The Mutual Exclusion Problem
Part I: A Theory of Interprocess Communication

L. Lamport[1]
Digital Equipment Corporation

6 October 1980
Revised:
1 February 1983
1 May 1984
27 February 1985
June 26, 2000

To appear in *Journal of the ACM*

[1]Most of this work was performed while the author was at SRI International, where it was supported in part by the National Science Foundation under grant number MCS-7816783.

Abstract

A novel formal theory of concurrent systems is introduced that does not assume any atomic operations. The execution of a concurrent program is modeled as an abstract set of operation executions with two temporal ordering relations: "precedence" and "can causally affect". A primitive interprocess communication mechanism is then defined. In Part II, the mutual exclusion is expressed precisely in terms of this model, and solutions using the communication mechanism are given.

Contents

1 Introduction

The mutual exclusion problem was first described and solved by Dijkstra in [3]. In this problem, there is a collection of asynchronous processes, each alternately executing a critical and a noncritical section, that must be synchronized so that no two processes ever execute their critical sections concurrently. Mutual exclusion lies at the heart of most concurrent process synchronization and, apart from its practical motivation, the mutual exclusion problem is of great theoretical significance.

The concept of mutual exclusion is deeply ingrained in the way computer scientists think about concurrency. Almost all formal models of concurrent processing are based upon an underlying assumption of mutually exclusive atomic operations, and almost all interprocess communication mechanisms that have been proposed require some underlying mutual exclusion in their implementation. Hence, these models and mechanisms are not satisfactory for a fundamental study of the mutual exclusion problem. We have therefore been forced to develop a new formalism for talking about concurrent systems, and a new way of viewing interprocess communication. Part I is entirely devoted to this formalism, which we believe provides a basis for discussing other fundamental problems in concurrent processing as well; the mutual exclusion problem itself is discussed in Part II [5].

The formal model we have developed is radically different from commonly used ones, and will appear strange to computer scientists accustomed to thinking in terms of atomic operations. (It is a slight extension to the one we introduced in [6].) When diverging from the beaten path in this way, one is in great danger of becoming lost in a morass of irrelevance. To guard against this, we have continually used physical reality as our guidepost. (Perhaps this is why hardware designers seem to understand our ideas more easily than computer scientists.) We therefore give a very careful physical justification for all the definitions and axioms in our formalism. Although this is quite unusual in theoretical computer science, we feel that it is necessary in explaining and justifying our departure from the traditional approach.

2 The Model

We begin by describing a formal model in which to state the problem and the solution. Except for the one introduced by us in [6], all formal models of concurrent processes that we know of are based upon the concept of an indivisible atomic operation. The concurrent execution of any two atomic operations is assumed to have the same effect as executing them in some order. However, if two operations can affect one another—e.g., if they perform interprocess communication—then implementing them to be atomic is equivalent to making the two operations mutually exclusive. Hence, assuming atomic operations is tantamount to assuming a lower-level solution to the mutual exclusion problem. Any algorithm based upon atomic operations cannot be considered a fundamental solution to the mutual exclusion problem. We therefore need a formalism that is not based upon atomic operations. The one we use is a slight extension to the formalism of [6].

2.1 Physical Considerations

For our results to be meaningful, our formalism must accurately reflect the physical reality of concurrent processes. We therefore feel that it is important to justify the formalism on physical grounds. We do this in terms of the geometry of space-time, which lies at the foundation of all modern physics. We begin with a brief exposition of this geometry. A more thorough exposition can be found in [15] and [16], but for the more sophisticated reader we recommend the original works [4, 11].

The reader may find the introduction of special relativity a bit far-fetched, since one is rarely, if ever, concerned with systems of processes moving at relativistic velocities relative to one another. However, the relativistic view of time is relevant whenever signal propagation time is not negligibly small compared to the execution time of individual operations, and this is certainly the case in most multiprocess systems.

Because it is difficult to draw pictures of four-dimensional space-time, we will discuss a three-dimensional space-time for a two-dimensional spatial universe. Everything generalizes easily to four-dimensional space-time.[1] We picture space-time as a three-dimensional Cartesian space whose points are called *events*, where the point (x, y, t) is the event occurring at time t at the point with spatial coordinates (x, y). Dimensional units are chosen so the speed of light equals 1.

The *world line* of a point object is the locus of all events (x, y, t) such that the object is at location (x, y) at time t. Since photons travel in a straight

[1] While it is even easier to draw pictures of a two-dimensional space-time with a single space dimension, a one-dimensional space has some special properties (such as the ability to send a light beam in only two directions) that can make such pictures misleading.

