A Fast Mutual Exclusion Algorithm

Leslie Lamport

November 14, 1985
revised October 31, 1986

This report appeared in the *ACM Transactions on Computer Systems, Volume 5*, Number 1, February 1987, Pages 1–11.

Author's Abstract

A new solution to the mutual exclusion problem is presented that, in the absence of contention, requires only seven memory accesses. It assumes atomic reads and atomic writes to shared registers.

Capsule Review

To build a useful computing system from a collection of processors that communicate by sharing memory, but lack any atomic operation more complex than a memory read or write, it is necessary to implement mutual exclusion using only these operations. Solutions to this problem have been known for twenty years, but they are linear in the number of processors. Lamport presents a new algorithm which takes constant time (five writes and two reads) in the absence of contention, which is the normal case. To achieve this performance it sacrifices fairness, which is probably unimportant in practical applications.

The paper gives an informal argument that the algorithm's performance in the absence of contention is optimal, and a fairly formal proof of safety and freedom from deadlock, using a slightly modified Owicki-Gries method. The proofs are extremely clear, and use very little notation.

Butler Lampson

Contents

1 Introduction

The mutual exclusion problem—guaranteeing mutually exclusive access to a critical section among a number of competing processes—is well known, and many solutions have been published. The original version of the problem, as presented by Dijkstra [2], assumed a shared memory with atomic read and write operations. Since the early 1970s, solutions to this version have been of little practical interest. If the concurrent processes are being time-shared on a single processor, then mutual exclusion is easily achieved by inhibiting hardware interrupts at crucial times. On the other hand, multiprocessor computers have been built with atomic test-and-set instructions that permitted much simpler mutual exclusion algorithms. Since about 1974, researchers have concentrated on finding algorithms that use a more restricted form of shared memory or that use message passing instead of shared memory. Of late, the original version of the problem has not been widely studied.

Recently, there has arisen interest in building shared-memory multiprocessor computers by connecting standard processors and memories, with as little modification to the hardware as possible. Because ordinary sequential processors and memories do not have atomic test-and-set operations, it is worth investigating whether shared-memory mutual exclusion algorithms are a practical alternative.

Experience gained since shared-memory mutual exclusion algorithms were first studied seems to indicate that the early solutions were judged by criteria that are not relevant in practice. A great deal of effort went into developing algorithms that do not allow a process to wait longer than it "should" while other processes are entering and leaving the critical section [1, 3, 6]. However, the current belief among operating system designers is that contention for a critical section is rare in a well-designed system; most of the time, a process will be able to enter without having to wait [5]. Even an algorithm that allows an individual process to wait forever (be "starved") by other processes entering the critical section is considered acceptable, since such starvation is unlikely to occur. This belief should perhaps be classified as folklore, since there does not appear to be enough experience with multiprocessor operating systems to assert it with great confidence. Nevertheless, in this paper it is accepted as fact, and solutions are judged by how fast they are in the absence of contention. Of course, a solution must not take much too long or lead to deadlock when there is contention.

With modern high-speed processors, an operation that accesses shared memory takes much more time than one that can be performed locally. Hence, the number of reads and writes to shared memory is a good measure of an algorithm's execution time. All the published N-process solutions that I know of require a process to execute $O(N)$ operations to shared memory in the absence of contention. This paper presents a solution that does only five writes and two reads of shared memory in this case. An even faster solution is also given, but it requires an upper bound on how long a process can remain in its critical section. An informal argument is given to suggest that these algorithms are optimal.

2 The Algorithms

Each process is assumed to have a unique identifier, which for convenience is taken to be a positive integer. Atomic reads and writes are permitted to single words of memory, which are assumed to be long enough to hold a process number. The critical section and all code outside the mutual exclusion protocol are assumed not to modify any variables used by the algorithms.

