A New Approach to Proving the Correctness of Multiprocess Programs

LESLIE LAMPORT
SRI International

A new, nonassertional approach to proving multiprocess program correctness is described by proving the correctness of a new algorithm to solve the mutual exclusion problem. The algorithm is an improved version of the bakery algorithm. It is specified and proved correct without being decomposed into indivisible, atomic operations. This allows two different implementations for a conventional, nondistributed system. Moreover, the approach provides a sufficiently general specification of the algorithm to allow nontrivial implementations for a distributed system as well.

Key Words and Phrases: program correctness, multiprocessing, concurrent processing, mutual exclusion
CR Categories: 4.32, 5.24

1. INTRODUCTION

Even a simple multiprocess program can exhibit very complicated behavior when it is executed, and it is hopeless to try to verify its correctness by exhaustive testing. The only way to guarantee the absence of errors in a multiprocess program is with a rigorous proof of its correctness. Recently the assertional techniques used for proving sequential programs correct have been extended to multiprocess programs by Owicki [7], Keller [2], Lamport [3], and others. However, these techniques have the following three limitations:

(1) Assertional techniques developed thus far require that a program be decomposed into indivisible, atomic operations (or operations that act as if they were atomic). This has prevented a general method for the hierarchical decomposition of correctness proofs. (Although a hierarchical design methodology is outlined in [3], a rigorous correctness proof is obtained only for the final low level program.)

(2) Assertional techniques require that the correctness conditions be expressed in terms of the objects (such as program variables) used in the implementation. This is satisfactory for proving the correctness of an individual subroutine. However, for a large program such as an entire airline reservation system, one would like the correctness conditions to be stated in terms of higher level concepts.

Author's address: SRI International, 333 Ravenswood Avenue, 415 Menlo Park, CA 94025.

(3) Assertional techniques have so far been useful only for traditional nondistributed multiprocess systems, in which processes communicate via shared memory. We would like a method which is also applicable to distributed systems, in which processes communicate by sending signals to one another.

In this paper, we present a new, nonassertional approach to proving the correctness of multiprocess programs which overcomes these limitations. We use the term "approach" to avoid suggesting that we have a well developed methodology. However, the concepts presented here are not introduced casually. They have been distilled from several years of experience with various aspects of concurrent processing. Our approach is related to the work of Greif [1], but differs significantly from it because we consider nonatomic operations.

Rather than giving an abstract general discussion of our ideas, we have chosen in this paper to introduce our approach through a single example. (It is our opinion that good examples are more instructive than formal theories.) A future paper will discuss in more generality the concepts introduced here.

The example we use is an improved version of the bakery algorithm [4] for solving the mutual exclusion problem. This algorithm was chosen because it is short and easy to follow, yet quite subtle; and the proof of its correctness is not trivial. With this example, we show how an algorithm involving complex, nonatomic operations can be proved correct without first specifying how these operations are implemented. The meaning of the operations is specified in a way which allows us to give two quite different implementations of the algorithm for a nondistributed system. We also sketch how the specifications permit nontrivial implementations for a distributed system. However, a thorough discussion of distributed systems must be deferred to a future paper. We also defer discussion of why this approach leads to more natural specifications of correctness conditions.

2. THE ALGORITHM

We assume N processes, each with a *critical section.* The problem is to synchronize the processes so that the following *mutual exclusion condition* is satisfied: two different processes may not execute their critical sections at the same time. There are a number of other properties which are required of a solution, but they will not concern us here. The reader is referred to [4] for a more complete statement of the problem.

Our algorithm is a variant of the bakery algorithm of [4]. We describe the new algorithm here without attempting to give any intuitive explanation of "why it works." The reader who wants a better understanding of this algorithm should first study the original bakery algorithm.

We wish to express the algorithm in its most general form, in order to allow the widest possible choice of implementations. This requires introducing some new notation. We let ":>" mean "set to any value greater than" (just as ":=" means "set to the value equal to"). The statement

$$\textbf{for all } j \in \{1, \ldots, N\} \textbf{ do } S_j \textbf{ od}$$

means that the statements $S_1, \ldots, S_N$ are to be executed concurrently (or in any order), where S_1 is the statement obtained by substituting 1 for j in S_j, etc. We

also introduce the statement

wait until condition

as an abbreviation for

L: **if not** condition **then goto** L **fi.**

The relation ">" on ordered pairs of integers is defined by $(a, b) > (a', b')$ if either (i) $a > a'$ or (ii) $a = a'$ and $b > b'$. The remaining notation should be self-explanatory.