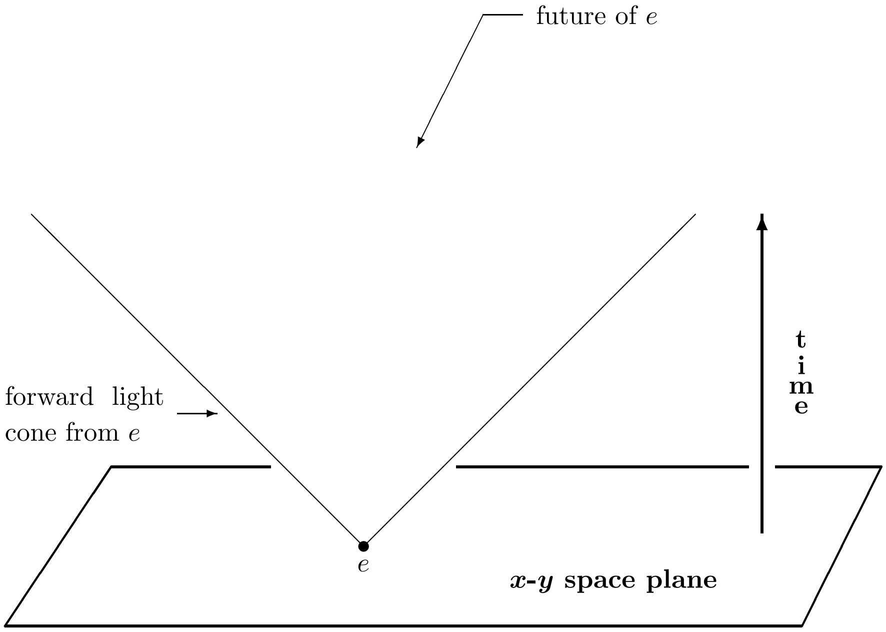

Figure 1: Space-Time

line with speed 1, the world line of a photon is a straight line inclined at 45° to the x-y plane. The forward light cone emanating from an event e is the surface formed by all possible world lines of photons created at that event. This is illustrated in Figure 1. The *future* of event e consists of all events other than e itself that lie on or inside the future light cone emanating from e. It is a fundamental principle of special relativity that an event e can only influence the events in its future.

We say that an event e *precedes* an event f, written $e \longrightarrow f$, if f lies in the future of e. It is easy to see that $\longrightarrow$ is an irreflexive partial ordering—i.e., that (i) $e \not\longrightarrow e$ and (ii) $e \longrightarrow f \longrightarrow g$ implies $e \longrightarrow g$. Two events are said to be *concurrent* if neither precedes the other. Since objects cannot travel faster than light, two different events lying on the world-line of an object cannot be concurrent.

We can think of the vertical line through the origin as the world line of some standard clock, where the event $(0, 0, t)$ on this world line represents the clock "striking" time t. A horizontal plane, consisting of all events having the same t-coordinate, represents the universe at time t—as viewed by us. However, another observer may have a different view of which events occur at time t. We define a *space-like* plane to be a plane making an angle of less than 45° with the x-y plane. For an inertial observer, the set of events occurring at time t forms a space-like plane through $(0, 0, t)$. (An inertial observer is one traveling in a straight line at constant speed.) For different values of t, these planes are parallel (for the same observer). Any space-like plane represents a set of events that some inertial observer regards as all occurring at the same time. Such a plane defines an obvious partitioning of space-time into three sets: the future, the past, and the present (the plane itself).

It follows from these observations that an event e precedes an event f if and only if every inertial observer regards e as happening before f, and events e and f are concurrent if and only if there is some observer who views them as happening at the same time.

2.2 System Executions

According to classical physics, the universe consists of a continuum of space-time events, each having no spatial or temporal extent. In computer science, one imposes a discrete structure upon this continuous universe, considering a system to consist of distinct *operation executions* such as reading a flip-flop or sending a message.[2] An infinite (usually bounded) set of space-time events is considered to be a single operation execution. For example, the operation execution of reading a flip-flop consists of events spatially located at the flip-flop and perhaps at some of the wires connected to it.

The boundary between the events of one operation execution and of other operation executions in the same processor is rather arbitrary; events occurring along the wire leading from the flip-flop can be included as part of the reading of the flip-flop or as part of a subsequent operation execution that uses the value that was read. The fine details of where the boundary is drawn do not matter; extending the region of space-time comprising the operation execution by a nanosecond here or a micron there makes no difference. However, the choice of which events belong to which operation executions can influence the properties we ascribe to the operations; the formalism used to describe a system can depend upon whether the events in the propagation of a value along a wire belong to the send or to the receive operation.

