When Does a Correct Mutual Exclusion Algorithm Guarantee Mutual Exclusion?

Leslie Lamport, Sharon Perl, William Weihl

23 November 1998
revised 23 Mar 2000

Dijkstra introduced mutual exclusion for an N-process system as the requirement "that at any moment only one of these N cyclic processes is in its critical section" [1]. This requirement, which we call *true* mutual exclusion, is still the standard definition of mutual exclusion.

Mutual exclusion algorithms for shared-memory multiprocessors do not guarantee true mutual exclusion. We give a simple example that shows why a mutual exclusion algorithm can permit two critical sections to be executing at the same time. We prove that a shared-memory mutual exclusion algorithm does provide true mutual exclusion if a processor does not know in advance what memory operations will follow the critical section. Modern processors can look ahead at operations that follow a critical section. If true mutual exclusion is needed, this look-ahead must be inhibited. However, as we will explain, true mutual exclusion is seldom useful in practice.

One of the strongest and most common assumptions made about multiprocessor shared memory is sequential consistency. Sequential consistency is defined as follows:

> [T]he result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [2]

Sequential consistency says nothing about the order in which the operations are actually executed. A mutual exclusion algorithm executed on sequentially consistent memory guarantees only what we call *virtual* mutual exclusion: memory operations act as if all those in one critical section were performed either before or after all those in another. Virtual mutual exclusion says nothing about when the operations are really performed.

As an example, consider the following basic mutual exclusion protocol that is used by several standard algorithms.

```
Processor A: a := 1; if b = 0 then critical section; a := 0 fi
Processor B: b := 1; if a = 0 then critical section; b := 0 fi
```

Suppose neither of the critical sections accesses a, b, or any variable accessed by the other. A multiprocessor could execute processor A's assignment $a := 1$, its **if** test, and its assignment $a := 0$, and then arbitrarily interleave the execution of its critical section with the execution of processor B's protocol. Because the memory operations of processor A's critical section commute with all the memory operations of processor B's protocol, the resulting execution is equivalent to executing the two protocols sequentially (in either order). Thus, the definition of sequential consistency, which says nothing about the order in which operations are actually executed, is satisfied. Hence virtual but not real mutual exclusion is satisfied.

Executing a mutual exclusion algorithm without satisfying real mutual exclusion requires aggressive reordering of instructions based on knowledge of more than one processor's instruction stream. It probably cannot occur on existing multiprocessors. However, processors and memory systems are becoming increasingly sophisticated, and this kind of interprocessor optimization may be possible in future systems that have multiple processors on the same chip.

If the critical sections do nothing but access memory, virtual mutual exclusion is good enough. It doesn't matter when they really are executed. However, the critical sections may contain I/O operations to external devices. If a mutual exclusion algorithm is being used to ensure that I/O operations by different processors are not issued to the device at the same time, then it must implement true mutual exclusion.

In many computers, I/O is performed by memory operations to certain addresses. If an external device is accessed by operations to only a single address, then virtual mutual exclusion disallows the interleaving of I/O operations from different critical sections to the same device. (The only way a memory system can make I/O operations to the same address act as if they occur in a certain order is to execute them in that order.) But, operations to two different addresses might affect the same device. Moreover, mutual exclusion might be used to prevent certain sequences of operations to different devices. In these cases, true mutual exclusion is needed.

We will show that true mutual exclusion is implied by virtual mutual exclusion if a processor cannot look ahead to instructions past its critical section. Let a processor be a device that takes as input a sequence of issued instructions that it executes, perhaps in a different order. An algorithm is then a rule that determines the sequence of instructions to be issued. We assume conditional instructions with the usual semantics, so executing (the sequence of instructions issued by) the statement

if $x = 0$ **then** S **fi**

causes the instructions of S to be executed only if executing the read of x obtains the value 0. We assume that the result of executing an instruction is independent of instructions that have not yet been issued. For example, issuing an instruction from a particular address does not imply that the next instruction issued from that address will be the same. We posit a statement (a finite sequence of instructions) *Lull* such that no instructions will be issued to a processor after the *Lull* statement until the processor has finished executing all instructions issued to it before the *Lull* statement—that is, until it has issued all reads and writes preceding the *Lull*, and has received the results of all the reads. (The processor does not have to wait for writes to complete.) More precisely, we assume:

A1. The result of executing an instruction cannot depend on what instructions are issued after any *Lull* that has not yet been executed.

We prove the following result:

> If two processors execute critical sections each containing a *Lull* statement, then critical-section instructions that precede the *Lull* statements cannot be concurrently executed by the two processors.

