How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor

Leslie Lamport[1]
Digital Equipment Corporation

February 14, 1993
Minor revisions January 18, 1996 and September 14, 1996

Abstract

A multiprocess program executing on a modern multiprocessor must issue explicit commands to synchronize memory accesses. A method is proposed for deriving the necessary commands from a correctness proof of the underlying algorithm in a formalism based on temporal relations among operation executions.

index terms concurrency, memory consistency, multiprocessor, synchronization, verification

[1]Author's current address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301

1 The Problem

Accessing a single memory location in a multiprocessor is traditionally assumed to be atomic. Such atomicity is a fiction; a memory access consists of a number of hardware actions, and different accesses may be executed concurrently. Early multiprocessors maintained this fiction, but more modern ones usually do not. Instead, they provide special commands with which processes themselves can synchronize memory accesses. The programmer must determine, for each particular computer, what synchronization commands are needed to make his program correct.

One proposed method for achieving the necessary synchronization is with a constrained style of programming specific to a particular type of multiprocessor architecture [7, 8]. Another method is to reason about the program in a mathematical abstraction of the architecture [5]. We take a different approach and derive the synchronization commands from a proof of correctness of the algorithm.

The commonly used formalisms for describing multiprocess programs assume atomicity of memory accesses. When an assumption is built into a formalism, it is difficult to discover from a proof where the assumption is actually needed. Proofs based on these formalisms, including invariance proofs [4, 16] and temporal-logic proofs [17], therefore seem incapable of yielding the necessary synchronization requirements. We derive these requirements from proofs based on a little-used formalism that makes no atomicity assumptions [11, 12, 14]. This proof method is quite general and has been applied to a number of algorithms. The method of extracting synchronization commands from a proof is described by an example—a simple mutual exclusion algorithm. It can be applied to the proof of any algorithm.

Most programs are written in higher-level languages that provide abstractions, such as locks for shared data, that free the programmer from concerns about the memory architecture. The compiler generates synchronization commands to implement the abstractions. However, some algorithms—especially within the operating system—require more efficient implementations than can be achieved with high-level language abstractions. It is to these algorithms, as well as to algorithms for implementing the higher-level abstractions, that our method is directed.

2 The Formalism

An execution of a program is represented by a collection of *operation executions* with the two relations $\longrightarrow$ (read *precedes*) and $\dashrightarrow$ (read *can affect*). An operation execution can be interpreted as a nonempty set of events, where the relations $\longrightarrow$ and $\dashrightarrow$ have the following meanings.

$A \longrightarrow B$: every event in A precedes every event in B.

$A \dashrightarrow B$: some event in A precedes some event in B.

However, this interpretation serves only to aid our understanding. Formally, we just assume that the following axioms hold, for any operation executions A, B, C, and D.

A1. $\longrightarrow$ is transitive ($A \longrightarrow B \longrightarrow C$ implies $A \longrightarrow C$) and irreflexive ($A \not\longrightarrow A$).

A2. $A \longrightarrow B$ implies $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$ implies $A \dashrightarrow C$.

A4. $A \longrightarrow B \dashrightarrow C \longrightarrow D$ implies $A \longrightarrow D$.

A5. For any A there are only a finite number of B such that $A \not\longrightarrow B$.

The last axiom essentially asserts that all operation executions terminate; nonterminating operations satisfy a different axiom that is not relevant here. Axiom A5 is useful only for proving liveness properties; safety properties are proved with Axioms A1–A4. properties. Anger [3] and Abraham and Ben-David [1] introduced the additional axiom

A6. $A \dashrightarrow B \longrightarrow C \dashrightarrow D$ implies $A \dashrightarrow D$.

and showed that A1–A6 form a complete axiom system for the interpretation based on operation executions as sets of events.

Axioms A1–A6 are independent of what the operation executions do. Reasoning about a multiprocess program requires additional axioms to capture the semantics of its operations. The appropriate axioms for read and write operations will depend on the nature of the memory system.