[2] Since the term "operation" often denotes a type of action that can be performed repeatedly, as in "the operation of addition", we write "operation execution" to emphasize that we are referring to a single instance of such an action.

An execution of a system therefore consists of a set of operation executions, where each operation execution consists of a nonempty set of space-time events. We define the relations $\longrightarrow$ and $\dashrightarrow$ on the set of operation executions as follows:

$$\begin{aligned} A \longrightarrow B &\stackrel{\text{def}}{=} \forall a \in A : \forall b \in B : a \longrightarrow b \\ A \dashrightarrow B &\stackrel{\text{def}}{=} \exists a \in A : \exists b \in B : a \longrightarrow b \text{ or } a = b \end{aligned}$$

Thus, $A \longrightarrow B$ means that every event of A precedes every event of B, and $A \dashrightarrow B$ means that some event of A either precedes or is the same as some event of B. (If a read of a flip-flop occurs while the flip-flop is also being set, some space-time events located at the flip-flop may belong to both operation executions.)

Remembering the meaning of the precedence relation for events, we read $A \longrightarrow B$ as "A precedes B", and $A \dashrightarrow B$ as "A can causally affect B". However, we think of "can causally affect" as a purely temporal relation, independent of what the operations are doing. Thus, $A \dashrightarrow B$ can hold even if A and B are *read* operations that cannot actually influence one another. We say that A and B are *concurrent* if $A \not\longrightarrow B$ and $B \not\longrightarrow A$. In other words, two operation executions are concurrent unless one precedes the other.

The following properties of the relations $\longrightarrow$ and $\dashrightarrow$ on operation executions follow directly from the fact that the relation $\longrightarrow$ on events is an irreflexive partial ordering:

A1. The relation $\longrightarrow$ is an irreflexive partial ordering.

A2. If $A \longrightarrow B$ then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. If $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$ then $A \dashrightarrow C$.

A4. If $A \longrightarrow B \dashrightarrow C \longrightarrow D$ then $A \longrightarrow D$.

There are two kinds of operation executions—terminating ones whose events all occur before some time (they are in the past of some space-like surface), and nonterminating ones that go on forever (their events do not lie in the past of any space-like surface). We make the following assumptions about these two classes of operation executions.

A5. For any terminating A, the set of B such that $A \not\longrightarrow B$ is finite.

A6. For any nonterminating A:

(a) The set of B such that $B \longrightarrow A$ is finite.

(b) For all B: $A \not\dashrightarrow B$.

Properties A5 and A6 can be derived from the following assumptions.

- At any time, there are only a finite number of operation executions that have begun by that time—i.e., for any space-like surface, there are only a finite number of operation executions containing events in the past of that surface.
- There are only a finite number of operation executions concurrent with any terminating operation execution.

The second assumption means that the speed with which the system is "spreading out" in space is bounded by some value less than the speed of light.

We have described operation executions in terms of events in order to justify A1–A6. In computer science, one ignores the space-time events that comprise operation executions. A programmer does not care that machine instructions are composed of more primitive events. In our formalism, operation executions are considered primitives elements, and A1–A6 are taken as axioms. We define a *system execution* to consist of a set of operation executions, partitioned into terminating and nonterminating ones, together with relations $\longrightarrow$ and $\dashrightarrow$ that satisfy Axioms A1–A6.

2.3 Higher-Level Views

A system can be viewed at many different levels; the programmer may consider the execution of a *load accumulator from memory* instruction to be a single operation, while the hardware designer considers it to be a sequence of lower-level register-transfer operations. The fundamental task in computing is to implement higher-level operations with lower-level ones. A hardware designer implements machine-language operations with register-transfer operations; a compiler writer implements Pascal operations with machine-language operations; and an applications programmer implements funds-transfering operations with Pascal operations. One assumes that the lower-level, primitive operations are given and uses them to construct the higher-level ones.

A higher-level operation execution consists of a set of lower-level ones. If we view operation executions as sets of space-time events, a higher-level operation execution is the union of the events of the (lower-level) operation executions it is composed of. It is nonterminating if and only if it consists of a finite number of nonterminating operation executions. It is not hard to show that the relations $\longrightarrow$ and $\dashrightarrow$ between higher-level operation executions can be computed from those relations between the lower-level operation executions as follows:

$$
\begin{aligned}
R \longrightarrow S &= \forall A \in R : \forall B \in S : A \longrightarrow B \\
R \dashrightarrow S &= \exists A \in R : \exists B \in S : A \dashrightarrow B \text{ or } A = B
\end{aligned}
\qquad (2\text{-}1)
$$

Since events do not appear in our formalism, we cannot proceed in this way. Instead, we take (2-1) to be the definition of the relations $\longrightarrow$ and $\dashrightarrow$ between any two sets of operation executions. By identifying an operation execution A with the set $\{A\}$, this definition also applies when R or S is a single operation execution rather than a set of them. A set of operation executions is defined to be *terminating* if and only if it consists of a finite number of terminating operation executions.