Perhaps the simplest possible algorithm is one suggested by Michael Fischer, in which process number i executes the following algorithm, where x is a word of shared memory, angle brackets enclose atomic operations, and **await** b is an abbreviation for **while** $\neg b$ **do** *skip*:

repeat await $\langle\, x = 0 \,\rangle$;
        $\langle\, x := i \,\rangle$;
        $\langle\, \mathit{delay} \,\rangle$
until $\langle\, x = i \,\rangle$;
critical section;
$x := 0$

The *delay* operation causes the process to wait sufficiently long so that, if another process j had read the value of x in its **await** statement before process i executed its $x := i$ statement, then j will have completed the following $x := j$ statement. It is traditional to make no assumption about process speeds because, when processes time-share a processor, a process can be delayed for quite a long time between successive operations. However, assumptions about execution times may be permissible in a true multiprocessor if the algorithm can be executed by a low-level operating system routine with hardware interrupts disabled. Indeed, an algorithm with busy waiting should never be used if contending processes can share a processor, since a waiting process i could be tying up a processor needed to run the other process that i is waiting for.

The algorithm above appears to require a total of only five memory access times in the absence of contention, since the delay must wait for only a single memory access to occur. However, the delay must be for the worst case access time. Since there could be $N-1$ processes contending for access to the memory, the worst case time must be at least $O(N)$ times the best case (most probable) time needed to perform a memory access.[1] Moreover, in computer systems that use a static priority for access to memory, there may not even be an upper bound to the time taken by a memory access. Therefore, an algorithm that has such a delay in the absence of contention is not acceptable.

Before constructing a better algorithm, let us consider the minimum sequence of memory accesses needed to guarantee mutual exclusion starting from the initial state of the system. The goal is an algorithm that requires a fixed number of memory accesses, independent of N, in the absence of contention. The argument is quite informal, some assertions having such flimsy justification that they might better be called assumptions, and the conclusion could easily be wrong. But even if it should be wrong, the argument can guide the search for a more efficient algorithm, since such an algorithm must violate some assertion in the proof.

Delays long enough to ensure that other processes have done something seem to require $O(N)$ time because of possible memory contention, so we may assume that no delay operations are executed. Therefore, only memory accesses need be considered. Let S_i denote the sequence of writes and reads executed by process i in entering its critical section when there is no contention—that is, the sequence executed when every read returns either the initial value or a value written by an earlier operation in S_i.

There is no point having a process write a variable that is not read by another process. Any access by S_i to a memory word not accessed by S_j can play no part in preventing both i and j from entering the critical section at the same time. Therefore, in a solution using the minimal number of memory references, all the S_i should access the same set of memory words. (Remember that S_i consists of the accesses performed in the absence

[1] Memory contention is not necessarily caused by processes contending for the critical section; it could result from processes accessing other words stored in the same memory module as x. Memory contention may be much more probable than contention for the critical section.

of contention.) Since the number of memory words accessed is fixed, independent of N, by increasing N we can guarantee that there are arbitrarily many processes i for which S_i consists of the identical sequence of writes and reads—that is, identical except for the actual values that are written, which may depend upon i. Therefore, by restricting our attention to those processes, we may assume with no loss of generality that every process accesses the same memory words in the same order.

There is no point making the first operation in S_i a read, since all processes could execute the read and find the initial value before any process executes its next step. So, the first operation in S_i should be a write of some variable x. It obviously makes no sense for the second operation in S_i to be another write to x. There is also no reason to make it a write to another variable y, since the two writes could be replaced by a single write to a longer word. (In this lower bound argument, no limit on word length need be assumed.) Therefore, the second operation in S_i should be a read. This operation should not be a read of x because the second operation of each process could be executed immediately after its first operation, with no intervening operations from other processes, in which case every process reads exactly what it had just written and obtains no new information.

Therefore, each process must perform a write to x followed by a read of another variable y. There is no reason to read a variable that is not written or write a variable that is not read, so S_i must also contain a read of x and a write of y.

The last operation in S_i, which is the last operation performed before entering the critical section in the absence of contention, should not be a write because that write could not help the process decide whether or not to enter the critical section. Therefore, the best possible algorithm is one in which S_i consists of the sequence *write* x, *read* y, *write* y, *read* x—a sequence that is abbreviated as w-x, r-y, w-y, r-x. Let us assume that S_i is of this form. Thus each process first writes x, then reads y. If it finds that y has its initial value, then it writes y and reads x. If it finds that x has the value it wrote in its first operation, then it enters the critical section.