The global variables consist of the array $n[1:N]$ of nonnegative integers. Each $n[i]$ is initially equal to zero. The following is the algorithm for process i. (Labels are inserted for future reference.)

```
integer j;
repeat noncritical section;
  L1: n[i] :> 0;
  L2: n[i] :> maximum (n[1], ... , n[N]);
  L3: for all j ∈ {1, ... , N} do
        wait until n[j] = 0 or (n[j], j) ≥ (n[i], i) od;
      critical section;
  L4: n[i] := 0
end repeat
```

This type of description would suffice to specify a single process algorithm. However, for a multiprocess algorithm, we must also specify what kind of interaction is permitted between the concurrent executions of the different processes. For our algorithm, this requires specifying the result of concurrent accessing of the variable $n[i]$ by two different processes. This has traditionally been done by specifying that certain operations are to behave as if they were instantaneously executed indivisible atomic operations. Thus we could specify that fetching or writing the value of $n[i]$ is an atomic operation. However, this would be unacceptable for our algorithm. The value of $n[i]$ can become arbitrarily large, so it may have to be stored in several separate memory registers—especially for machines with shorter word lengths. Implementing the fetch and write operations to be atomic would then be nontrivial, so simply defining them to be atomic in our algorithm would sweep a significant implementation problem under the rug.

Note. The unboundedness of $n[i]$ has nothing to do with the nondeterministic ":>" statements, but is inherent in the bakery algorithm. It seems to be the price one must pay for the elegance and simplicity of the algorithm. The problem of finding practical bounds on the values assumed by $n[i]$ is discussed in Section 6. □

We could define each $n[i]$ to consist of an array of elementary variables, and specify the fetch and write operations in terms of atomic operations on these elementary variables. However, this would overly specify the algorithm, and would rule out other valid implementations. In Section 6, we describe two quite different ways of implementing the fetch and write operations in terms of atomic operations on elementary variables.

Our approach is to specify directly what the effect of concurrent operations on the variable $n[i]$ must be. The correctness of any particular implementation can

then be verified by showing that it meets this specification. We can state the requirements informally as follows. They are stated more precisely in Section 4.

R1. A fetch of $n[i]$ which does not overlap any write of $n[i]$ must obtain the correct value.

R2. A fetch of $n[i]$ in statement L3 (by a process $i' \neq i$) which overlaps the write of $n[i]$ in statement L2 (by process i) must obtain a value which is greater than zero and less than or equal to the value being written.

Note that a fetch which overlaps a write is allowed to return any value at all if either (a) the write is performed while executing statement L1 or L4, or (b) the fetch is performed while executing statement L2. Requirement R1 implies that concurrent fetches of $n[i]$ by different processes do not interfere with one another. (Concurrent writing of $n[i]$ by different processes is impossible, since process i is the only one which modifies $n[i]$.)

The major advantage of this algorithm over the original bakery algorithm is that process i executes only one **wait until** loop for each other process j, rather than two. However, an implementation must satisfy the additional requirement R2. (The original bakery algorithm required only R1.)

The new algorithm has all of the same properties as the original bakery algorithm; e.g. it behaves the same way in the presence of process failure, and processes enter their critical sections on a first-come, first-served basis. (The "doorway" consists of statements L1 and L2.) However, we will not bother to prove these properties. Their informal correctness proofs are essentially the same as for the original bakery algorithm. We restrict ourselves to proving the mutual exclusion condition, both because it is the most difficult property to verify, and because it requires a completely new proof.

3. OPERATIONS

We now introduce some general concepts and notation. In a future paper, we will discuss these concepts in more detail, and show how they can be applied in a wider variety of situations. Here, we restrict ourselves to a brief exposition, and we do not try to justify the choice of these particular concepts.

We consider an execution of our algorithm to consist of a collection of *operations*. Each operation is composed of a set of indivisible *actions*. To be consistent with the terminology of [5], we use the term "event" instead of "action." Figure 1 shows some of the operations which are generated by a single process. Note that an execution of the entire statement L2 is considered to be a single operation. Each "test $n[j]$" operation consists of executing a single iteration of the **wait until** loop in statement L3. We do not specify what the operations are in the noncritical section. In particular, we do not assume that an execution of the noncritical section must terminate.