The proof requires five additional assumptions. First, we make three assumptions about the shared memory:

A2. A memory cell can assume at least two values, which we take to be 0 and 1.

A3. If memory cell x is initially 0 and no write to x is ever executed, then every read of x obtains the value 0.

A4. If memory cell x is initially 0 and a processor executes $x := 1$, then any other processor that keeps reading the cell will eventually find it equal to 1. (The operation $x := 1$ could include some form of "cache flush" or memory synchronization operation.)

A memory system that didn't satisfy these assumptions would not be very useful. We make the following assumption about the mutual exclusion algorithm. It rules out uninteresting algorithms that have to access all of memory.

A5. The algorithm remains correct if three memory cells, not currently used by any processor, are added to the system.

We have defined virtual mutual exclusion for sequentially consistent memories. However, our result applies to a much larger class of memories. Instead of trying to find the appropriate generalization, we make the following assumption about the interaction of mutual exclusion and the memory's semantics.

A6. If a memory cell x is initially 0, and the critical sections of two processors both read x and then execute $x := 1$, then it is impossible for both reads to obtain the value 0.

This assumption holds for a virtual mutual exclusion algorithm executed on sequentially consistent memory, since the read by the processor whose critical section acts as if it were executed last must see the value written by the other processor. For other memory models, mutual exclusion wouldn't be of much use if it didn't satisfy this condition.

We now prove our result by contradiction. We assume that processors A and B are both in their critical sections, executing instructions that precede the critical sections' *Lull* statements, and we obtain a contradiction. Since neither processor has executed its *Lull*, assumption A1 implies that we would obtain the same execution if the following instruction sequences are issued after those *Lull*s, where x, y, and z are memory cells, initially equal to 0, that are not used by either processor:

```
Processor $A$:  **if** $x = 0$ **then** $y := 1$; **while** $z = 0$ **do od fi**; *Lull*
Processor $B$:  **if** $x = 0$ **then** $z := 1$; **while** $y = 0$ **do od fi**; *Lull*
```

We suppose that A and B issue no other writes to y or z, and that any write to x is executed after these *Lull* statements.

At least one of the processors must execute its read of x before either processor has issued any instructions past its subsequent *Lull* statement. Without loss of generality, we may assume it is processor A. We now reason as follows.

1. Processor A's read of x obtains the value 0.
   PROOF: By the assumption that A reads x before either processor executes the following *Lull*, assumption A1 implies that the result of the read cannot depend on what instructions follow those *Lull*s. If neither A nor B contains any write of x, then assumption A3 implies that the read obtains 0. Hence, it must obtain that value regardless of whether there is a subsequent write to x.
2. Processor B's read of x obtains the value 0.
   PROOF: Suppose B's read of x obtains a value other than 0. Because we are assuming that there are no writes to z except the one in B's code above, A3 implies that processor A cannot exit its **while** loop. Hence, it never executes the following *Lull*. Since neither *Lull* has been executed when B reads x, A1 implies that the result of the read cannot depend on the instructions that follow the *Lull*s. Since there might be no writes to x following the *Lull*s, A3 implies that the read could not have obtained any value other than 0.
3. The reads of x by A and B cannot both obtain the value 0.
   PROOF: Suppose both A and B both read x equal to 0. Assumptions A4 and A5 then imply that both processors will exit their **while** loops. We could then violate assumption A6 by following the *Lull* in each processor's code above with $x := 1$ and then the instructions to exit its critical section. Hence, the reads of x by A and B cannot both obtain 0.

Steps 1–3 provide the required contradiction.

We have shown that a shared-memory mutual exclusion algorithm guarantees true mutual exclusion—two processors are never in their critical sections at the same time—if each critical section is ended with a *Lull* statement that inhibits look-ahead. Implementation of the *Lull* statement will depend on the particular processor and memory-system architecture.

True mutual exclusion can be used to implement mutually exclusive access to external devices if and only if there is a bound on the time needed for an operation generated by a processor to reach and be completed by the device. With such a bound, mutually exclusive access is achieved by having the processor remain in its critical section until the operation it generated has been completed by the device. However, this kind of real-time bound is seldom known. Without it, mutually exclusive access requires a completion signal from the device. A processor enters its critical section, issues the operation to the device, and then remains in the critical section until it receives the completion signal. It is easy to see that, since the processor has no way of predicting that the completion signal will actually arrive, this guarantees mutually exclusive access to the device even when the critical-section algorithm implements only virtual mutual exclusion.

Acknowledgment

Mike Burrows pointed out to us that true mutual exclusion does not help achieve mutually exclusive access to an external device without a bound on how long it takes to access the device.

References

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

[2] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, July 1997.