After executing its critical section, a process must execute at least one write operation to indicate that the critical section is vacant, so processes entering later realize there is no contention. The process cannot do this with a write of x, since every process writes x as the first access to shared memory when performing the protocol. Therefore, a process must write y, resetting y to its initial value, after exiting the critical section.

Thus, the minimum sequence of memory accesses in the absence of con-

$$
\begin{array}{ll}
\textit{start:} & \langle\, x := i \,\rangle; \\
& \textbf{if } \langle\, y \neq 0 \,\rangle \textbf{ then goto } \textit{start} \textbf{ fi}; \\
& \langle\, y := i \,\rangle; \\
& \textbf{if } \langle\, x \neq i \,\rangle \textbf{ then } \textit{delay}; \\
& \qquad\qquad\qquad \textbf{if } \langle\, y \neq i \,\rangle \textbf{ then goto } \textit{start} \textbf{ fi} \quad \textbf{fi}; \\
& \textit{critical section}; \\
& \langle\, y := 0 \,\rangle
\end{array}
$$

Figure 1: Algorithm 1—process i's program.

tention that a mutual exclusion algorithm must perform is: w-x, r-y, w-y, r-x, *critical section*, w-y. This is the sequence of memory accesses performed by Algorithm 1 in Figure 1, where y is initially zero, the initial value of x is irrelevant, and the program for process number i is shown. It is described in this form, with **goto** statements, to put the operations performed in the absence of conflict at the left margin.

The delay in the second **then** clause must be long enough so that, if another process j read y equal to zero in the first **if** statement before i set y equal to i, then j will either enter the second **then** clause or else execute the critical section and reset y to zero before i finishes executing the *delay* statement. (This delay is allowed because it is executed only if there is contention.) It is shown in Section 3 that this algorithm guarantees mutual exclusion and is deadlock free. However, an individual process may be starved.

Algorithm 1 requires an upper bound not only on the time required to perform an individual operation such as a memory reference, but also on the time needed to execute the critical section. While such an upper bound may exist and be reasonably small in some applications, this is not usually the case. In most situations, an algorithm that does not require this upper bound is needed. Let us consider how many memory accesses such an algorithm must perform in the absence of contention.

Remember that the minimal protocol to enter the critical section had to be of the form w-x, r-y, w-y, r-x. Consider the following sequence of operations performed by processes 1, 2, and 3 in executing this protocol, where subscripts denote the process performing an operation:

$$w_2\text{-}x,\ w_1\text{-}x,\ r_1\text{-}y,\ r_2\text{-}y,\ w_1\text{-}y,\ w_2\text{-}y,\ r_1\text{-}x,\ w_3\text{-}x,\ r_2\text{-}x$$

At this point, process 1 can enter its critical section. However, the values that process 1 wrote in x and y have been overwritten without having been

```
start: ⟨ b[i] := true ⟩;
       ⟨ x := i ⟩;
       if ⟨ y ≠ 0 ⟩ then ⟨ b[i] := false ⟩;
                         await ⟨ y = 0 ⟩;
                         goto start   fi;
       ⟨ y := i ⟩;
       if ⟨ x ≠ i ⟩ then ⟨ b[i] := false ⟩;
                         for j := 1 to N do await ⟨ ¬b[j] ⟩ od;
                         if ⟨ y ≠ i ⟩ then await ⟨ y = 0 ⟩;
                                           goto start   fi   fi;
       critical section;
       ⟨ y := 0 ⟩;
       ⟨ b[i] := false ⟩
```

Figure 2: Algorithm 2—process i's program.

seen by any other process. The state is the same as it would have been had process 1 not executed any of its operations. Process 2 has discovered that there is contention, but has no way of knowing that process 1 is in its critical section. Since no assumption about how long a process can stay in its critical section is allowed, process 1 must set another variable to indicate that it is in its critical section, and must reset that variable to indicate that it has left the critical section. Thus, an optimal algorithm must involve two more memory accesses (in the case of no contention) than Algorithm 1. Such an algorithm is given in Figure 2, where $b[i]$ is a Boolean variable initially set to *false*. Like Algorithm 1, this algorithm guarantees mutual exclusion and is deadlock free, but allows starvation of individual processes.

In private correspondence, Gary Peterson has described a modified version of Algorithm 2 that is starvation free. However, it requires one additional memory reference in the absence of contention.