To describe the temporal ordering of operations, we define two relations between a pair of operations A and B: (1) $A \rightarrow B$ (read A *precedes* B), and (2) $A \dashrightarrow B$ (read A *can influence* B). In a nondistributed system, they can be defined as follows: (1) $A \rightarrow B$ if A is completed before B is begun, and (2) $A \dashrightarrow B$ if A is begun before B is completed. For a distributed system, they may be

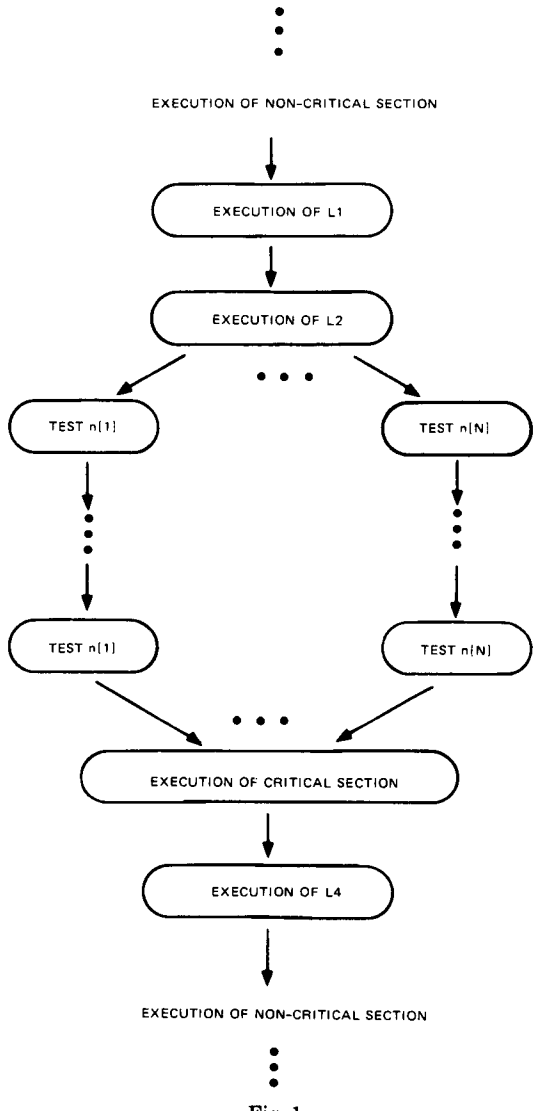

Fig. 1

defined as follows in terms of the precedence relation between events defined in [5]: (1) $A \rightarrow B$ if every event in A precedes every event in B, and (2) $A \dashrightarrow B$ if some event in A precedes some event in B. We say that A and B are *concurrent* if $A \not\rightarrow B$ and $B \not\rightarrow A$.

Figure 1 describes all the relations $\rightarrow$ between the operations of a single process which are specified by the algorithm. For example, the algorithm specifies that an execution of statement L1 must precede the execution of the subsequent statement L2. However, the algorithm specifies no temporal ordering between a "test $n[j]$" operation and a "test $n[j']$" operation during a single execution of statement L3, for $j \neq j'$. Any $\rightarrow$ or $\dashrightarrow$ relation that exists between these two

operations is due to the details of how the algorithm is implemented, and is extraneous to its correctness.

Either of our definitions of the relations $\rightarrow$ and $\dashrightarrow$ allow us to derive the following simple laws. (The proofs are trivial.)

A1. The relation $\rightarrow$ is transitively closed ($A \rightarrow B \rightarrow C$ implies $A \rightarrow C$) and irreflexive ($A \not\rightarrow A$).
A2. If $A \rightarrow B$ then $A \dashrightarrow B$ and $B \dashrightarrow A$.
A3. If $A \rightarrow B \dashrightarrow C$ or $A \dashrightarrow B \rightarrow C$ then $A \dashrightarrow C$.
A4. If $A \rightarrow B \dashrightarrow C \rightarrow D$ then $A \rightarrow D$.

We take A1–A4 to be axioms. This allows us to consider the operations to be the fundamental entities, and to forget that they are composed of indivisible events.

We must augment these axioms with an additional axiom A5. It states essentially that the algorithm begins executing at some time (rather than having been running forever), and that each operation takes a finite, positive length of time.

A5. For every operation A there exist only a finite number of operations B such that $B \dashrightarrow A$.

The mutual exclusion condition can be stated quite simply as follows: executions of the critical section by two different processes must not be concurrent. That is, if the operation CS is an execution of the critical section by process i, and CS' is an execution of the critical section by process $i' \neq i$, then either $CS \rightarrow CS'$ or $CS' \rightarrow CS$. Observe that this is a precise statement of the mutual exclusion condition which is independent of the algorithm used to implement it.