The only assumptions we make about operation executions are axioms A1–A5 and axioms about read and write operations. We do not assume that $\longrightarrow$ and $\dashrightarrow$ are the relations obtained by interpreting an operation execution as the set of all its events. For example, sequential consistency [10] is equivalent to the condition that $\longrightarrow$ is a total ordering on the set of operation executions—a condition that can be satisfied even though the events comprising different operation executions are actually concurrent.

This formalism was developed in an attempt to provide elegant proofs of concurrent algorithms—proofs that replace conventional behavioral arguments with axiomatic reasoning in terms of the two relations $\longrightarrow$ and $\dashrightarrow$. Although the simplicity of such proofs has been questioned [6], they do tend to capture the essence of why an algorithm works.

3 An Example

3.1 An Algorithm and its Proof

Figure 1 shows process i of a simple N-process mutual exclusion algorithm [13]. We prove that the algorithm guarantees mutual exclusion (two processes are never concurrently in their critical sections). The algorithm is also deadlock-free (some critical section is eventually executed unless all processes halt in their noncritical sections), but we do not consider this liveness property. Starvation of individual processes is possible.

The algorithm uses a standard protocol to achieve mutual exclusion. Before entering its critical section, each process i must first set x_i true and then find x_j false, for all other processes j. Mutual exclusion is guaranteed because, when process i finds x_j false, process j cannot enter its critical section until it sets x_j true and finds x_i false, which is impossible until i has exited the critical section and reset x_i. The proof of correctness formalizes this argument.

To prove mutual exclusion, we first name the following operation executions that occur during the n^{th} iteration of process i's **repeat** loop.

L_i^n The last execution of statement l prior to entering the critical section. This operation execution sets x_i to *true*.

$R_{i,j}^n$ The last read of x_j before entering the critical section. This read obtains the value *false*.

CS_i^n The execution of the critical section.

X_i^n The write to x_i after exiting the critical section. It writes the value *false*.

```
repeat forever
   noncritical section;
l: x_i := true;
   for j := 1 until i − 1
      do  if x_j then x_i := false;
                     while x_j do od;
                     goto l              fi  od;
   for j := i + 1 until N do  while x_j do od  od;
   critical section;
   x_i := false
end repeat
```

Figure 1: Process i of an N-process mutual-exclusion algorithm.

Mutual exclusion asserts that CS_i^n and CS_j^m are not concurrent, for all m and n, if $i \neq j$.[1] Two operations are nonconcurrent if one precedes ($\longrightarrow$) the other. Thus, mutual exclusion is implied by the assertion that, for all m and n, either $CS_i^n \longrightarrow CS_j^m$ or $CS_j^m \longrightarrow CS_i^n$, if $i \neq j$.

The proof of mutual exclusion, using axioms A1–A4 and assumptions B1–B4 below, appears in Figure 2. It is essentially the same proof as in [13], except that the properties required of the memory system have been isolated and named B1–B4. (In [13], these properties are deduced from other assumptions.)

B1–B4 are as follows, where universal quantification over n, m, i, and j is assumed. B4 is discussed below.

B1. $L_i^n \longrightarrow R_{i,j}^n$

B2. $R_{i,j}^n \longrightarrow CS_i^n$

B3. $CS_i^n \longrightarrow X_i^n$

B4. If $R_{i,j}^n \not\dashrightarrow L_j^m$ then X_j^m exists and $X_j^m \dashrightarrow R_{i,j}^n$.

Although B4 cannot be proved without additional assumptions, it merits an informal justification. The hypothesis, $R_{i,j}^n \not\dashrightarrow L_j^m$, asserts that process i's read $R_{i,j}^n$ of x_j occurred too late for any of its events to have preceded any

[1] Except where indicated otherwise, all assertions have as an unstated hypothesis the assumption that the operation executions they mention actually occur. For example, the theorem in Figure 2 has the hypothesis that CS_i^n and CS_j^m occur.

Theorem For all m, n, i, and j such that $i \neq j$, either $CS_i^n \longrightarrow CS_j^m$ or $CS_j^m \longrightarrow CS_i^n$.

Case A: $R_{i,j}^n \dashrightarrow L_j^m$.