3 Correctness Proofs

There are two properties of the algorithms to be proved: mutual exclusion and deadlock freedom, the latter meaning that, if a process is trying to enter its critical section, then some process (perhaps a different one) eventually is in its critical section.

α: $\langle\, x := i \,\rangle$;
β: **if** $\langle\, y \neq 0 \,\rangle$ **then goto** α **fi**;
γ: $\langle\, y := i \,\rangle$;
$\{P_i^\delta\}$ δ: **if** $\langle\, x \neq i \,\rangle$ **then** *achieve* P_i^ϵ;
$\{P_i^\epsilon\}$ ϵ: **if** $\langle\, y \neq i \,\rangle$ **then goto** α **fi** **fi**;
$\{P_i^{cs}\}$ [ζ: *critical section*];
$\{P_i^{cs}\}$ η: $\langle\, y := 0 \,\rangle$

Figure 3: A generic algorithm—process i's program.

The proofs for both algorithms are based upon the "generic" algorithm of Figure 3, where the program for process i is shown. This program differs from Algorithm 1 in the following ways: (i) labels have been added, (ii) assertions, enclosed in curly braces, have been attached, (iii) the critical section is enclosed in square brackets, whose meaning is explained below, and (iv) the *delay* has been replaced by an *achieve* statement. The *achieve* statement represents some unspecified code to guarantee that, if and when it is finished executing, the assertion P_i^ϵ is true. More precisely, it represents a sequence of atomic operations that, if finite, includes one operation that makes P_i^ϵ true and no later operations that make P_i^ϵ false.

It is clear that this generic algorithm represents Algorithm 1 if the *achieve* statement is implemented by the *delay.* For the purpose of proving mutual exclusion, it also adequately represents Algorithm 2 if the *achieve* statement is implemented by the **for** loop in the second **then** clause. This is because, to enter its critical section, a process executes the same sequence of reads and writes of x and y in the generic algorithm as in Algorithm 2. The **await** $y = 0$ statements and the reads and writes of the $b[i]$ in Algorithm 2 can be viewed as delays in the execution of the generic algorithm. Adding delays to a program, even infinite delays, cannot invalidate a safety property such as mutual exclusion. Hence, the mutual exclusion property of the generic algorithm will imply the same property for Algorithm 2. The adequacy of the generic algorithm for proving deadlock freedom of Algorithm 2 is discussed below.

3.1 Mutual Exclusion

Mutual exclusion is a safety property, and safety properties are usually proved by assertional reasoning—for example, with the Owicki–Gries method [8]. However, since Algorithm 1 is based upon timing considerations, it cannot be proved correct with ordinary assertional methods, so a hybrid proof is given.

The assertions in Figure 3 are for a proof with the Owicki–Gries method, as described by us in [7] and Owicki and Gries in [8]. As explained below, a slight generalization of the usual Owicki–Gries method is used. Each assertion is attached to a control point, except that the square brackets surrounding the critical section indicate that the assertion P_i^{cs} is attached to every control point within the critical section. Let $\mathcal{A}_i$ denote the assertion that is true if and only if process i is at a control point whose attached assertion is true, where the trivial assertion *true* is attached to all control points with no explicit assertion. One proves that $\bigwedge_i \mathcal{A}_i$ is always true by proving that it is true of the initial state and that, for every i:

Sequential Correctness Executing any atomic action of process i in a state with $\bigwedge_j \mathcal{A}_j$ true leaves $\mathcal{A}_i$ true. This is essentially a Floyd-style proof [4] of process i, except that one can assume, for all $j \neq i$, that $\mathcal{A}_j$ is true before executing an action of i. (The assumption that $\mathcal{A}_j$ is true provides a more powerful proof method than the standard Owicki–Gries method, in the sense that simpler assertions may be used.)

Interference Freedom For each $j \neq i$, executing any atomic action of process j in a state in which $\mathcal{A}_i$ and $\mathcal{A}_j$ are true leaves $\mathcal{A}_i$ true. This proves that executing an action of process j cannot falsify an assertion attached to process i.

The assertions are chosen so that the truth of $\mathcal{A}_i \wedge \mathcal{A}_j$ implies that processes i and j are not both in their critical sections. That is, the intersection of the assertions attached to points in the critical sections of i and j equals *false*.