4. OPERATIONS INVOLVING n[i]

Having introduced a general notation for discussing operations, we now consider the special case of the operations which fetch and write the shared variable $n[i]$. Let W_1, W_2, ... denote the operations which write $n[i]$—i.e., all executions of statements L1, L2, and L4 by process i. We can assume that $W_1 \rightarrow W_2 \rightarrow \ldots$, since the algorithm does not allow concurrent operations to write $n[i]$. We let W_0 denote the operation which initializes $n[i]$ to zero, and assume that W_0 precedes any other operation which fetches or writes the value of $n[i]$.

As in [6], we let $n[i]^{[r]}$ denote the value written by W_r, and we say that an operation which fetches the value of $n[i]$ obtains the "value" $n[i]^{[r,s]}$ if it observes traces of the values $n[i]^{[r]}$, $n[i]^{[r+1]}$, ... , $n[i]^{[s]}$. This can be defined more precisely as follows.

Definition. An operation R which fetches the value of $n[i]$ is said to obtain the value $n[i]^{[r,s]}$ where r = maximum $\{t: W_t \rightarrow R\}$ and s = maximum $\{t: W_t \dashrightarrow R\}$.

Note. The existence of the r and s follows from A2, A5, and the assumption that W_0 precedes R. □

If the operation R obtains the value 7 as the result of fetching the value of $n[i]$, then we write $n[i]^{[r,s]} = 7$. This notation is not strictly correct, because another operation R' executed concurrently with R by another process could obtain $n[i]^{[r,s]} = 17$. However, it is convenient and should cause no confusion.

Axioms A1–A3 easily imply the following.

PROPOSITION. *Let R be a fetch of $n[i]$ which obtains the value $n[i]^{[r,s]}$. Then $r \leq s$ and (i) $W_t \rightarrow R$ if and only if $t \leq r$, (ii) $W_t \dashrightarrow R$ if and only if $t \leq s$.*

We can now state requirements R1 and R2 precisely as follows:

R1. For any r: $n[i]^{[r,r]} = n[i]^{[r]}$.

R2. Let W_r be an execution of L1 by process i, and let W_{r+1} be the subsequent execution of L2. If a "test $n[i]$" operation in the execution of L3 by another process obtains the value $n[i]^{[r,r+1]}$, then $0 < n[i]^{[r,r+1]} \leq n[i]^{[r+1]}$.

We will also need the following requirement:

R3. If W_r is an execution of L1 by process i, and R is a "test $n[i]$" operation in the execution of L3 by another process, then either $W_r \rightarrow R$ or $R \dashrightarrow W_r$.

This requirement is always satisfied in a nondistributed system, since for such a system either $A \rightarrow B$ or $B \dashrightarrow A$ must hold for any pair of distinct operations A and B. (This follows from the definitions of $\rightarrow$ and $\dashrightarrow$ for a nondistributed system given in Section 3.) However, this is not true for a distributed system. In Section 7, we discuss the implication of R3 for implementations in a distributed system.

5. THE PROOF OF CORRECTNESS

We now use A1–A5 and R1–R3 to prove that our algorithm satisfies the mutual exclusion condition. We begin by introducing some notation needed for the proof. We define *I1–I4* and *CSI* to be the following operations executed during a single iteration of the **repeat** loop of process i, where $i' \neq i$:

I1	execution of L1: $n[i]^{[t]} := 0$,
I2	execution of L2: $n[i]^{[t+1]} := \mathit{maximum}\ (\ldots, n[i']^{[p,q]}, \ldots)$,
I3	last "test $n[i']^{[r,s]}$" during execution of L3,
CSI	execution of the critical section,
I4	execution of L4: $n[i]^{[t+2]} := 0$.

Thus the operation *I2* is an execution of statement L2 by process i. This operation is both a fetch of $n[i']$ which obtains the "value" $n[i']^{[p,q]}$ and a write of $n[i]$ which writes the value $n[i]^{[t+1]}$. Similarly, *I3* is the last "test $n[i']$" operation performed by process i before entering its critical section. It is a fetch of $n[i']$ which obtains the "value" $n[i']^{[r,s]}$.

The following relations follow immediately from the algorithm and from our definitions:

$$I1 \rightarrow I2 \rightarrow I3 \rightarrow CSI \rightarrow I4 \tag{1}$$

$$r \leq s \quad \text{and} \quad p \leq q \tag{2}$$

$$0 < n[i]^{[t]} < n[i]^{[t+1]} \tag{3}$$

$$n[i']^{[p,q]} < n[i]^{[t+1]} \tag{4}$$

$$n[i']^{[r,s]} = 0 \quad \text{or} \quad (n[i]^{[t+1]}, i) < (n[i']^{[r,s]}, i'). \tag{5}$$

Relation (5) follows from the fact that process i leaves its **wait until** loop for $j = i'$ after executing operation $I3$. (The strict inequality is because $i' \neq i$.)