Given a system execution, a higher-level view of that execution consists of a partitioning of its operation executions into sets, which represent higher-level operation executions. The machine-language view of a system is obtained by partitioning the register-transfer operations into executions of machine-language instructions. This need not be a true partition; a single register-transfer operation could be part of the execution of two separate machine-language instructions. We therefore define a *higher-level view* of a system execution to be a collection $\mathcal{H}$ of nonempty sets of operation executions such that each operation execution belongs to a finite number, greater than zero, of sets in $\mathcal{H}$. The elements of $\mathcal{H}$ (which are sets of operation executions) are called the operation executions of the higher-level view, or simply the *higher-level operation executions*.

Given a higher-level view of a system execution, we have defined the relations $\longrightarrow$ and $\dashrightarrow$ (by (2-1)) and the concept of termination on its high-level operation executions. Using these definitions and Axioms A1–A6 for the (lower-level) operation executions, it is easy to show that A1–A6 hold for the higher-level operation executions. Hence, the higher-level view of a system execution is itself a system execution.

In any study of computer systems, there is a lowest-level view that is of interest. The operation executions in that view will be called *elementary* operation executions. A set of elementary operation executions will be called an operation execution. (It is an operation execution in some higher-level view.)

3 Interprocess Communication

To achieve mutual exclusion, processes must be able to communicate with one another. We must therefore assume some interprocess communication mechanism. However, almost every communication primitive that has been proposed implicitly assumes mutual exclusion. For example, the first mutual exclusion algorithms assumed a central memory that can be accessed by all the processes, in which any two operations to a single memory cell occur in some definite order. In other words, they assumed mutually exclusive access to a memory cell. We will define an interprocess communication mechanism that does not assume any lower-level mutual exclusion. In order to explain our choice of a mechanism, we begin by examining the nature of interprocess communication.

The simplest form of interprocess communication is for a process i to send one bit of information to a process j. This can be done in two ways: by sending a message or by setting a bit. For example, if the physical communication medium is a wire, then "sending a message" might mean sending a pulse and "setting a bit" might mean setting a level. However, a message is a transient phenomenon, and j must be waiting for i's message in order to be sure of receiving it. We now show that with only this kind of transient communication, the mutual exclusion problem does not admit a solution in which the following two conditions hold:

- A process need communicate only when trying to enter or leave its critical section, not in its critical or noncritical sections.

- A process may remain forever in its noncritical section.

These conditions rule out algorithms in which processes take turns entering, or declining to enter, their critical section; such algorithms are really solutions to the producer/consumer problem [1].

Assume that a process i wants to enter its critical section first, while another process j is in its noncritical section. Since j could remain in its noncritical section forever, i must be able to enter its critical section without communicating with j. Assume that this has happened and i is in its critical section when j decides it wants to enter its critical section. Since i is not required to communicate while in its critical section, j cannot find out if i is in its critical section until i leaves the critical section. However, j cannot wait for a communication because i might be in, and remain forever in, its noncritical section. Hence, no solution is possible.

This conclusion is based upon the assumption that communication by transient messages can only be achieved if the receiving process is waiting for the message. This assumption may seem paradoxical since distributed systems often provide a message-passing facility with which a process can receive messages while engaged in other activity. A closer examination of such systems reveals that the receiving process actually consists of two concurrently executing subprocesses: a main subprocess that performs the process's major activity, and a communication subprocess that receives messages and stores them in a buffer to be read by the main subprocess, where one or more bits in the buffer may signal the main subprocess that it should interrupt its activity to process a message. The activity of the communication subprocess can be regarded as part of the sending operation, which effects the communication by setting bits in the buffer that can be read by the receiving process. Thus, this kind of message passing really involves the setting of bits at the remote site by the sender.

Hence, we assume that a process i communicates one bit of information to a process j by setting a communication bit that j can read. A bit that can be set but not reset can be used only once. Since there is no bound on the number of times a process may need to communicate, interprocess synchronization is impossible with a finite number of such "once only" communication bits. Therefore, we require that it be possible to reset the bit. This gives us three possibilities:

1. Only the reader can reset the communication bit.
2. Only the writer can reset the communication bit.
3. Both can reset the communication bit.

