A Fine-Grained Solution to the Mutual Exclusion Problem*

James H. Anderson†
Department of Computer Science
The University of Maryland at College Park
College Park, Maryland 20742-3255

December 1991, Revised August 1992

Abstract

We present a "fine-grained" solution to the mutual exclusion problem. A program is *fine-grained* if it uses only single-reader, single-writer boolean variables and if each of its atomic operations has at most one occurrence of at most one shared variable. In contrast to other fine-grained solutions that have appeared in the literature, processes in our solution do not busy-wait, but wait on one another only by executing **await** statements. Such statements can be implemented in practice either by means of context switching or by means of "local" spinning. We show that our algorithm is correct even if shared variables are accessed nonatomically.

Keywords: Busy-waiting, mutual exclusion, nonatomic operations, shared data, synchronization primitives.

CR Categories: D.4.1, D.4.2, F.3.1

1 Introduction

The mutual exclusion problem is a paradigm for resolving conflicting accesses to shared resources and has long been recognized as one of the classic problems in concurrent programming. In this problem, each of a set of processes repeatedly executes a program fragment known as its "critical section". Intuitively, the critical section of a process is a section of code in which all accesses to a shared resource are confined. In order to execute its critical section, a process must first execute another program fragment, its "entry section"; upon the termination of its critical section, a process must execute a third program fragment, its "exit section". The entry and exit sections for each process must be designed so as to ensure that (i) at most one process executes its critical section at any time, and (ii) each process in its entry section eventually executes its critical section.

*To appear in *Acta Informatica*.

†Work supported, in part, by NSF Contract CCR 9109497 and by the Center of Excellence in Space Data and Information Sciences.

The mutual exclusion problem has been studied for many years, dating back to the seminal paper of Dijkstra [3]. Since then, many solutions have been proposed, most of which are quite complicated and difficult to understand. A notable exception is an especially simple solution presented by Peterson in [10]. In Peterson's paper, a two-process solution is presented and then generalized to apply to an arbitrary number of processes. Although Peterson's algorithm requires only read/write atomicity, it employs shared variables that can be read and written by multiple processes. As such, its correctness is predicated upon the existence of an underlying mechanism for properly linearizing concurrent reads and concurrent writes of the same shared variable. A refinement of Peterson's algorithm in which only single-writer variables are used was later presented by Kessels in [5]. Although Kessels' algorithm is more fine-grained than Peterson's, it still employs multi-reader shared variables.

In this paper, we present a mutual exclusion algorithm, based upon that of Kessels, in which only single-reader, single-writer boolean variables are used and in which each atomic operation has at most one occurrence of at most one shared variable. We call such a solution *fine-grained*. Previous fine-grained algorithms include the "One-Bit" algorithm discovered independently by Burns and Lynch [2] and by Lamport [7] and also the boolean variable implementation of the algorithm presented by Peterson in [11]. (Strictly speaking, none of these algorithms is fine-grained as presented because each employs multi-reader shared variables. Also, the One-Bit algorithm allows individual processes to starve, i.e., it fails to satisfy requirement (ii) in the first paragraph of this section. However, by adding an extra boolean variable to one of the processes, it is possible to obtain a two-process version of this algorithm that is starvation-free [8].) Like these previous solutions, our algorithm is correct even if shared variables are accessed nonatomically; thus, no underlying mechanism is required for linearizing statements that access the same shared variable. However, unlike these other solutions, processes in our algorithm employ **await** statements — specifically, statements of the form "**await** X", where X is a shared, single-reader, single-writer boolean variable — rather than busy-waiting loops in order to wait on one another. As discussed in Section 6, such statements can be efficiently implemented either by performing a context switch or by means of "local" spinning [9]. The algorithms given in [2, 7, 11] all employ busy-waiting loops in which shared variables are repeatedly read *and* updated and thus do not admit such implementations.

The rest of this paper is organized as follows. In Section 2, we present our model of concurrent programs. In Section 3, we describe our solution to the mutual exclusion problem informally, and in Section 4, we formally establish its correctness. In Section 5, we show that our algorithm is correct even if shared variables are accessed nonatomically. Concluding remarks appear in Section 6.

2 Concurrent Programs

In this section, we present our model of concurrent programs and define the basic relations used in reasoning about such programs. A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements, and is specified using guarded commands [4] and **await** statements. An **await** statement has the form "**await** B", where B is a predicate over program variables. This statement is enabled for execution only when predicate B is true and is atomically executed (when enabled) by transferring control to the next

executable statement. Each *variable* of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be accessed by more than one process. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k. For an example of the syntax we employ for programs, see Figure 2.

A program's semantics is defined by its set of "fair histories". The definition of a fair history, which is given below, formalizes the requirement that each statement of a program is subject to weak fairness. Before giving the definition of a fair history, we introduce a number of other concepts; all of these definitions apply to a given concurrent program.

A *state* is an assignment of values to the variables of the program. One or more states are designated as *initial states*. If state u can be reached from state t via the execution of statement s, then we say that s is *enabled* at state t and we write $t \xrightarrow{s} u$. (Unless otherwise indicated, in the case of a **do** or **if** statement, "execution of statement s" means the evaluation of each guard in the statement's set of guards, and the subsequent transfer of control.) If statement s is not enabled at state t, then we say that s is *disabled* at t. A *history* is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state. A history may be either finite or infinite; in the former case, it is required that no statement be enabled at the last state of the history. A history is *fair* if it is finite or if it is infinite and each statement is either disabled at infinitely many states of the history or is infinitely often executed in the history. Note that this fairness requirement implies that each continuously enabled statement is eventually executed. Unless otherwise noted, we henceforth assume that all histories are fair.

When reasoning about the correctness of a concurrent program, safety properties are defined using invariants and progress properties are defined using leads-to assertions. A first-order predicate P (over program variables) is an *invariant* of a program iff it is true in each state of every history of that program. Predicate P *leads-to* predicate Q in a given program, denoted $P \mapsto Q$, iff for each history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ of the program, if P is true at some state t_i, then Q is true at some state t_j where $j \geq i$.