Assertions explicitly mention process control points, as in [7], instead of encoding them with dummy variables as Owicki and Gries did in [8]. The assertion $at(\lambda_i)$ is true if and only if control in process i is just before the statement labeled λ. The assertion $in(cs_i)$ is true if and only if control in process i is at the beginning of the critical section, within it, or right after it (and at the beginning of statement η). The assertions in Figure 3 are defined as follows:

$$
\begin{aligned}
P_i^{\delta}: &\quad x = i \supset y \neq 0 \\
P_i^{\epsilon}: &\quad y = i \supset \forall j : \neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j)) \\
P_i^{cs}: &\quad y \neq 0 \wedge \forall j \neq i : [\neg in(cs_j)] \wedge [(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j]
\end{aligned}
$$

Note that $P_i^{cs} \wedge P_j^{cs} \equiv \mathit{false}$, so proving that $\bigwedge_i \mathcal{A}_i$ is always true establishes the desired mutual exclusion property.

Since no assertions are attached to the entry point of the algorithm, or to the rest of a process's program, $\bigwedge_i \mathcal{A}_i$ is true initially. The proof of sequential correctness for process i requires the following verifications:

- *Executing γ leaves P_i^δ true.* This is obvious, since γ sets y equal to i, and $i \neq 0$.

- *If the test in statement δ finds $x = i$, causing i to enter the critical section, then P_i^{cs} is true.* The assumed truth of P_i^δ before the test implies that $y > 0$. It is obvious that, for any $j \neq i$, $(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j$ is true, since $x = i$ implies that $x \neq j$. The truth of $\neg in(cs_j)$ is proved as follows. We may assume that $\mathcal{A}_j$ is true before i executes the test. Since $at(\delta_i)$ is true, $\mathcal{A}_j$ implies that if $in(cs_j)$ is true, then P_j^{cs} is true, so $x \neq i$. Hence, if $in(cs_j)$ is true before executing the test, then the test must find $x \neq i$ and not enter the critical section. (The assumption that $\mathcal{A}_j$ is true is crucial; a more complicated program annotation is needed for a standard Owicki–Gries style proof.)

- *Upon termination of the* achieve P_i^ϵ *statement, P_i^ϵ is true.* This is the assumed semantics of the *achieve* statement.

- *If the test in statement ϵ finds $y = i$, causing i to enter the critical section, then P_i^{cs} is true.* Since $i \neq 0$, the first conjunct $(y \neq 0)$ of P_i^{cs} is obviously true if executing ϵ causes i to enter its critical section. The assumed truth of P_i^ϵ before executing ϵ implies that, if $y = i$, then for all $j \neq i$: $\neg(at(\gamma_j) \vee at(\delta_j) \vee in(cs_j))$ is true. This in turn implies the truth of the second conjunct of P_i^{cs} before the execution of ϵ, which implies the truth of that conjunct after the execution of ϵ, since executing the test does not affect control in any other process.

- *Executing any step of the critical section leaves P_i^{cs} true.* This follows from the implicit assumption that a process does not modify x or y while in the critical section, and the fact that executing one process does not affect control in another process.

The second part of the Owicki–Gries method proof, showing noninterference, requires proving that no action by another process j can falsify any of the assertions attached to process i. Note that the implication $A \supset B$ can be falsified only by making A true or B false.

P_i^δ: Process i is the only one that sets x to i, so process j can falsify P_i^δ only by setting y to zero. It does this only by executing statement η.

However, the assertion P_j^{cs}, which is assumed to be true when j executes η, states that, if process i is at control point δ, then $x \neq i$, in which case setting y to zero does not falsify P_i^{δ}.

P_i^{ϵ}: Only process i sets y to i, so j can falsify this assertion only by reaching control point γ or δ or by entering its critical section when $y = i$. However, it cannot reach δ without being at γ, it can reach γ only by executing the test at β and finding $y = 0$, and, if it is not at δ, it can enter its critical section only by executing the test at ϵ and finding $y = j$, none of which are possible when $y = i$.