In case 1, with a finite number of bits, a process i can send only a bounded amount of information to another process j before j resets the bits. However, in the mutual exclusion problem, a process may spend arbitrarily long in its noncritical section, so process i can enter its critical section arbitrarily many times while process j is in its noncritical section. An argument similar to the one demonstrating that transient communication cannot be used shows that process i must communicate with process j every time it executes its critical section, so i may have to send an unbounded amount of information to j while j is in its noncritical section. Since a process need not communicate while in its noncritical section, the problem cannot be solved using the first kind of communication bit.[3]

Of the remaining two possibilities, we choose number 2 because it is more primitive than number 3. We are therefore led to the use of a communication bit that can be set to either of two values by one process and read by another—i.e., a boolean-valued communication variable with one writer and one reader. We let *true* and *false* denote the two values. We say that such a variable "belongs to" the process that can write it.

We now define the semantics of the operations of reading and writing a communication variable. A write operation execution for a communication variable has the form *write* $v := v'$, where v is the name of the variable, and v' denotes the value being written—either *true* or *false*. A read operation execution has the form *read* $v = v'$, where v' is the value obtained as the result of performing the read.

The first assumptions we make are:

C0. Reads and writes are terminating operation executions.

C1. A read of a communication variable obtains either the value *true* or the value *false*.

Physically, C1 means that no matter what state the variable is in when it is being read, the reader will interpret that state as one of the two possible values.

We require that all writes to a single communication variable be totally ordered by the $\longrightarrow$ relation. Since all of these writes are executed by the same process—the one that owns the variable—this is a reasonable requirement. We will see below that it is automatically enforced by the programming language in which the algorithms are described. This requirement allows us to introduce the following notation.

Definition 1 *For any variable v we let $V^{[1]}$, $V^{[2]}$, ... denote the write operation executions to v, where*

$$V^{[1]} \longrightarrow V^{[2]} \longrightarrow \cdots \quad .$$

We let $v^{[i]}$ denote the value written by the operation execution $V^{[i]}$.

[3]However, it is possible to solve producer/consumer problems with it. In fact, an interrupt bit of an ordinary computer is precisely this kind of communication bit, and it is used to implement producer/consumer synchronization with its peripheral devices.

Thus $V^{[i]}$ is a *write* $v := v^{[i]}$ operation execution. We assume that the variable can be initialized to either possible value. (The initial value of a variable can be specified as part of the process's program.)

If a read is not concurrent with any write, then we expect it to obtain the value written by the most recent write—or the initial value if it precedes all writes. However, it turns out that we need a somewhat stronger requirement. To justify it, we return to our space-time view of operations. The value of a variable must be stored in some collection of objects. Communication is effected by the reads and writes acting on these objects—i.e., by each read and write operation execution containing events that lie on the world lines of these objects. A read or write operation may also contain "internal" events not on the world line of any of these shared objects. For example, if the variable is implemented by a flip-flop accessed by the reader and writer over separate wires, the flip-flop itself is the shared object and the events occurring on the wires are internal events. The internal events of a write do not directly affect a read. However, for a write to precede ($\longrightarrow$) a read, all events of the write, including internal events, must precede all events of the read.

For two operation executions A and B on the same variable, we say that A "effectively precedes" B if every event in A that lies on the world line of one of the shared objects precedes any events in B that lie on the same object's world line. For a read to obtain the value written by $V^{[k]}$, it suffices that (i) $V^{[k]}$ effectively precedes the read, and (ii) the read effectively precedes $V^{[k+1]}$. "Effectively precedes" is weaker than "precedes", since it does not specify any ordering on internal events, so this condition is stronger than requiring that the read obtain the correct value if it is not concurrent with any write.

This definition of "effectively precedes" involves events, which are not part of our formalism, so we cannot define this exact concept. However, observe that if events a and b lie on the same world line, then either $a \longrightarrow b$ or $b \longrightarrow a$. Hence, if A and B both have events occurring on the same world line, then $A \dashrightarrow B$ and/or $B \dashrightarrow A$. If $B \not\dashrightarrow A$, then no event in B precedes any event in A. Hence, $A \dashrightarrow B$ and $B \not\dashrightarrow A$ imply that A effectively precedes B. We therefore are led to the following definition:

Definition 2 *We say that two operation executions A and B are* effectively nonconcurrent *if either $A \dashrightarrow B$ or $B \dashrightarrow A$, but not both.*

If two operation executions are effectively nonconcurrent according to this definition, then one "effectively precedes" the other according to the above definition in terms of events. We therefore expect a read that is effectively nonconcurrent with every write to obtain the correct value. This leads us to the following requirement.