3 Fine-Grained Mutual Exclusion

In this section, we present a fine-grained program that solves the mutual exclusion problem. First, we state the conditions that must be satisfied by such a program. In the mutual exclusion problem, there are N processes, each of which has the following structure.

do *true* $\rightarrow$
    Noncritical Section;
    Entry Section;
    Critical Section;
    Exit Section
od

It is assumed that each process begins execution in its noncritical section. It is further assumed that each critical section execution terminates. By contrast, a process is allowed to halt in its

```
process i            {i ranges over 0..N − 1}
do true →
        Noncritical Section;
        ENTRY(i, 0);
        ENTRY(i, 1);
        ⋮
        ENTRY(i, N − 1);
        Critical Section;
        EXIT(i, N − 1);
        EXIT(i, N − 2);
        ⋮
        EXIT(i, 0)
od
```

Figure 1: N-process mutual exclusion algorithm.

noncritical section. No variable appearing in any entry or exit section may be referred to in any noncritical or critical section (except, of course, program counters). Let $ES(i)$ ($CS(i)$) be a predicate that is true iff the value of process i's program counter equals a label of a statement appearing in its entry section (critical section). Then, the requirements that must be satisfied by a program that solves this problem are as follows.

- *Mutual Exclusion*: $(\forall i, j : i \neq j :: CS(i) \Rightarrow \neg CS(j))$ is an invariant. Informally, at most one process can execute its critical section at a time.
- *Progress*: $ES(i) \mapsto CS(i)$ holds for each i. Informally, if a process is in its entry section, then that process eventually executes its critical section.

We also require that each process in its exit section eventually enters its noncritical section; this requirement is trivially satisfied by our solution (and most others), so we will not consider it further.

As in [5, 10], we obtain our solution to the mutual exclusion problem by solving the two-process case and by using two-process mutual exclusion to solve the N-process case. We use a well-known approach to do the latter. In particular, as illustrated in Figure 1, we obtain a solution to the N-process case by "nesting" $N-1$ different two-process solutions. In this figure, ENTRY(i, j) denotes the entry section from a two-process solution that process i executes to compete with process j; if $i = j$, then ENTRY(i, j) simply equals "**skip**". EXIT(i, j) is defined similarly. If the underlying two-process solution satisfies the Mutual Exclusion and Progress requirements, then it is straightforward to show that the N-process solution does as well. (In establishing the Progress requirement, it is important that the two-process entry sections be executed in "increasing" order; this is similar to a two-phase locking protocol, where deadlock is avoided by locking data items in a fixed linear order.) If the underlying two-process solution is fine-grained, then the N-process program is also fine-grained.

(In [5], Kessels describes a different approach for using two-process mutual exclusion to solve N-process mutual exclusion. In this approach, a two-process solution is applied in a binary arbitration tree. Associated with each link in the tree is an entry section and an exit section. The entry and exit sections associated with the two links connecting a given node to its sons constitute a two-process mutual exclusion algorithm. Initially, all processes are "located" at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each link on this path. Upon exiting its critical section, a process traverses this path in reverse, this time executing the exit section of each link. Observe that Kessels' approach is more efficient than that described in the previous paragraph. In particular, for a process to enter its critical section, $O(\log_2 N)$ two-process entry sections must be executed rather than $O(N)$. However, because different processes may traverse the same link during the course of the algorithm, in a direct application of this approach, the entry and exit sections associated with each link must employ multi-writer variables. Depending on the two-process solution being used, the resulting construction may require a solution to the very problem it attempts to solve. In particular, in order to replace such multi-writer variables by single-reader, single-writer, boolean ones, an underlying fine-grained solution to the mutual exclusion problem may be required.)

Our two-process algorithm is depicted in Figure 2. The two processes are denoted u and v. With the exception of statements 0, 8, and 10, the two processes are identical. (Note that the statements of the program are not labeled in increasing linear order; this is done in order to facilitate the Progress proof.) The program is similar to the two-process solution given by Peterson in [10] and also to that given by Kessels in [5], but uses only single-reader, single-writer boolean variables.

The two variables $T[u]$ and $T[v]$ together correspond to the variable $TURN$ of Peterson's algorithm, and are used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. Process u attempts to establish $T[u] = T[v]$ and process v attempts to establish $T[u] \neq T[v]$. Informally, process u enters its critical section only if $T[u] \neq T[v]$ holds or if process v has not expressed interest in entering its critical section. Similarly, process v enter its critical section only if $T[u] = T[v]$ holds or if process u has not expressed interest in entering its critical section. Thus, in the event of a "tie", process u is favored if $T[u]$ and $T[v]$ differ in value and process v is favored if $T[u]$ and $T[v]$ are equal. This is essentially the idea of Kessels' algorithm. In our algorithm, each process checks the condition that is required for it to enter its critical section by waiting on one single-reader, single-writer boolean variable. The manner in which this is accomplished is explained next.

Because $T[u]$ is written only by process u, process u can keep track of its value by using a private variable; variable $u.x$ is used for this purpose. In order for process u to wait until $T[u] \neq T[v]$ holds, it simply tests $u.x$ and then waits for $T[v]$ to have the appropriate value. In particular, if $u.x$ is true (which implies that $T[u]$ is true), then process u waits until $T[v]$ is false, and if $u.x$ is false (which implies that $T[u]$ is false), then process u waits until $T[v]$ is true. Process u waits for $T[v]$ to have the appropriate value by waiting on either $P[v]$ or $Q[v]$. As explained next, these variables serve the dual purpose of "signaling" the value of $T[v]$ and "signaling" that process v is in its noncritical section.