Note. Relations $I1$ and (3)–(5) can be viewed as a formal specification of the algorithm described informally by the Algol style program of Section 2. For example, the requirement that (4) hold for all $i' \neq i$, and the inequality $n[i]^{[t]} < n[i]^{[t+1]}$ of (3) constitute a precise specification of statement L2. □

The operations $I1'$–$I4'$ and CSI' in process i' are defined in exactly the same way, except with primed and unprimed values interchanged. For example, we have

$I2'$ execution of L2: $n[i']^{[t'+1]} :> maximum\ (\ldots, n[i]^{[p',q']}, \ldots)$.

The relations (1′)–(5′) are similarly defined; e.g.

$$n[i]^{[p',q']} < n[i']^{[t'+1]}. \tag{4'}$$

To verify that our algorithm satisfies the mutual exclusion condition, we must prove that either $CSI \rightarrow CSI'$ or $CSI' \rightarrow CSI$. The proof is a matter of purely formal manipulations, using a1–A5, R1–R3, the proposition of Section 4, and relations (1)–(5) and (1′)–(5′). The reasoning involved can be formulated rigorously enough to permit mechanical verification. However, to facilitate human comprehension, the proof is presented in the style of ordinary, informal mathematics.

We separately consider three cases:

Case I: $r < t'$ or $r' < t$.
Case II: $s > t' + 1$ or $s' > t + 1$.
Case III: $t' \leq r \leq s \leq t' + 1$ and $t \leq r' \leq s' \leq t + 1$.

By (2) and (2′), these (nondisjoint) cases cover all possibilities.

Case I. Assume $r < t'$. The proposition implies that $I1' \not\rightarrow I3$. By R3, this implies that $I3 \dashrightarrow I1'$. Combining this with (1) and (1′) gives $I2 \rightarrow I3 \dashrightarrow I1' \rightarrow I2' \rightarrow I3'$. Using A4 and A1, we can then conclude that $I2 \rightarrow I2'$ and $I2 \rightarrow I3'$. The proposition then implies that

$$p' \geq t + 1 \quad \text{and} \quad r' \geq t + 1. \tag{I.1}$$

Since $q' \geq p'$ [by (2′)], (I.1) implies that we need only consider the following two subcases:

Case Ia. $q' > t + 1$.
Case Ib. $q' = p' = t + 1$.

Case Ia. Assume $q' > t + 1$. The proposition then implies that $I4 \dashrightarrow I2'$. Combining this with (1) and (1′), we have $CSI \rightarrow I4 \dashrightarrow I2' \rightarrow CSI'$. Axiom A4 then implies that $CSI \rightarrow CSI'$.

Case Ib. Assume $q' = p' = t + 1$. By R1 and (4′), this implies

$$n[i]^{[t+1]} < n[i']^{[t'+1]}. \tag{Ib.1}$$

By (I.1) and (2′), we need only consider the following two cases:

Case Ib(i): $s' > t + 1$.
Case Ib(ii): $s' = r' = t + 1$.

Case Ib(i). Assume $s' > t + 1$. The proposition then implies that $I4 \dashrightarrow I3'$. Using (1) and (1′) this gives $CSI \rightarrow I4 \dashrightarrow I3' \rightarrow CSI'$, and we conclude from A4 that $CSI \rightarrow CSI'$.

Case Ib(ii). Assume $s' = r' = t + 1$. By R1, (3), and (5′), this implies that $(n[i']^{[t'+1]}, i') < (n[i]^{[t+1]}, i)$. However, this inequality contradicts (Ib.1) (which still holds in the subcase), so this subcase is impossible.

This completes the proof of case I for $r < t'$. The proof for $r' < t$ is the same, except with primed and unprimed quantities interchanged. Thus we have finished with case I.

Case II. Assume $s > t' + 1$. The proposition then implies that $I4' \dashrightarrow I3$. Using (1) and (1′), we then obtain $CSI' \rightarrow I4' \dashrightarrow I3 \rightarrow CSI$, and use A4 to conclude that $CSI' \rightarrow CSI$. The proof for $s' > t + 1$ is obtained by interchanging primed and unprimed quantities.

Case III. Assume that $t' \leq r \leq s \leq t' + 1$. There are three subcases to be considered:

Case IIIa: $r = s = t'$.
Case IIIb: $r = s = t' + 1$.
Case IIIc: $r = t'$ and $s = t' + 1$.