C2. A read R of v that is effectively nonconcurrent with every $V^{[i]}$ obtains the value $v^{[k]}$, where k is the largest number such that $V^{[k]} \dashrightarrow R$, or it obtains the initial value if there is no such k.

It follows from A2 and A5 that the set of k such that $V^{[k]} \dashrightarrow R$ is finite, so C2 specifies the value obtained by a read that is effectively nonconcurrent with every write. The only assumption we make about a read that is "effectively concurrent" with some write is that it obtain either the value *true* or the value *false* (by C1).

In the above space-time discussion of reading and writing, it is clear that for communication to take place, every pair of reads and writes must have events on the world line of the same object. The following requirement is therefore quite reasonable (although it may not be obvious why we need it).

C3. If R is a read of the communication variable v, then for every write $V^{[i]}$ of v: $R \dashrightarrow V^{[i]}$ or $V^{[i]} \dashrightarrow R$ (or both).

It has been argued that the kind of communication variable we are assuming is equivalent to one in which reads and writes are atomic actions that cannot be concurrent. The reasoning used is as follows.

> If the value of the variable is not changed by a write, then there is no reason to do the write. We may therefore assume that a process executes a write only if it will change the value. By C1, a read that is concurrent with such a write must obtain either the old or the new value, since those are the only possible values. If the read obtains the old value, then we may consider it to have preceded the write, and if it obtains the new value then we may consider the write to have preceded it.[4]

This reasoning is fallacious under our assumptions because if two successive reads are concurrent with the same write, then the first read can obtain the new value and the second the old value. This is impossible if reads and writes are nonconcurrent atomic actions.

[4] In fact, we made this unfortunate claim in our original correctness proof for the bakery algorithm [7]. Happily, it was only the proof and not the algorithm that turned out to be incorrect.

Several people have devised mutual exclusion algorithms using communication variables similar to ours, except with the stronger assumption that writing and reading are atomic operations [13], [14]. We believe that these algorithms do not work with the more primitive type of communication variable that we are assuming.[5] Other than the ones mentioned here, we know of no published mutual exclusion algorithms that are correct using these communication variables.

4 Processes

An algorithm implements higher-level operations such as *request service* in terms of lower-level ones like reading and writing a one-bit variable. A synchronization problem is posed as a set of conditions on the higher-level system execution—for example, that each *request service* operation execution is followed by a *grant service* operation execution. A solution consists of a specification of a lower-level system execution together with a higher-level view—for example, an algorithm for generating reads and writes together with a specification of which sets of these lower-level operation executions correspond to *request service* and *grant service* executions. The system execution defined by this higher-level view must satisfy the problem conditions.

We now consider how the lower-level system execution is specified. We assume that the set of all elementary operation executions is partitioned into N sets called *processes*. A process is described by an ordinary program, each operation execution of the process being generated by the execution of some statement in its program. For example, suppose the program for a process contains the following program statement:

begin
    $x := 1;$
    $z := y + z$
end

Executing this statement might generate the following four elementary operation executions, with the indicated $\longrightarrow$ relations.

[5] We have found counterexamples to the simpler algorithms, and have no reason to expect the more complicated ones to work better.

$$
\begin{tikzcd}
 & \textit{read } y = y' \arrow[dr] & \\
\textit{write } x := 1 \arrow[ur] \arrow[dr] & & \textit{write } z := y' + z' \\
 & \textit{read } z = z' \arrow[ur] &
\end{tikzcd}
$$

Although we think of the program as generating the operation executions, formally the set of operation executions is given and the processes' programs provide a set of conditions on it. For example, if this statement were the only place where y is read, then it would provide the following formal condition:

> For every *read* $y = y'$ operation execution, there must exist three operation executions *write* $x := 1$, *read* $z = z'$, and *write* $z := y' + z'$ such that the above $\longrightarrow$ relations hold.

Each process will be described by a program written in an Algol-like language with two kinds of program variables:

- *Private variables* read and written by that process only.
- *Communication variables* used for interprocess communication.

We can define a formal semantics for the programming language as follows. The elementary operation executions of a process are of the form *write* $v := v'$ or *read* $v = v'$, where v is a variable and v' is an element in the range of values of that variable. The variable v must be one that the process can write or read, respectively. For the critical section problem, there are also elementary operation executions of the form *critical section execution* and *noncritical section execution*.

We assume that C0–C3 hold for reads and writes of communication variables. We also assume that C0 and C2 hold for private variables. We will not need C1 or C3 because a read of a private variable will never be concurrent with a write of that variable. In fact, a read will not be concurrent with any write performed by the same process.