Loosely speaking, $P[v]$ is used by process v to signal to process u that $T[v]$ is false, and $Q[v]$ is similarly used to signal that $T[v]$ is true. While the value of $T[v]$ is being determined in statements

shared var P, Q, T : **array**$[u, v]$ **of boolean**
initially $P[u] = true \ \wedge \ P[v] = true \ \wedge \ Q[u] = true \ \wedge \ Q[v] = true$

process u

private var $u.x$: **boolean**
initially $u.x = T[u]$

do $true \ \rightarrow$
3: Noncritical Section;
2: $P[u] := false$;
1: $Q[u] := false$;
0: $u.x := T[v]$;
12: $T[u] := u.x$;
11: **if** $u.x \ \rightarrow$
10: $P[u] := true$;
9: **await** $P[v]$
$[\!] \ \neg u.x \ \rightarrow$
8: $Q[u] := true$;
7: **await** $Q[v]$
fi;
6: Critical Section;
5: $P[u] := true$;
4: $Q[u] := true$
od

process v

private var $v.x$: **boolean**
initially $v.x = T[v]$

do $true \ \rightarrow$
3: Noncritical Section;
2: $P[v] := false$;
1: $Q[v] := false$;
0: $v.x := \neg T[u]$;
12: $T[v] := v.x$;
11: **if** $v.x \ \rightarrow$
10: $Q[v] := true$;
9: **await** $P[u]$
$[\!] \ \neg v.x \ \rightarrow$
8: $P[v] := true$;
7: **await** $Q[u]$
fi;
6: Critical Section;
5: $P[v] := true$;
4: $Q[v] := true$
od

Figure 2: Two-process mutual exclusion algorithm.

0 and 12 of process v, the appropriate value to signal is not known; thus, to ensure that process u does not enter its critical section when it should not, $P[v]$ and $Q[v]$ are both kept equal to false while this value is being determined. When process v is in its noncritical section (where it may halt), process u should not be blocked in its entry section; thus, while v is in its noncritical section, $P[v]$ and $Q[v]$ are both kept equal to true. In this way, $P[v]$ and $Q[v]$ serve both purposes mentioned in the previous paragraph. Variables $P[u]$ and $Q[u]$ are similarly used by process u to signal the value of $T[u]$ to process v, except their roles are reversed: $P[u]$ is used to signal that $T[u]$ is true, and $Q[u]$ is used to signal that $T[u]$ is false. This concludes our informal explanation of the two-process algorithm.

A formal correctness proof for the algorithm is given in the next section. As this proof shows, the protocol that is followed in assigning values to $P[u]$, $Q[u]$, $P[v]$, and $Q[v]$ ensures that both processes never execute their critical sections at the same time. Thus, the Mutual Exclusion requirement is satisfied. The proof also shows that whenever one of the processes waits on some predicate, that predicate eventually becomes true and remains true. By our definition of a fair history, this implies that the Progress requirement is satisfied. In order to establish the Progress requirement, we first

show that the program is free from deadlocks, i.e., it is never the case that both processes are simultaneously waiting on predicates that are false. Ensuring that deadlock does not occur can be especially tricky in a fine-grained solution that uses **await** statements: unlike a solution that employs busy-waiting, a process cannot break a potential deadlock by assigning new values to shared variables while it is waiting.

4 Correctness Proof

We prove that the program of Figure 2 satisfies the Mutual Exclusion and Progress requirements. We begin by presenting notational conventions that will be used in the remainder of this section, and then discuss our proof obligations in establishing various assertions.

Notational Conventions: Unless otherwise specified, we assume that i ranges over $\{u, v\}$. We denote statement number k of process i as $k.i$. We let $Enabled(k.i)$ be a predicate that is true iff statement $k.i$ is enabled. Let S be a subset of the statement labels in process i. Then, $i@\{S\}$ holds iff the program counter for process i equals some value in S. The following is a list of symbols we will use ordered by increasing binding power: $\equiv$, $\mapsto$, $\Rightarrow$, $(\vee, \wedge)$, $(=, \neq, >)$, $(+, -)$, $\neg$, $(., @)$, $(\{, \}, [,])$. The symbols enclosed in parentheses have the same priority. □

Proof Obligations: Two approaches are used in this paper for the purpose of proving that a given assertion is an invariant. Under the first approach, the proof obligation is to show that the given assertion holds initially and is stable. An assertion P is *stable* for a program iff for each statement $k.i$ of that program, $P \ \wedge \ Enabled(k.i) \ \Rightarrow \ wp(k.i, \ P)$ holds, where wp is the "weakest precondition" predicate transformer [4]. The second approach involves proving that the given assertion follows from a set of known invariants. Specifically, if $P_0, \ \ldots, \ P_k$ are invariants and if $P_0 \wedge \cdots \wedge P_k \ \Rightarrow \ Q$ holds, then we can conclude that Q is also an invariant.

The basic approach that we follow in proving $P \mapsto Q$ involves the use of an integer well-founded ranking R, where $R > 0 \ \Rightarrow \ P$ and $R = 0 \ \Rightarrow \ Q$. Under this approach, the proof obligation is to show that if $R > 0$, then (i) there exists some enabled statement that is eventually executed, and (ii) each statement execution decreases the value of R. Several of the leads-to assertions that are required in the correctness proof are quite trivial. For such cases, we either note that the given assertion follows directly from the program text and our fairness assumption, or appeal to various simple properties of the leads-to relation. An example of such a property is transitivity, i.e., if $U \mapsto V$ and $V \mapsto W$, then $U \mapsto W$. □

The following simple invariants, which are stated without proof, follow directly from the program text.

invariant $\quad i@\{9, 10\} \ \Rightarrow \ i.x$ (I0)

invariant $\quad i@\{7, 8\} \ \Rightarrow \ \neg i.x$ (I1)

invariant $\neg i@\{12\} \;\Rightarrow\; T[i] = i.x$ (I2)

invariant $u@\{0,1,7,8,11,12\} \;\Rightarrow\; \neg P[u]$ (I3)

invariant $u@\{0,9,10,11,12\} \;\Rightarrow\; \neg Q[u]$ (I4)

invariant $v@\{0,1,9,10,11,12\} \;\Rightarrow\; \neg P[v]$ (I5)

invariant $v@\{0,7,8,11,12\} \;\Rightarrow\; \neg Q[v]$ (I6)

invariant $i@\{3,4\} \;\Rightarrow\; P[i]$ (I7)

invariant $i@\{2,3\} \;\Rightarrow\; Q[i]$ (I8)