P_i^{cs}: Since P_i^{cs} asserts that no other process is at control point η, no other process can make $y \neq 0$ become false. To show that no other process j can make $in(cs_j)$ become true, observe that it can do so only in two ways: (i) by executing the test at statement δ with $x = j$, or (ii) by executing ϵ and finding $y = j$. The first is impossible because P_i^{cs} asserts that if j is at δ then $x \neq j$, and the second is impossible because P_j^{ϵ}, which is assumed to be true at that point, asserts that if $y = j$ then $in(cs_i)$ is false, contrary to the hypothesis.

Finally, we must show that process j cannot falsify $(at(\gamma_j) \vee at(\delta_j)) \supset x \neq j$. It could do this only by reaching control point γ, which it can do only by executing the test in statement β and finding y equal to zero. However, this is impossible because P_i^{cs} asserts that $y \neq 0$.

This completes the proof of the mutual exclusion property for the generic algorithm of Figure 3. To prove that Algorithms 1 and 2 satisfy this property, it is necessary to prove that the program for process i correctly implements the *achieve* P_i^{ϵ} statement. In these proofs, control points in the two algorithms will be labeled by the same names as the corresponding control points in the generic algorithm. Thus, ϵ is the control point just before the **if** test in the second **then** clause.

Let γ–η denote the set of control points consisting of γ, δ, all control points in the critical section, and η. For Algorithm 1, we must show that, if at the end of the delay $y = i$, then no other process j has control in γ–η. Since no other process can set y to i, if y equals i upon completion of the delay, then it must have equaled i at the beginning of the delay. If process j has not yet entered γ–η by the time i began executing the *delay* statement, then it cannot enter before the end of the *delay* statement, because the only way j can enter γ–η is by executing β when $y = 0$ or ϵ when $y = j$, both of which are impossible with $y = i$. By assumption, the delay is chosen to be long enough so that any process in γ–η at the beginning of the delay will have exited before the end of the delay. Hence, at the end of the delay, no process is in γ–η, so P_i^ϵ is true.

This completes the proof of mutual exclusion for Algorithm 1. Note how behavioral reasoning was used to prove that P_i^ϵ holds after the delay. An assertional proof of this property would be quite difficult, requiring the introduction of an explicit clock and complicated axioms about the duration of operations.

It is not difficult to convert the proof for the generic algorithm into a completely assertional proof for Algorithm 2, and this will be left as an exercise for the reader who wants a completely rigorous proof. A less formal behavioral proof is given here. Once again, we must prove that, if $y = i$ when control reaches ϵ, then no other process j is in γ–η. As in Algorithm 1, if y equals i when process i reaches ϵ, then it must have equaled i throughout the execution of the **for** statement. Hence, if process j is outside γ–η some time during the execution of i's **for** statement, then it is not in γ–η when i reaches ϵ. However, $b[j]$ is true when process j is in γ–η. To reach ϵ, process i must find $b[j]$ false when executing the **for** loop, so j was not in γ–η at that time and is thus not in it when i reaches ϵ. This completes the proof of mutual exclusion for Algorithm 2.

3.2 Deadlock Freedom

Deadlock freedom means that, if a process tries to enter the critical section, then it or some other process must eventually be in the critical section. This is a liveness property, which can be proved formally using temporal logic—for example, with the method of Owicki and Lamport [9]. However, only an informal sketch of the proof will be given. The reader who is well versed in temporal logic will be able to flesh out the informal proof into a formal one in the style of Owicki and Lamport.

Once again, correctness is proved first for the generic algorithm of Figure 3. Let $in(\delta_i)$ be true if and only if control in process i is at the beginning of or within the statement δ, but not within the **then** clause of ϵ. Deadlock freedom rests upon the following safety property:

S. $y = i \neq 0 \supset (in(\delta_i) \vee in(cs_i))$

It is a simple matter to show that this assertion is true initially and is left true by every program action, so it is always true.

For convenience, the proof will be expressed in terms of some simple temporal assertions—assertions that are true or false at a certain time during the execution. For any temporal assertions P and Q, the assertion $\Box P$ (read *henceforth* P) is true at some instant if and only if P is true then and at all later times; and $P \leadsto Q$ (read P *leads to* Q) is true at some instant if P is false then, or Q is true then or at some future time. A precise semantics of the temporal operators $\Box$ and $\leadsto$ can be found in [9].

