of three buffers.

Algorithm for the i th reader:

```
entryprotocol(i,n);
reading[i] := not writing[i];
sflag := wflag;
sswitch := switch;
read buff1;
bad1 := sswitch  $\neq$  switch
      or sflag or wflag;
read buff2;
if reading[i] = writing[i] then
  return copybuff value;
if bad1 then
  return buff2 value;
else return buff1 value;
exitprotocol(i,n);
```

Algorithm for the writer:

```
wflag := true;
write buff1;
switch := not switch;
wflag := false;
for j := 1 step 1 until n do
  if reading[j]  $\neq$  writing[j] then
    begin
      write copybuff;
      writing[j] := reading[j];
    end;
  write buff2;
```

Theorem. Three buffers are necessary and sufficient to solve the CRWW problem where the writer is waitfree and readers only wait for other readers.

Proof: The upper bound is the fifth solution and the lower bound is a special case of the main solution theorem with one reader. (With one reader, there is no other reader to wait for.) []

The last solution shows that allowing unrestricted waiting by readers saves another buffer. Again the mutual exclusion protocols are used by the readers, but the fourth solution is used as the basis instead of the main solution.

Algorithm for the i th reader:

```
entryprotocol(i,n);
reading[i] := not writing[i];
top: sflag := wflag;
sswitch := switch;
read buff1;
bad1 := sswitch  $\neq$  switch
      or sflag or wflag;
if reading[i] = writing[i] then
    return copybuff value;
if bad1 then goto top;
else return buff1 value;
exitprotocol(i,n);
```

Algorithm for the writer:

```
wflag := true;
write buff1;
switch := not switch;
wflag := false;
for j := 1 step 1 until n do
    if reading[j]  $\neq$  writing[j] then
        begin
            write copybuff;
            writing[j] := reading[j];
        end;
```

Theorem. Two buffers are necessary and sufficient to solve the CRWW problem where the writer is waitfree and the readers may wait.

Proof: The upper bound is the sixth solution and the lower bound is again a special case of the $(n+1)$ theorem, where only one reader is considered. []

Summary

Several versions of the CRWW problem have been studied and the number of buffers required by each solution is shown to be optimal. Even with a small amount of waiting allowed, the number of buffers must be proportional to the number of readers. Only when large amounts of waiting are permitted is the necessary number of buffers reducible to a small constant. There appears to be no simple explanation of why such a jump in complexity occurs.

The main algorithm should be useful in a large number of situations. One possible objection is that it is time inefficient when the shared data is very large, due to the possibly expensive task of copying. In such a situation, however, the writer may be changing only a small part of the data at a time, so the shared data may be logically divided into a group of smaller sets of

data which can be easily copied when necessary.

Throughout this paper, the phrase "strongly simulates" has been used. This means that when one system (using large distributed variables) is reduced to another system (using binary variables), waitfree sections of code remain waitfree in the equivalent system. Other waiting properties such as lockout and deadlock are also preserved. If the first solution were used instead, waiting would be introduced where it did not exist before; therefore the original system is weakly simulated. These different notions of simulation and equivalence are explored further in [6,7].

The CRWW problem may be generalized to allow m writers. The (n,m) -CRWW problem remains unsolved for the cases where mutual exclusion is forbidden. A solution to the problem would show that a system with large shared variables, which can be written by any process, can be strongly simulated by a system with only small binary distributed variables. The definition of a correct, recent value has to be extended. Correctness remains the same unless two writers are allowed to modify the same buffer. In that case, concurrent writes of a buffer result in an incorrect value left in the buffer. Recent becomes harder to define. In general, the values returned by the readers should correspond to the reads and writes having occurred atomically; each at some point within the time interval of their actual occurrence.

Borodin has pointed out a paper by Sorenson and Hamacher [9] which considers a similar problem where N senders (writers) are sending messages (independently) to M receivers (readers). They show that $N+M+1$ copies are necessary and sufficient. Therefore this is the same bound for one writer as shown here. However, they do not permit concurrent reading while writing of a

copy. The proof is much simpler since correctness of a value is guaranteed. The application of their problem is in real-time systems. It is essential for buffering to be waitfree so that as much time as possible is saved in interprocess communication.

One of the main results of this paper is that mutual exclusion is not always necessary in order to have concurrent control. Because of people's unfamiliarity with parallel processing, when confronted by a novel problem, they invariably use a mutual exclusion approach so as to serialize the processes. This defeats the primary advantages of parallel computation over serial computation. The development of techniques which avoid serialization will make better use of concurrent computing.

References

- [1] Brinch Hansen, P., "A Comparison of Two Synchronizing Concepts," Acta Informatica 1, 3, 190-199, 1972.
- [2] Brinch Hansen, P., "Concurrent Programming Concepts," Computing Surveys 5, 4, 223-245, December 1973.
- [3] Courtois, P.J., F. Heymans, and D.L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," CACM 14, 10, 667-668, October 1971.
- [4] Katseff, H.P., "A New Solution to the Critical Section Problem," Proceedings 10th ACM Symposium on Theory of Computing, 86-88, 1978.
- [5] Lamport, L., "Concurrent Reading and Writing," CACM 20, 11, 806-811,

November 1977.

- [6] Peterson, G.L., "Time-Space Trade-Offs Between Asynchronous Parallel Models," Proceedings 11th ACM Symposium on Theory of Computing, 224-230, 1979.
- [7] Peterson, G.L., "Concurrency and Complexity," TR59, Computer Science Department, University of Rochester, August 1979.
- [8] Peterson, G.L., "A New Solution to Lamport's Concurrent Programming Control Problem Using Small Shared Variables," submitted for publication.
- [9] Sorenson, P.G. and V.C. Hamacher, "A Real-Time System Design Methodology," INFOR 13, 1, 1-18, February 1975.