For each of the remaining invariants, a correctness proof is given.

invariant $u@\{6,7,9\} \;\Rightarrow\; P[u] = u.x \;\wedge\; Q[u] = \neg u.x$ (I9)

Proof: We prove that (I9) is initially true and stable. The former is straightforward, as $u@\{6,7,9\}$ is initially false. To prove that (I9) is stable, it suffices to consider those statements that may establish the antecedent or modify $u.x$, $P[u]$, or $Q[u]$. The statements to check are $0.u$, $1.u$, $2.u$, $4.u$, $5.u$, $8.u$, and $10.u$. Note that $u@\{6,7,9\}$ is false after the execution of $0.u$, $1.u$, $2.u$, $4.u$, and $5.u$. Thus, these statements do not falsify (I9). To see that statement $8.u$ does not falisfy (I9), consider the following derivation.

I9 $\wedge$ *Enabled*$(8.u)$

$\Rightarrow$ $u@\{8\}$, by the program text.

$\Rightarrow$ $\neg u.x \;\wedge\; \neg P[u]$, by (I1) and (I3).

$\Rightarrow$ $wp(8.u,\; \neg u.x \;\wedge\; \neg P[u] \;\wedge\; Q[u])$, $8.u$ establishes $Q[u] = true$ and does not modify $u.x$ or $P[u]$.

$\Rightarrow$ $wp(8.u,$ I9$)$, by the definition of (I9) and the monotonicity of wp.

The proof that statement $10.u$ does not falsify (I9) is similar, except that (I0) and (I4) are used instead of (I1) and (I3). □

invariant $v@\{6,7,9\} \;\Rightarrow\; P[v] = \neg v.x \;\wedge\; Q[v] = v.x$ (I10)

Proof: Similar to the proof of (I9). □

invariant $u@\{6\} \Rightarrow (\ (T[u] \wedge (P[v] \vee v@\{0..5\} \vee (\neg v.x \wedge v@\{8,11,12\}))) \vee$
$(\neg T[u] \wedge (Q[v] \vee v@\{0..5\} \vee (v.x \wedge v@\{10,11,12\}))) \ ) \qquad$ (I11)

Proof: We prove that (I11) is initially true and stable. Establishing the former is straightforward, as $u@\{6\}$ is initially false. To prove that (I11) is stable, we must consider those statements that may establish the antecedent or falsify the consequent. The antecedent is established by statements $7.u$ and $9.u$. To see that statement $7.u$ does not falsify (I11), observe the following.

I11 $\wedge$ $Enabled(7.u)$

$\Rightarrow$ I11 $\wedge$ $Q[v]$ $\wedge$ $u@\{7\}$, by the program text.

$\Rightarrow$ $Q[v] \wedge \neg T[u]$, by (I1) and (I2).

$\Rightarrow$ $wp(7.u,\ Q[v] \wedge \neg T[u] \wedge u@\{6\})$, by the program text.

$\Rightarrow$ $wp(7.u,$ I11) , by the definition of (I11) and the monotonicity of wp.

The proof that $9.u$ does not falsify (I11) is similar, except that (I0) is used instead of (I1).

The consequent may potentially be falsified by any statement that modifies $T[u]$, falsifies $P[v]$ or $Q[v]$, or falsifies any of the predicates $v@\{0..5\}$, $\neg v.x \wedge v@\{8,11,12\}$, or $v.x \wedge v@\{10,11,12\}$. The statements to check are $12.u$, $0.v$, $1.v$, $2.v$, $8.v$, $10.v$, and $11.v$. By the program text, $u@\{6\}$ is false after the execution of $12.u$, $v.x = \neg T[u] \wedge v@\{12\}$ is true after the execution of $0.v$, and $v@\{0..5\}$ is true after the execution of $1.v$ and $2.v$. It follows, then, that these statements do not falsify (I11). In the remainder of the proof, we consider statements $8.v$, $10.v$, and $11.v$. For statement $8.v$, we have the following.

I11 $\wedge$ $Enabled(8.v)$

$\Rightarrow$ I11 $\wedge$ $v@\{8\}$, by the program text.

$\Rightarrow$ I11 $\wedge$ $v@\{8\} \wedge \neg Q[v]$, by (I6).

$\Rightarrow$ $\neg u@\{6\} \vee (T[u] \wedge (P[v] \vee (\neg v.x \wedge v@\{8\})))$, by the definition of (I11).

$\Rightarrow$ $wp(8.v,\ \neg u@\{6\} \vee (T[u] \wedge P[v]))$, $8.v$ establishes $P[v] = true$.

$\Rightarrow$ $wp(8.v,$ I11) , by the definition of (I11) and the monotonicity of wp.

The proof that statement $10.v$ does not falsify (I11) is similar, except that (I5) is used instead of (I6). This leaves only statement $11.v$. To see that $11.v$ does not falsify (I11), consider the following derivation.

I11 $\wedge$ $Enabled(11.v)$

$\Rightarrow$ I11 $\wedge$ $v@\{11\}$, by the program text.

$\Rightarrow$ I11 $\wedge$ $v@\{11\}$ $\wedge$ $\neg P[v]$ $\wedge$ $\neg Q[v]$, by (I5) and (I6).

$\Rightarrow$ $\neg u@\{6\} \vee (T[u] \wedge \neg v.x \wedge v@\{11\}) \vee (\neg T[u] \wedge v.x \wedge v@\{11\})$
, by the definition of (I11).

$\Rightarrow$ $wp(11.v,\ \neg u@\{6\} \vee (T[u] \wedge \neg v.x \wedge v@\{8\}) \vee (\neg T[u] \wedge v.x \wedge v@\{10\}))$
, $11.v$ establishes $v@\{8\}$ or $v@\{10\}$,
depending on the value of $v.x$, and
doesn't modify $u@\{6\}$, $T[u]$, or $v.x$.