1. $L_i^n \longrightarrow R_{j,i}^m$
   Proof: B1 , case assumption, B1 (applied to L_j^m and $R_{j,i}^m$), and A4.
2. $R_{j,i}^m \not\dashrightarrow L_i^n$
   Proof: 1 and A2.
3. $X_i^n \dashrightarrow R_{j,i}^m$
   Proof: 2 and B4 (applied to $R_{j,i}^m$, L_i^n, and X_i^n).
4. $CS_i^n \longrightarrow CS_j^m$
   Proof: B3, 3, B2 (applied to $R_{j,i}^m$ and CS_j^m), and A4.

Case B: $R_{i,j}^n \not\dashrightarrow L_j^m$.

1. $X_j^m \dashrightarrow R_{i,j}^n$
   Proof: Case assumption and B4.
2. $CS_j^m \longrightarrow CS_i^n$.
   Proof: B3 (applied to CS_j^m and X_j^m), 1, B2, and A4.

Figure 2: Proof of mutual exclusion for the algorithm of Figure 1.

of the events in process j's write L_j^m of x_j. It is reasonable to infer that the value obtained by the read was written by L_j^m or a later write to x_j. Since L_j^m writes *true* and $R_{i,j}^n$ is a read of *false*, $R_{i,j}^n$ must read the value written by a later write. The first write of x_j issued after L_j^m is X_j^m, so we expect $X_j^m \dashrightarrow R_{i,j}^n$ to hold.

3.2 The Implementation

Implementing the algorithm for a particular memory architecture may require synchronization commands to assure B1–B4. Most proposed memory systems satisfy the following property.

C1. All write operations to a single memory cell by any one process are observed by other processes in the order in which they were issued.

They also provide some form of synchronization command, *synch*, (for example, a "cache flush" operation) satisfying

C2. A *synch* command causes the issuing process to wait until all previously issued memory accesses have completed.

Properties C1 and C2 are rather informal. We restate them more precisely as follows.

C1′. If the value obtained by a read A issued by process i is the one written by process j, then that value is the one written by the last-issued write B in process j such that $B \dashrightarrow A$.

C2′. If operation executions A, B, and C are issued in that order by a single process, and B is a *synch*, then $A \longrightarrow C$.

Property C2′ implies that B1–B3 are guaranteed if *synch* operations are inserted in process i's code immediately after statement l (for B1), immediately before the critical section (for B2), and immediately after the critical section (for B3). Assumption B4 follows from C1′.

Now let us consider a more specialized memory architecture in which each process has its own cache, and a write operation (asynchronously) updates every copy of the memory cell that resides in the caches. In such an architecture, the following additional condition is likely to hold:

C3. A read of a memory cell that resides in the process's cache precedes ($\longrightarrow$) every operation execution issued subsequently by the same process.

If the memory system provides some way of ensuring that a memory cell is permanently resident in a process's cache, then B2 can be satisfied by keeping all the variables x_j in process i's cache. In this case, the *synch* immediately preceding the critical section is not needed.

3.3 Observations

One might think that the purpose of memory synchronization commands is to enforce orderings between commands issued by different processes. However, B1–B3 are precedence relations between operations issued by the same process. In general, one process cannot directly observe all the events in the execution of an operation by another process. Hence, when viewing a particular execution of an algorithm, the results of executing two operation executions A and D in different processes can permit the deduction only of a causality ($\dashrightarrow$) relation between A and D. Only if A and D occur in the same process can $A \longrightarrow D$ be deduced by direct observation. Otherwise, deducing $A \longrightarrow D$ requires the existence of an operation B in the same process as A and an operation C in the same process as D such that $A \longrightarrow B \dashrightarrow C \longrightarrow D$. Synchronization commands can guarantee the relations $A \longrightarrow B$ and $C \longrightarrow D$.

The example of the mutual exclusion algorithm illustrates how a set of properties sufficient to guarantee correctness can be extracted directly from a correctness proof. Implementations of the algorithm on different memory architectures can be derived from the assumptions, with no further reasoning about the algorithm. An implementation will be efficient only if the architecture provides synchronization primitives that efficiently implement the assumed properties.