We now indicate how a process's program can be formally translated into a set of conditions on possible system executions. Syntactically, a program is composed of a hierarchy of program statements—more complicated statements being built up from simpler ones.[6] In any system execution, to

[6] If function calls are permitted, then we have to include expression evaluations as well as statement executions in this hierarchy.

each program statement corresponds a (not necessarily elementary) operation execution—intuitively, it is the set of elementary operation executions performed when executing that statement. The execution of the entire program, which is a single statement, is a single operation execution consisting of all the process's elementary operation executions. The semantics of each type of statement in the language is defined by a collection of axioms on the (set of elementary operation executions in the) execution of a statement of that type. For assignment statements, we have the following axiom:

> An execution of the statement
>
> $$v \ := \ F(v_1, \ldots, v_m)$$
>
> consists of the elementary operation executions *read* $v_1 = v'_1$, $\ldots$, *read* $v_m = v'_m$, and *write* $v := F(v'_1, \ldots, v'_m)$, where each read precedes ($\longrightarrow$) the write.

This axiom, together with conditions C0–C3 for communication variables and C0 and C2 for private variables, defines the semantics of the simple assignment statement.

The following axioms define the semantics of the concatenation construction $S;T$. (Recall that a statement execution, being a set of elementary operation executions, is defined to terminate if and only if it consists of a finite number of terminating operation executions.)

> An execution of $S;T$ is one of the following:
>
> - A nonterminating execution of S.
> - An operation execution of the form $A \cup B$, where:
>   - A is a terminating execution of S.
>   - B is an execution of T.
>   - $A \longrightarrow B$.

In this way, one can give a complete formal semantics for our programming language. However, we will not bother to do so, and will reason somewhat informally about system executions. We merely note the following properties:

- A write is not concurrent with any other operation generated by the same process. (However, it may be concurrent with operations generated by other processes.)

- Any elementary operation execution is concurrent with only a bounded number of elementary operation executions in the same process.

5 Multiple-Reader Variables

Thus far, we have assumed that communication variables can be read by only a single process. Using such variables, it is easy to construct a communication variable satisfying C0–C3 that can be read by several processes, though only written by one process. To implement a communication variable v that can be written by process i and read by processes $1, \ldots, N$, we use an array $v[1], \ldots, v[N]$ of variables, where $v[j]$ can be written by i and read by j. (All the $v[j]$ are communication variables except for $v[i]$, which is a private variable of process i.) Any statement $v := \ldots$ in process i's program is implemented as an operation of assigning the value of the right-hand expression to each element of the array, and any occurrence of the variable v in an expression within the program of process j is interpreted as an occurrence of $v[j]$. The fact that this construction works is implied by the following result, whose proof is left to the reader.

Theorem 1 *For each $j \neq i$, let $v[j]$ be a communication variable that is written by process i and read by process j. Assume that for all j, j' and all k:*

- *The initial values of $v[j]$ and $v[j']$ are equal.*
- $v[j]^{[k]} = v[j']^{[k]}$.
- $V[j]^{[k]} \longrightarrow V[j']^{[k+1]}$.

Let the initial value of v be defined to equal the initial value of the $v[j]$, let $V^{[k]}$ be defined to be $\{V[1]^{[k]}, \ldots, V[N]^{[k]}\}$, and define a read of v by a process $j \neq i$ to be a read of $v[j]$. Then C0–C3 are satisfied by the variable v (where $\longrightarrow$ and $\dashrightarrow$ are defined for the set of operation executions $V^{[k]}$ by (2-1)).

Formally, we are defining a higher-level view whose operation executions are the same as those of the original system execution except that the reads and writes of the $v[j]$ are partitioned into reads and writes of v. This theorem shows that the resulting higher-level system execution satisfies C0–C3. We take the reads and writes of v to be elementary operation executions, ignoring the lower-level operation executions that comprise them.

We will therefore assume that a communication variable can be written by its owner and read by any process. However, we must remember that the "cost" of implementing such a communication variable may depend upon the number of processes that actually read it. If the physical communication mechanism involves wires that join two processors, then the number of wires needed to implement a communication variable equals the number of readers of that variable, so a variable read by r processes may be almost r times as expensive as one read by a single process. However, it is quite reasonable to suppose that the variable could be implemented with a single wire to which each reader is connected. In this case, the cost of an r-reader variable may not be much greater than the cost of a single-reader variable.

6 Discussion of the Assumptions

In this section, we have made some tacit assumptions that may have passed unnoticed. The most obvious of these is the assumption that each process knows in advance who it might communicate with. This assumption seems to us to be reasonable for an underlying physical model in which processors (the physical hardware that executes processes) are connected in pairs by direct physical connections—e.g., wires or optical fibers. In such a model, it is natural to assume that a processor knows the existence of every physical connection. Indeed, it is only for such a model that a communication variable owned by a single process is reasonable. Thus, our work is not applicable to systems of anonymous processors connected along a common wire, as in an Ethernet [10].