$\Rightarrow$ $wp(11.v,\ \text{I11})$, by the definition of (I11) and
the monotonicity of wp. □

invariant $v@\{6\} \Rightarrow (\ (T[v] \wedge (P[u] \vee u@\{0..5\} \vee (u.x \wedge u@\{10,11,12\}))) \vee$
$(\neg T[v] \wedge (Q[u] \vee u@\{0..5\} \vee (\neg u.x \wedge u@\{8,11,12\}))) \ )$ (I12)

Proof: Similar to the proof of (I11). □

The following invariant shows that the Mutual Exclusion requirement holds.

invariant $\neg(u@\{6\} \wedge v@\{6\})$ (I13)

Proof: We show that (I13) follows from the preceding invariants. Specifically, we use these prior invariants to prove that $(u@\{6\} \wedge v@\{6\}) \Rightarrow \mathit{false}$.

$u@\{6\} \wedge v@\{6\}$

$\Rightarrow$ $(u@\{6\} \wedge v@\{6\}) \wedge$
$((T[u] \wedge P[v]) \vee (\neg T[u] \wedge Q[v])) \wedge ((T[v] \wedge P[u]) \vee (\neg T[v] \wedge Q[u]))$
, by (I11) and (I12).

$\Rightarrow$ $((T[u] \wedge \neg T[v]) \vee (\neg T[u] \wedge T[v])) \wedge ((T[v] \wedge T[u]) \vee (\neg T[v] \wedge \neg T[u]))$
, by (I2), (I9), and (I10),
$(u@\{6\} \wedge v@\{6\})$ implies
$P[u] = T[u]$, $P[v] = \neg T[v]$,
$Q[u] = \neg T[u]$, and $Q[v] = T[v]$.

$\Rightarrow$ $(T[u] \neq T[v]) \wedge (T[u] = T[v])$, predicate calculus.

$\Rightarrow$ false , predicate calculus. □

The next invariant shows that the program does not deadlock. According to this invariant, if each process is either in its noncritical section or at one of its **await** statements, then either one of

the processes is waiting on a variable whose value is true, or both processes are in their noncritical sections.

invariant $u@\{3,7,9\} \wedge v@\{3,7,9\} \Rightarrow (u@\{7\} \wedge Q[v]) \vee (v@\{7\} \wedge Q[u]) \vee (u@\{9\} \wedge P[v]) \vee (v@\{9\} \wedge P[u]) \vee (u@\{3\} \wedge v@\{3\})$ (I14)

Proof: We show that (I14) follows from the preceding invariants.

$u@\{3,7,9\} \wedge v@\{3,7,9\}$

$\Rightarrow$ $(u@\{3\} \vee u@\{7\} \vee u@\{9\}) \wedge (v@\{3\} \vee v@\{7\} \vee v@\{9\})$
, by the definition of "@".

$\Rightarrow$ $(u@\{3\} \wedge v@\{3\}) \vee (u@\{3\} \wedge (v@\{7\} \vee v@\{9\})) \vee (v@\{3\} \wedge (u@\{7\} \vee u@\{9\})) \vee$
$((u@\{7\} \vee u@\{9\}) \wedge (v@\{7\} \vee v@\{9\}))$, predicate calculus.

$\Rightarrow$ $(u@\{3\} \wedge v@\{3\}) \vee$
$(P[u] \wedge Q[u] \wedge (v@\{7\} \vee v@\{9\})) \vee (P[v] \wedge Q[v] \wedge (u@\{7\} \vee u@\{9\})) \vee$
$((u@\{7\} \vee u@\{9\}) \wedge (v@\{7\} \vee v@\{9\}))$, by (I7) and (I8).

$\Rightarrow$ $(u@\{3\} \wedge v@\{3\}) \vee$
$(P[u] \wedge Q[u] \wedge (v@\{7\} \vee v@\{9\})) \vee (P[v] \wedge Q[v] \wedge (u@\{7\} \vee u@\{9\})) \vee$
$(((u@\{7\} \wedge \neg P[u] \wedge Q[u]) \vee (u@\{9\} \wedge P[u] \wedge \neg Q[u])) \wedge$
$((v@\{7\} \wedge P[v] \wedge \neg Q[v]) \vee (v@\{9\} \wedge \neg P[v] \wedge Q[v])))$, by (I0), (I1), and (I9),
$u@\{7\}$ implies $\neg P[u] \wedge Q[u]$ and
$u@\{9\}$ implies $P[u] \wedge \neg Q[u]$;
by (I0), (I1), and (I10),
$v@\{7\}$ implies $P[v] \wedge \neg Q[v]$ and
$v@\{9\}$ implies $\neg P[v] \wedge Q[v]$.

$\Rightarrow$ $(u@\{3\} \wedge v@\{3\}) \vee$
$(P[u] \wedge Q[u] \wedge v@\{7\}) \vee (P[u] \wedge Q[u] \wedge v@\{9\}) \vee$
$(P[v] \wedge Q[v] \wedge u@\{7\}) \vee (P[v] \wedge Q[v] \wedge u@\{9\}) \vee$
$(u@\{7\} \wedge \neg P[u] \wedge Q[u] \wedge v@\{7\} \wedge P[v] \wedge \neg Q[v]) \vee$
$(u@\{7\} \wedge \neg P[u] \wedge Q[u] \wedge v@\{9\} \wedge \neg P[v] \wedge Q[v]) \vee$
$(u@\{9\} \wedge P[u] \wedge \neg Q[u] \wedge v@\{7\} \wedge P[v] \wedge \neg Q[v]) \vee$
$(u@\{9\} \wedge P[u] \wedge \neg Q[u] \wedge v@\{9\} \wedge \neg P[v] \wedge Q[v])$, by rewriting previous assertion as a disjunction of conjunctions.

$\Rightarrow$ $(u@\{7\} \wedge Q[v]) \vee (v@\{7\} \wedge Q[u]) \vee (u@\{9\} \wedge P[v]) \vee (v@\{9\} \wedge P[u]) \vee$
$(u@\{3\} \wedge v@\{3\})$, each disjunct of previous assertion implies some disjunct of this assertion. □

We now establish the Progress requirement.