From (3′), R1, and R2, it follows easily for each of these cases that $0 < n[i']^{[r,s]} \leq n[i']^{[t'+1]}$. We then conclude from (5) that

$$(n[i']^{[t+1]}, i) < (n[i']^{[t'+1]}, i'). \tag{III.1}$$

Starting from the assumption that $t \leq r' \leq s' \leq t + 1$, the same reasoning with primed and unprimed quantities interchanged yields

$$(n[i']^{[t'+1]}, i') < (n[i]^{[t+1]}, i). \tag{III.1′}$$

Since (III.1) and (III.1′) are contradictory, this case is impossible. This completes the proof.

Note. Axiom A5 was not used in the proof and could have been eliminated entirely by allowing r or s to equal infinity in the definition of Section 4. In the terminology of [3], this is because A5 is needed only to prove "liveness" properties, whereas mutual exclusion is a "safety" property. One would need A5 to prove that a process will eventually enter its critical section. □

If the reader has not examined other rigorous correctness proofs, then he may feel that the above proof is rather long and tedious. In this case, we urge him to consider the amount of detail involved in verifying the formal assertional proof for the original bakery algorithm in [3]. Rigorous correctness proofs for multiprocess programs are not easy. We feel that our new approach compares quite favorably to assertional methods in terms of the length and difficulty of the proofs.

6. IMPLEMENTATION IN A NONDISTRIBUTED SYSTEM

We now describe two different ways of implementing our algorithm in a "traditional" nondistributed environment, in which processes communicate via shared memory. The value of an integer variable is stored as a multidigit number; i.e. as a finite-length string of nonnegative integers in the usual way. Thus the integer 1 is always represented by the string of digits 00 ... 01. We assume that fetching or writing a single digit is an atomic operation. The algorithm is assumed to be executed as if no two atomic operations are executed concurrently.

The obvious way to implement such an integer in a conventional multiprocess computer is with a fixed number of digits, each occupying a single memory word. Implementations with a variable-length list of digits are also possible, but they are nontrivial and will be left as an exercise for the interested reader.

Note. Like the original bakery algorithm, the new algorithm can be implemented even with truly concurrent operations to the same memory word. Hardware implemented mutual exclusion is not required. However, the discussion of such an implementation would lead us away from the main purpose of this paper, so we simply assume mutually exclusive access to individual digits. □

With these assumptions, it is easy to see that condition R1 is satisfied by any reasonable implementation.[1] As we mentioned earlier, R3 is always satisfied in a nondistributed system. More precisely, R3 follows from the assumption that the atomic events comprising any two operations are totally ordered in time. The only implementation difficulty is satisfying R2. We give two ways of doing this.

Implementation 1. We begin with the most straightforward implementation. The value of $n[i]$ is stored as a list of digits, as described above. Statement L1 is implemented as follows:

$$\text{L1: } n[i] := 1.$$

In statement L2, $n[i]$ is set to the smallest integer greater than *maximum* $(n[1], \ldots, n[N])$ whose rightmost (least significant) digit is nonzero. Each digit is written at most once. To show that R2 is satisfied, we observe that the fetched value $n[i]^{[r,r+1]}$ in the statement of R2 is composed of a string of digits each of which is a digit of either $n[i]^{[r]} = 0 \cdots 01$ or $n[i]^{[r+1]}$. The right-hand digit of both these numbers is nonzero, so $n[i]^{[r,r+1]} > 0$. Each digit of $n[i]^{[r]}$ is less than or equal to the corresponding digit of $n[i]^{[r+1]}$, so $n[i]^{[r,r+1]} \leq n[i]^{[r+1]}$. Hence R2 is satisfied.

The problem with this implementation is that we do not know how fast the value of $n[i]$ can grow. The implementation is satisfactory if critical sections are not executed too frequently, since the values of all the $n[i]$ drop back to zero when all processes are in their noncritical sections. However, this may never happen if critical sections are executed frequently. Let μ_j denote the total number of iterations of the **repeat** loop which process j has begun. In order to allow practical implementations with fixed-length integers, we would like some inequality such as $n[i] \leq k\sum\mu_j$ to hold (where k is a small constant). We do not know if this is true for implementation 1. However, the inequality $n[i] \leq \sum(\mu_j + 1)$ is

[1] An example of an unreasonable implementation would be one in which a fetch of $n[i]$ examines only some of its digits and tries to guess the rest.

easily proved for the following implementation using the results of [6]:

Implementation 2. We encode the value of $n[i]$ in the following way, using an integer variable $nn[i]$ and a Boolean variable $zf[i]$:

$$n[i] \equiv \textbf{if } zf[i] \textbf{ then } 0 \textbf{ else } nn[i] \textbf{ fi}$$

Initially, $zf[i] = \textbf{true}$ and $nn[i] = 0 \ldots 01$. We use the notation introduced in [6] that an arrow over a fetch or store instruction indicates that the individual digits are to be fetched or stored either from left to right (most significant to least significant) or right to left, depending upon the direction of the arrow. the implementation is given below. (Note that process i can read the digits of $nn[i]$ in any order, since no other process can change $nn[i]$.)

repeat *noncritical section*;
L1: $zf[i] := \textbf{false}$;
L2: $\overleftarrow{nn[i]} := 1 + \mathit{maximum}\ (\overrightarrow{nn[1]}, \ldots, \overrightarrow{nn[N]})$
L3: **for all** $j \in \{1, \ldots, N\}$ **do**
wait until $zf[j]$ **or** $(\overrightarrow{nn[j]}, j) \geq (nn[i], i)$ **od**;
critical section;
L4: $zf[i] := \textbf{true}$
end.repeat

The fact that R2 is satisfied is an easy consequence of the following result: if a "test $n[i]$" operation fetches the value $nn[i]^{[r, s]}$, then $nn[i]^{[r, s]} \leq nn[i]^{[s]}$. This result in turn follows immediately from theorem 2 of [6].

7. IMPLEMENTATION IN A DISTRIBUTED SYSTEM

We now consider the implementation of our algorithm in a distributed system. A rigorous, detailed discussion of this case would be rather long, and would involve some fundamental issues which we prefer not to introduce in this paper. Therefore we will just briefly describe our approach, omitting the details.

We assume that there is no shared memory, but that processes communicate by sending signals to one another. A set of signals used to convey some unit of information is called a *message*. The actual mechanism by which messages are transmitted does not concern us.

We implement a program variable by having each process maintain its own local copy of the variable. A fetch is performed using that local copy. To store a value into the variable, a process must send a message containing the new value to every other process. We make the following requirement on how this is done.

MX. For any pair of processes i and i': messages sent from process i to process i' are acted upon by process i' in the same order in which they were sent.

Thus, process i' will perceive the changes to $n[i]$ as occurring in the same order that they are made by process i. How MX is implemented will depend upon the details of how messages are transmitted, and might require the use of sequencing information in the message.

We first observe that in the correctness proof of Section 5, we did not need to assume that axioms A1–A5 hold for the entire set of operations. It sufficed to assume that these axioms are satisfied for every pair of processes. In other words, we need only assume that for every pair of processes, A1–A5 hold for the set of

operations generated by those two processes. For example, we did not have to assume that $A \rightarrow B \rightarrow C$ implies $A \rightarrow C$ if A, B, and C are operations generated by three different processes. It can be shown that with the proper definition of $\rightarrow$ and $\dashrightarrow$ (i.e. with the proper identification of operations with sets of events), MX implies that A1–A5 are satisfied for every pair of processes, and that R1 also holds.[2] Moreover, the two implementations described in Section 6 guarantee that R2 holds in this case too. (In some situations, it will be reasonable to implement fetches and stores to an entire local copy of $n[i]$ as atomic operations, making R2 trivially true.)

Note. For implementation 2, it is necessary to show that the theorems of [6] can be proved using only axioms A1–A5. We leave this as an exercise for the reader. □

Finally, we must insure that R3 holds. We can do this by adding the following requirement for the implementation of statement L1:

> After sending a store message to other processes, process i must wait until it receives a message from every other process acknowledging that that process has executed the stored operation in its local copy of $n[i]$.

Any solution to the mutual exclusion problem in a distributed system must involve some such waiting for acknowledgments from other processes. (Otherwise, there would be a solution that worked despite infinitely long propagation delays—i.e. without any interprocess communication at all.) We could simply have required that a process await such an acknowledgment for every store operation. This would have introduced all the delays needed to make our distributed system behave exactly like a nondistributed one with a very slow central memory. However, the precise statement of the requirements R1–R3 enabled us to implement the algorithm with the minimum amount of waiting for acknowledgments.