4 Further Remarks

The atomicity condition traditionally assumed for multiprocess programs is sequential consistency, meaning that the program behaves as if the memory accesses of all processes were interleaved and then executed sequentially [10]. It has been proposed that, when sequential consistency is not provided by the memory system, it can be achieved by a constrained style of programming. Synchronization commands are added either explicitly by the programmer, or automatically from hints he provides. The method of [7, 8] can be applied to our simple example, if the x_i are identified by the programmer as synchronization variables. However, in general, deducing what synchronization commands are necessary requires analyzing all possible executions of the program, which is seldom feasible. Such an analysis is needed to find the precedence relations that, in the approach described here, are derived from the proof.

Deriving synchronization commands from a correctness proof guarantees correctness of the implementation. However, the set of synchronization commands will be minimal only if the proof is based on a minimal set of synchronization assumptions. The set of assumptions is minimal if a counterexample to the theorem can be found when any assumption is eliminated. In practice, unnecessary assumptions are often uncovered simply because they are not used in the proof.

Although it replaces traditional informal reasoning with a more rigorous, axiomatic style, the proof method we have used is essentially behavioral—one reasons directly about the set of operation executions. Behavioral methods do not seem to scale well, and our approach is unlikely to be practical for large, complicated algorithms. Most multiprocess programs for modern multiprocessors are best written in terms of higher-level abstractions. The method presented here can be applied to the algorithms that implement these abstractions and to those algorithms, usually in the depths of the operating system, where efficiency and correctness are crucial.

Assertional proofs are practical for more complicated algorithms. The obvious way to reason assertionally about algorithms with nonatomic memory operations is to represent a memory access by a sequence of atomic operations [2, 9]. With this approach, the memory architecture and synchronization operations are encoded in the algorithm. Therefore, a new proof is needed for each architecture, and the proofs are unlikely to help discover what synchronization operations are needed. A less obvious approach uses the predicate transformers *win* (weakest invariant) and *sin* (strongest invariant) to write assertional proofs for algorithms in which no atomic operations are assumed, requirements on the memory architecture being described by axioms [15]. Such a proof would establish the correctness of an algorithm for a large class of memory architectures. However, in this approach, all intraprocess $\longrightarrow$ relations are encoded in the algorithm, so the proofs are unlikely to help discover the very precedence relations that lead to the introduction of synchronization operations.

Acknowledgments

I wish to thank Allan Heydon, Michael Merritt, David Probst, Garrett Swart, Fred Schneider, and Chuck Thacker for their comments on earlier versions.

References

[1] Uri Abraham, Shai Ben-David, and Menachem Magidor. On global-time and inter-process communication. In M. Z. Kwiatkowska, M. W. Shields, and R.M. Thomas, editors, *Semantics for Concurrency*, pages 311–323. Springer-Verlag, Leicester, 1990.

[2] James H. Anderson and Mohamed G. Gouda. Atomic semantics of nonatomic programs. *Information Processing Letters*, 28:99–103, June 1988.

[3] Frank D. Anger. On Lamport's interprocessor communication model. *ACM Transactions on Programming Languages and Systems*, 11(3):404–417, July 1989.

[4] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[5] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 679–690, 1992.

[6] Shai Ben-David. The global time assumption and semantics for concurrent systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing*, pages 223–232. ACM Press, 1988.

[7] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the International Conference on Computer Architecture*, 1990.

[8] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Symposium on Parallel Algorithms and Architectures*, July 1991. A full version available as an AT&T Bell Laboratories technical report, May, 1991.

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

[10] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[11] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[12] Leslie Lamport. The mutual exclusion problem—part i: A theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, January 1985.

[13] Leslie Lamport. The mutual exclusion problem—part ii: Statement and solutions. *Journal of the ACM*, 32(1):327–348, January 1985.

[14] Leslie Lamport. On interprocess communication—part i: Basic formalism. *Distributed Computing*, 1:77–85, 1986.

[15] Leslie Lamport. *win* and *sin*: Predicate transformers for concurrency. *ACM Transactions on Programming Languages and Systems*, 12(3):396–428, July 1990.

[16] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–284, May 1976.

[17] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.