$u@\{9\} \mapsto \neg u@\{9\}$ (L0)

Proof: Let $v.pc$ denote the program counter of process v; i.e., $v.pc = k$ iff $v@\{k\}$ holds. We define a well-founded ranking R as follows.

$$R \equiv \begin{cases} 0 & \text{if } \neg u@\{9\} \\ v.pc + 1 & \text{if } u@\{9\} \ \wedge \ \neg v.x \ \wedge \ v@\{7..12\} \\ v.pc + 14 & \text{if } u@\{9\} \ \wedge \ (v.x \ \vee \ \neg v@\{7..12\}) \end{cases}$$

By definition, R is always nonnegative, $R = 0 \ \Rightarrow \ \neg u@\{9\}$, and $R > 0 \ \Rightarrow \ u@\{9\}$. Therefore, to establish that (L0) holds, it suffices to prove that if $R > 0$, then (i) there exists some enabled statement other than $3.v$ (note that process v may halt in its noncritical section), and (ii) each statement execution decreases the value of R. Requirement (i) follows from (I14) and the program text. We now show that requirement (ii) holds.

First, note that if $R > 0$, then $u@\{9\}$ holds; this implies that $9.u$ is the only statement of process u that can possibly be enabled. Each execution of this statement decreases the value of R from a positive value to 0.

In the remainder of the proof, we consider the statements of process v. We begin by noting that each statement of process v other than $0.v$, $7.v$, and $9.v$ decrements $v.pc$ by at least 1 and leaves $v.x$ and $v@\{7..12\}$ unchanged. It follows, then, that if any of these statements is executed when $R > 0$, then R is decremented by at least 1.

Next, we dispose of statement $7.v$. By the definition of R, $R > 0$ implies that $u@\{9\}$ holds. By (I4), this implies that $\neg Q[u]$ holds. By the program text, this implies that $7.v$ is not enabled.

Now, consider statement $0.v$. We prove that $R > 0 \ \wedge \ R = r \ \wedge \ \mathit{Enabled}(0.v) \ \Rightarrow \ wp(0.v, \ R = r - 1)$. First, consider the following derivation.

$R > 0 \ \wedge \ R = r \ \wedge \ \mathit{Enabled}(0.v)$

$\Rightarrow \ u@\{9\} \wedge \ R = r \ \wedge \ v@\{0\}$, $R > 0$ implies $u@\{9\}$; $\mathit{Enabled}(0.v)$ implies $v@\{0\}$.

$\Rightarrow \ u@\{9\} \wedge \ R = r \ \wedge \ v@\{0\} \ \wedge \ v.pc = 0$, by the definition of $v.pc$.

$\Rightarrow \ u@\{9\} \wedge \ R = r \ \wedge \ r = 14$, by the definition of R, preceding assertion implies $R = v.pc + 14$.

$\Rightarrow \ u@\{9\} \wedge \ R = r \ \wedge \ r = 14 \ \wedge \ T[u]$, by (I0) and (I2).

$\Rightarrow \ wp(0.v, \ u@\{9\} \wedge \ r = 14 \ \wedge \ \neg v.x \ \wedge \ v@\{12\})$, by the program text.

Now, consider the following derivation.

$u@\{9\} \wedge \ r = 14 \ \wedge \ \neg v.x \ \wedge \ v@\{12\}$

$\Rightarrow \ u@\{9\} \wedge \ r = 14 \ \wedge \ \neg v.x \ \wedge \ v@\{12\} \ \wedge \ v.pc = 12$, by the definition of $v.pc$.

$\Rightarrow\ R = r - 1$, by the definition of R, preceding assertion implies $R = v.pc + 1$.

By the above two derivations and the monotonicity of wp, $R > 0\ \wedge\ R = r\ \wedge\ \mathit{Enabled}(0.v)\ \Rightarrow\ wp(0.v,\ R = r - 1)$.

Finally, consider statement $9.v$. We prove that $R > 0\ \wedge\ R = r\ \wedge\ \mathit{Enabled}(9.v)\ \Rightarrow\ wp(9.v,\ R = r - 3)$. First, consider the following derivation.

$R > 0\ \wedge\ R = r\ \wedge\ \mathit{Enabled}(9.v)$

$\Rightarrow\ u@\{9\} \wedge\ R = r\ \wedge\ v@\{9\}\ \wedge\ P[u]$, $R > 0$ implies that $u@\{9\}$; $\mathit{Enabled}(9.v)$ implies $v@\{9\}\ \wedge\ P[u]$.

$\Rightarrow\ u@\{9\} \wedge\ R = r\ \wedge\ v@\{9\}\ \wedge\ P[u]\ \wedge\ v.pc = 9$, by the definition of $v.pc$.

$\Rightarrow\ u@\{9\} \wedge\ R = r\ \wedge\ v@\{9\}\ \wedge\ P[u]\ \wedge\ v.pc = 9\ \wedge\ v.x$, by (I0).

$\Rightarrow\ u@\{9\} \wedge\ R = r\ \wedge\ r = 23\ \wedge\ P[u]$, by the definition of R, preceding assertion implies $R = v.pc + 14$.

$\Rightarrow\ wp(9.v,\ u@\{9\} \wedge\ r = 23\ \wedge\ v@\{6\})$, by the program text.

Now, observe the following.

$u@\{9\} \wedge\ r = 23\ \wedge\ v@\{6\}$

$\Rightarrow\ u@\{9\} \wedge\ r = 23\ \wedge\ v@\{6\}\ \wedge\ v.pc = 6$, by the definition of $v.pc$.

$\Rightarrow\ R = r - 3$, by the definition of R, preceding assertion implies $R = v.pc + 14$.