8. CONCLUSION

When multiprocess algorithms were first studied, their correctness was proved using informal arguments based upon execution sequences, as in [4]. This method of proving correctness was found to be unreliable—it was too easy to give convincing correctness "proofs" for incorrect algorithms. The unreliability resulted from the enormous variety of possible execution sequences allowed by a multiprocess algorithm, and the difficulty of insuring that all possibilities had been considered. This led to the development of assertional techniques by Owicki [7], Keller [2], Lamport [3], and others which generalized the methods previously developed for sequential programs. These assertional techniques are based upon considering the states of the processes rather than their execution sequences. Proofs based upon assertional techniques appear to be more reliable than the old style of proof.

The approach introduced here, utilizing axioms A1–A5 and the definition and proposition of Section 4, represents a step backwards in that it is based upon execution sequences rather than assertions about states. However, it represents

[2] For A1–A5 to be satisfied for the entire set of operations, MX would have to be strengthened to guarantee that information transmitted indirectly from process i to process i' via intermediary processes is acted upon in the correct order.

an advance beyond the earlier, informal methods for two reasons. First, we have introduced a collection of concepts and notations which seem to permit more reliable, rigorous proofs. Second, we have reduced the complexity of the proofs by allowing nonatomic operations. (For example, we could consider executing the entire complex statement L2 to be a single operation.) With fewer individual operations, there are fewer execution sequences to be considered.

Despite these advances, our approach still lacks the strong formal foundation of assertional techniques. However, it offers some significant advantages over those techniques. First of all, it can handle nonatomic operations. This permits a multiprocess system to be designed by hierarchical decomposition. Because it is so simple, our algorithm may appear to be an ordinary low level synchronization algorithm. However, it is actually a higher level specification of an algorithm, since it permits the two very different implementations of Section 6. We were able to prove the correctness of the algorithm independently of the lower level algorithms used to implement its operations. So far, assertional techniques have not worked for nonatomic operations. We know no way of using assertional techniques to prove the correctness of such a higher level specification.

Note. Recent work by Owicki and others has addressed the problem of determining when a sequence of atomic operations can be treated as a single atomic operation. However, this is different from our approach which does not demand that the operations behave as though they were atomic. □

The other advantages of our approach stem from its generality. We believe that the relations $\rightarrow$ and $\dashrightarrow$ and axioms A1–A5 are fundamental to all concurrent systems. Thus our approach should permit natural specifications of the problem to be solved (or of the system to be implemented) in a manner that is completely independent of the solution. In contrast, assertional methods require that the correctness conditions be stated in terms of the objects (such as program variables) used in the solution. Notice how natural the statement of the mutual exclusion condition is in terms of the relation $\rightarrow$. This is in sharp contrast with the indirect method of proving mutual exclusion described by Owicki [7]. The other requirements for a solution to the mutual exclusion problem, described informally in [4], can also be expressed quite naturally with our notation. A similar conclusion has been reached by Greif [1].

The generality of our approach also means that it is applicable to any kind of multiprocess system. Assertional techniques have so far proved useful only for traditional, nondistributed multiprocess systems. Our approach is applicable to distributed systems as well. Thus, our specification of the algorithm permitted nontrivial implementations with a distributed system, as well as conventional implementations using shared memory.

We expect that ultimately our approach will be used in the high level design of a multiprocess system, while assertional techniques will be used to verify the correct implementation of the higher level operations. However, much more experience is needed to determine the approach's range of applicability and its relation to assertional techniques. In a future paper, we intend to describe in more detail the concepts introduced in Section 3, and to discuss some fundamental issues of distributed systems which we could not deal with in this paper. The example we have considered here should illustrate the application of our approach to traditional multiprocessing problems in a nondistributed environment.

ACKNOWLEDGMENTS

The final version of the algorithm described here was discovered during a discussion with C.S. Scholten and E.W. Dijkstra. The excellent Dutch beer consumed on that occasion and the subsequent passage of time have conspired to prevent us from accurately apportioning the credit for its discovery.

REFERENCES

1. GREIF, I. A language for formal problem specifications. *Comm. ACM 20*, 12 (Dec. 1977), 931–935.
2. KELLER, R. Formal verification of parallel programs. *Comm. ACM 19*, 7 (July 1976), 371–384.
3. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng. SE-3*, 7 (March 1977), 125–143.
4. LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM 17*, 8 (Aug. 1974), 453–455.
5. LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Comm. ACM 21*, 7 (July 1978), 558–565.
6. LAMPORT, L. Concurrent reading and writing. *Comm. ACM 20*, 11 (Nov. 1977), 806–811.
7. OWICKI, S., AND GRIES, D. Verifying properties of parallel programs: an axiomatic approach. *Comm. ACM 19*, 5 (May 1976), 279–285.

Received January 1978; revised November 1978