Deadlock freedom is expressed by the formula $at(\alpha_i) \leadsto \exists j : in(cs_j)$, which is proved by assuming that $at(\alpha_i)$ and $\Box(\forall j\colon \neg in(cs_j))$ are true and obtaining a contradiction. (This is a proof by contradiction, based upon the temporal logic tautology $\neg(P \leadsto Q) \equiv (P \wedge \Box\neg Q)$.) The proof is done by demonstrating a sequence of $\leadsto$ relations ($A_1 \leadsto A_2$, $A_2 \leadsto A_3$, etc.) leading to *false*, which is the required contradiction. Note that when one of these relations is of the form $P \leadsto Q \wedge \Box R$, we can assume that $\Box R$ is true in all the subsequent proofs. (Once $\Box R$ becomes true, it remains true forever.) Also note that $P \supset Q$ implies $P \leadsto Q$.

The proof requires the following assumption about the *achieve* statement:

T. If process i executes the *achieve* statement with $\Box(y = i \wedge \forall j\colon \neg in(cs_j))$ true, then that statement will terminate.

The sequence of $\leadsto$ relations is given below.

$at(\alpha_i) \leadsto y \neq 0$ Process i either finds $y \neq 0$ in statement β or else sets y to i in the following statement.

$y \neq 0 \supset \Box y \neq 0$ Once y is nonzero, it can be set to zero only by some process executing statement η. However, this cannot happen since we are assuming $\Box\forall j\colon \neg in(cs_j)$.

$(\Box y \neq 0) \leadsto \exists j\colon \Box y = j$ Once y becomes and remains forever nonzero, no process can reach statement γ that has not already done so. Eventually, all the processes that are at γ will execute γ, after which the value of y remains the same.

$(\Box y = j) \leadsto at(\epsilon_j)$ By the invariant S, $y = j$ implies $in(cs_j) \vee in(\delta_j)$. Since we have assumed $\Box\neg in(cs_j)$, this implies that control in process j is within δ and, if it is at the beginning of δ, must find $x \neq j$. By Assumption T, this implies that control in process j must eventually reach ϵ.

$(\Box y = j \wedge at(\epsilon_j)) \leadsto$ *false* Process j must eventually execute the test in statement ϵ, find $y = j$, and enter the critical section, contradicting the assumption $\Box \neg in(cs_j)$.

This completes the proof of deadlock freedom for the generic algorithm. Since Assumption T is obviously true for Algorithm 1, this proves deadlock freedom for Algorithm 1. For Algorithm 2, observe that the proof for the generic algorithm remains valid even if the two **goto**'s can be delayed indefinitely. Thus, the proof holds for Algorithm 2 even though a process can remain forever in an **await** $\langle y = 0 \rangle$ statement. To prove the deadlock freedom of Algorithm 2, it suffices to prove Assumption T, that $\Box(y = i \wedge \forall j : \neg in(cs_j))$ implies that process i's **for** loop eventually terminates. This is easy to see, since $b[j]$ must eventually become false and remain forever false for every process j. A more formal proof, in the style of Owicki and Lamport [9], is left as an exercise for the reader.

Acknowledgements

I wish to thank Jeremy Dion and Michael Powell for bringing the problem to my attention. Michael Powell independently discovered the basic *write x*, *read y*, *write y*, *read x* mutual exclusion protocol used in the algorithms. I also wish to thank Michael Fischer for his comments on the problem and on the manuscript.

References

[1] N. G. deBruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 8(9):137–138, March 1967.

[2] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[3] Murray A. Eisenberg and Michael R. McGuire. Further comments on Dijkstra's concurrent programming control problem. *Communications of the ACM*, 15(11):999, November 1972.

[4] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium on Applied Math., Vol. 19*, pages 19–32, American Mathematical Society, 1967.

[5] Anita K. Jones and Peter Schwarz. Experience using multiprocessor systems—A status report. *ACM Computing Surveys*, 12(2):121–165, June 1980.

[6] D. E. Knuth. Additional commments on a problem in concurrent program control. *Communications of the ACM*, 9(5):321, May 1966.

[7] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[8] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[9] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.