By the above two derivations and the monotonicity of wp, $R > 0\ \wedge\ R = r\ \wedge\ \mathit{Enabled}(9.v)\ \Rightarrow\ wp(9.v,\ R = r - 3)$. This completes the proof. □

$u@\{7\} \mapsto \neg u@\{7\}$ (L1)

Proof: Similar to the proof of (L0). □

$v@\{9\} \mapsto \neg v@\{9\}$ (L2)

Proof: Similar to the proof of (L0). □

$v@\{7\} \mapsto \neg v@\{7\}$ (L3)

Proof: Similar to the proof of (L0). □

$i@\{2\} \mapsto i@\{6\}$ (L4)

Proof: By the program text, our fairness assumption, and the transitivity of leads-to, $i@\{2\} \mapsto i@\{7,9\}$. By (L0), (L1), (L2), (L3), and the program text, $i@\{7,9\} \mapsto i@\{6\}$. Hence, by transitivity of leads-to, $i@\{2\} \mapsto i@\{6\}$. □

5 Allowing Nonatomic Variables

Until now, we have limited our attention to programs comprised of shared variables that are accessed atomically, i.e., by statements that cause a single state transition when executed. It turns out, however, that our solution to the mutual exclusion problem is correct even if shared variables are accessed nonatomically, i.e., by statements that induce several state transitions when executed. We show this below after specifying more precisely the notion of a nonatomic shared variable.

In concurrent programs with nonatomic shared variables, it is possible for two or more statements that access such a variable to be executed concurrently; such is the case when a state transition induced by one such statement occurs between the first and last state transitions induced by another such statement. To be able to reason about such programs, it is necessary to specify the effect of such a concurrent execution. Observe that in the program of Figure 2, each shared variable is written by one of the processes and is either read by the other process ($T[u]$ and $T[v]$) or is awaited on by the other process ($P[u]$, $Q[u]$, $P[v]$, and $Q[v]$). Thus, in defining the effect of concurrent statement executions, there are only two cases of interest to us: a read of a shared variable that is concurrent with a write to that variable, and an await on a shared variable that is concurrent with a write to that variable.

To keep the ensuing discussion as simple as possible, we follow the approach taken by Anderson and Gouda in [1] and assume that each statement that reads or awaits on a shared variable is atomic and that each statement that writes a shared variable is nonatomic; the semantics of each nonatomic write is specified by *defining* each such write to be equivalent to a particular sequence of atomic statements. In Figure 2, each statement that writes a shared variable is of the form "$k:\ X := y$", where k is a statement label of some process i, X is a shared boolean variable, and y is either a private variable of process i or a constant. If such a statement is executed nonatomically, then we define its semantics by the following program fragment.

k: **do** $\langle true \ \rightarrow \ X := \neg X \rangle$
 $[\!]$ $\langle true \ \rightarrow \ X := y;$ **exit** $\rangle$
 od

This program fragment consists of a **do**-loop comprised of two atomic statements; the scope of each statement is indicated by enclosing it within angle brackets. When the program counter of process i equals k, both statements are enabled. Executing the first statement inverts the value of X and does not change the program counter of process i. Executing the second statement causes the value of y to be assigned to X and modifies the program counter of process i so as to terminate the loop. Note that, while the program counter of process i equals k, the second atomic statement of the loop is continuously enabled. Thus, in any fair history in which this loop is executed, the

second statement of the loop is eventually executed. Thus, in any fair history, each nonatomic write eventually terminates.

Observe that our notion of a nonatomic shared variable is consistent with Lamport's definition of a "safe" shared register [6]. In particular, if a read of a shared variable is executed concurrently with a write to that variable, then the read may obtain any value from the value domain of the variable. Also, the statement "**await** X", where X is a boolean shared variable, may be executed concurrently with a write to X, even if that write "changes" the value of X from false to false. As expected, if a read of a shared variable is not concurrent with any write to that variable, then the read obtains the value most recently written to the variable (or the variable's initial value if no preceding write exists). Likewise, "**await** X" may be enabled for execution at a state that is not concurrent with a write to X only if X is true at that state.

To see that our solution to the mutual exclusion problem is correct under the above definition of nonatomic statement execution, consider Figure 3. This figure shows the program of Figure 2 with each nonatomic write replaced by its semantically equivalent program fragment (variable declarations have been omitted for brevity). In the subsequent discussion, we refer to the program of Figure 2 as the "atomic" program, and the program of Figure 3 as the "nonatomic" program. To show that the nonatomic program is a correct solution to the mutual exclusion problem, we must show that it satisfies the Mutual Exclusion and Progress requirements.

Verifying that the Mutual Exclusion requirement holds for the nonatomic program is straightforward. As the reader can verify, we have carefully defined each invariant given in Section 4 so that it holds not only for the atomic program, but also for the nonatomic program. In particular, invariant (I13) holds, implying that the Mutual Exclusion requirement holds for the nonatomic program.

As for the Progress requirement, it should be clear that if either process of the nonatomic program leaves its noncritical section, then that process eventually reaches one of its **await** statements. (Recall that in any fair history each nonatomic write eventually terminates.) Thus, to establish the Progress requirement, it suffices to show that neither process can wait forever at one of its **await** statements. This amounts to proving that assertions (L0), (L1), (L2), and (L3) of Section 4 hold. Because a waiting process does not modify any shared variables, these assertions can be proved for the nonatomic program in much the same manner as was originally done in Section 4 for the atomic program. To see this, consider the original proof of (L0). The crux of this proof involved showing that if $u@\{9\}$ holds (i.e., if process u is at its "**await** $P[v]$" statement), then each statement execution decreases the value of the well-founded ranking R. To show that (L0) holds for the nonatomic program in Figure 3, a slightly modified version of this argument can be applied. In particular, note that R's value depends only on the value of $u.pc$, $v.pc$, and $v.x$. Because the first atomic statement of each loop that simulates a nonatomic write in process v leaves each of these variables unchanged, each execution of such a statement leaves R unchanged. Nonetheless, by reasoning as in the original proof, it can be shown that the second atomic statement of each such loop decreases the value of R. By our fairness assumption and the original proof of (L0), it follows that while $u@\{9\}$ continues to hold, the value of R is never increased by any statement execution and is *eventually* decreased by some statement execution. This is sufficient for establishing that (L0) holds for the nonatomic program.