Our first two assumptions, C0 and C1, appear quite innocent. However, the fact that reading and writing are not synchronized means that the reader can become suspended for arbitrarily long in a meta-stable state if it happens to read at exactly the wrong time. This is the "arbiter problem" discussed in [2] and [12]. As explained in [2], one can construct a device in which the reader has probability zero of remaining in such a meta-stable state forever. Hence, our assumptions can be satisfied if we interpret truth to mean "true with probability one".

There is an additional subtle assumption hidden in the combination of C2 with the ordering of operation executions within a process that we have been assuming. Suppose $v := \mathit{true}; \ldots$ is part of the program for process i. We are assuming that the *write* $v := \mathit{true}$ generated by an execution of the first statement precedes any operation execution A generated by the subsequent execution of the next statement. Now suppose that this is the last *write* v generated by process i, and that there is a *read* v execution by another process that is preceded by A. By A1, the *read* v is preceded by the *write* $v := true$, and since this is the last *write* v operation execution, C2 implies that the *read* v must obtain the value *true*.

Let us consider what this implies for an implementation. To guarantee that the *read* v obtains the value *true*, after executing the *write* $v := true$ the writer must be sure that v has settled into a stable state before beginning the next operation. For example, if the value of v is represented by the voltage level on the wire joining two processors, then the writer cannot proceed until the new voltage has propagated to the end of the wire. If a bound cannot be placed upon the propagation time, then such a simple representation cannot be used. Instead, the value must be stored in a flip-flop local to the writer, and the reader must interrogate its value by sending a signal along the wire and waiting for a response. Since the flip-flop is located at the writing process, and is set only by that process, it is reasonable to assume a bound upon its settling time. The wire, with its unknown delay, becomes part of the reading process. Thus, although satisfying our assumption in a distributed process poses difficulties, they do not seem to be insurmountable. In any case, we know of no way to achieve interprocess synchronization without such an assumption.

7 Conclusion

In Section 2, we developed a formalism for reasoning about concurrent systems that does not assume the existence of atomic operations. This formalism has been further developed in [8], which addresses the question of what it means for a lower-level system to implement a higher-level one.

Section 3 considered the nature of interprocess communication, and argued that the simplest, most primitive form of communication that can be used to solve the mutual exclusion problem consists of a very weak form of shared register that can be written and read concurrently. Interprocess communication is considered in more detail in [9], where the form of shared register we have defined is called a *safe* register. Algorithms for constructing stronger registers from safe ones are given in [9].

Acknowledgements

Many of these ideas have been maturing for quite a few years before appearing on paper for the first time here. They have been influenced by a number of people during that time, most notably Carel Scholten, Edsger Dijkstra, Chuck Seitz, Robert Keller, and Irene Greif.

References

[1] P. Brinch Hansen. Concurrent programming concepts. *Computing Surveys*, 5:223–245, 1973.

[2] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Trans. Comput.*, C-22:421–422, April 1973.

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

[4] A. Einstein. Zur electrodynamik bewegter korper. *Annalen der Physik*, 17:, 1905. Translated as "On the Electrodynamics of Moving Bodies" in *The Principle of Relativity*, Dover Publications.

[5] Leslie Lamport. The mutual exclusion problem—part ii: statement and solutions. To appear in *JACM*.

[6] 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.

[7] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[8] Leslie Lamport. On interprocess communication—part i: basic formalism. 1985. To appear in Distributed Computing.

[9] Leslie Lamport. On interprocess communication—part ii: algorithms. 1985. To appear in Distributed Computing.

[10] R. Metcalfe and D. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[11] H. Minkowski. *Space and Time*, pages 73–91. Dover.

[12] Richard Palais and Leslie Lamport. *On the Glitch Phenomenon*. Technical Report CA-7611-0811, Massachusetts Computer Associates, November 1976.

[13] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proc. ACM Symp. Thy. Comp.*, pages 91–97, ACM, 1977.

[14] Ronald L. Rivest and Vaughn R. Pratt. The mutual exclusion problem for unreliable processes: preliminary report. In *Proc. IEEE Symp. Found. Comp. Sci.*, pages 1–8, IEEE, 1976.

[15] J.T. Schwartz. *Relativity in Illustrations*. New York University Press, 1962.

[16] E. F. Taylor and J. A. Wheeler. *Space-Time Physics*. W.H. Freeman, San Francisco, 1966.