process u

do $true \rightarrow$

3: Noncritical Section;

2: **do** $\langle true \rightarrow P[u] := \neg P[u] \rangle$
$[\!]$ $\langle true \rightarrow P[u] := false;$ **exit** $\rangle$
od;

1: **do** $\langle true \rightarrow Q[u] := \neg Q[u] \rangle$
$[\!]$ $\langle true \rightarrow Q[u] := false;$ **exit** $\rangle$
od;

0: $u.x := T[v]$;

12: **do** $\langle true \rightarrow T[u] := \neg T[u] \rangle$
$[\!]$ $\langle true \rightarrow T[u] := u.x;$ **exit** $\rangle$
od;

11: **if** $u.x \rightarrow$

10: **do** $\langle true \rightarrow P[u] := \neg P[u] \rangle$
$[\!]$ $\langle true \rightarrow P[u] := true;$ **exit** $\rangle$
od;

9: **await** $P[v]$

$[\!]$ $\neg u.x \rightarrow$

8: **do** $\langle true \rightarrow Q[u] := \neg Q[u] \rangle$
$[\!]$ $\langle true \rightarrow Q[u] := true;$ **exit** $\rangle$
od;

7: **await** $Q[v]$

fi;

6: Critical Section;

5: **do** $\langle true \rightarrow P[u] := \neg P[u] \rangle$
$[\!]$ $\langle true \rightarrow P[u] := true;$ **exit** $\rangle$
od;

4: **do** $\langle true \rightarrow Q[u] := \neg Q[u] \rangle$
$[\!]$ $\langle true \rightarrow Q[u] := true;$ **exit** $\rangle$
od

od

process v

do $true \rightarrow$

3: Noncritical Section;

2: **do** $\langle true \rightarrow P[v] := \neg P[v] \rangle$
$[\!]$ $\langle true \rightarrow P[v] := false;$ **exit** $\rangle$
od;

1: **do** $\langle true \rightarrow Q[v] := \neg Q[v] \rangle$
$[\!]$ $\langle true \rightarrow Q[v] := false;$ **exit** $\rangle$
od;

0: $v.x := \neg T[u]$;

12: **do** $\langle true \rightarrow T[v] := \neg T[v] \rangle$
$[\!]$ $\langle true \rightarrow T[v] := v.x;$ **exit** $\rangle$
od;

11: **if** $v.x \rightarrow$

10: **do** $\langle true \rightarrow Q[v] := \neg Q[v] \rangle$
$[\!]$ $\langle true \rightarrow Q[v] := true;$ **exit** $\rangle$
od;

9: **await** $P[u]$

$[\!]$ $\neg v.x \rightarrow$

8: **do** $\langle true \rightarrow P[v] := \neg P[v] \rangle$
$[\!]$ $\langle true \rightarrow P[v] := true;$ **exit** $\rangle$
od;

7: **await** $Q[u]$

fi;

6: Critical Section;

5: **do** $\langle true \rightarrow P[v] := \neg P[v] \rangle$
$[\!]$ $\langle true \rightarrow P[v] := true;$ **exit** $\rangle$
od;

4: **do** $\langle true \rightarrow Q[v] := \neg Q[v] \rangle$
$[\!]$ $\langle true \rightarrow Q[v] := true;$ **exit** $\rangle$
od

od

Figure 3: Two-process mutual exclusion algorithm with nonatomic shared variables.

6 Concluding Remarks

We have presented a fine-grained program that solves the mutual exclusion problem, along with a formal proof of its correctness. We have also shown that if shared variables are accessed nonatomically, then the program's correctness is unaffected. The processes of our solution wait on one another solely by executing statements of the form "**await** X", where X is a shared, single-reader, single-writer boolean variable. Such a statement can be implemented either by context switching or by means of a lower level busy-waiting loop such as the following.

$done := X$;
do $\neg done \rightarrow done := X$ **od**

As pointed out by Mellor-Crummey and Scott in [9], many of today's multiprocessing systems permit shared variables to be locally accessible; such is the case if coherent caching schemes are employed, or if shared variables can be allocated in a local portion of distributed shared memory. If the shared variable X in the busy-waiting loop above is locally accessible, then this loop can be executed with relatively little overhead, as each read of X can be accomplished without tying up the interconnection network between processors and shared memory. This stands in sharp contrast to the algorithms of [2, 7, 11], where busy-waiting loops are employed in which shared variables are repeatedly tested *and* updated. The performance studies of [9] suggest that such busy-waiting may result in an unacceptable degree of memory and interconnect contention.

Acknowledgement: I would like to thank Mohamed Gouda, Leslie Lamport, Udaya Shankar, and Jae-Heon Yang for their comments on earlier drafts of this paper.

References

[1] J. Anderson and M. Gouda, "Atomic Semantics of Nonatomic Programs", *Information Processing Letters*, Vol. 28, June 24, 1988, pp. 99-103.

[2] J. Burns and N. Lynch, "Bounds on Shared Memory for Mutual Exclusion", to appear in *Information and Computation*. Originally appeared as "Mutual Exclusion Using Indivisible Reads and Writes", *Proceedings of the 18th Allerton Conference on Communication, Control, and Computing*, October 1980, pp. 833-842.

[3] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 9, 1965, pp. 569.

[4] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[5] J. Kessels, "Arbitration Without Common Modifiable Variables", *Acta Informatica*, Vol. 17, 1982, pp. 135-141.

[6] L. Lamport, "On Interprocess Communication, Parts I and II", *Distributed Computing*, Vol. 1, 1986, pp. 77-101.

[7] L. Lamport, "The Mutual Exclusion Problem II – Statement and Solutions", *Journal of the ACM*, Vol. 33, No. 2, April 1986, pp. 327-348.

[8] L. Lamport, personal communication.

[9] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February 1991, pp. 21-65.

[10] G. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, No. 3, June 1981, pp. 115-116.

[11] G. Peterson, "A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 1, January 1983, pp. 56-65.