The Mutual Exclusion Problem
Part II: Statement and Solutions

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

The theory developed in Part I is used to state the mutual exclusion problem and several additional fairness and failure-tolerance requirements. Four "distributed" N-process solutions are given, ranging from a solution requiring only one communication bit per process that permits individual starvation, to one requiring about $N!$ communication bits per process that satisfies every reasonable fairness and failure-tolerance requirement that we can conceive of.

Contents

List of Figures

1 Introduction

This is the second part of a two-part paper on the mutual exclusion problem. In Part I [9], we described a formal model of concurrent systems and used it to define a primitive interprocess communication mechanism (communication variables) that assumes no underlying mutual exclusion. In this part, we consider the mutual exclusion problem itself.

The mutual exclusion problem was first described and solved by Dijkstra in [2]. 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. Dijkstra's original solution was followed by a succession of others, starting with [6].

These solutions were motivated by practical concerns—namely, the need to synchronize multiprocess systems using the primitive operations provided by the hardware. More recent computers usually provide sophisticated synchronization primitives that make it easy to achieve mutual exclusion, so these solutions are of less practical interest today. However, mutual exclusion lies at the heart of most concurrent process synchronization, and the mutual exclusion problem is still of great theoretical significance. This paper carefully examines the problem and presents new solutions of theoretical interest. Although some of them may be of practical value as well—especially in distributed systems—we do not concern ourselves here with practicality.

All of the early solutions assumed a central memory, accessible by all processes, which was typical of the hardware in use at the time. Implementing such a central memory requires some mechanism for guaranteeing mutually exclusive access to the individual memory cells by the different processes. Hence, these solutions assume a lower-level "hardware" solution to the very problem they are solving. From a theoretical standpoint, they are thus quite unsatisfactory as solutions to the mutual exclusion problem. The first solution that did not assume any underlying mutual exclusion was given in [10]. However, it required an unbounded amount of storage, so it too was not theoretically satisfying. The only other published solution we are aware of that does not assume mutually exclusive access to a shared resource is by Peterson [18].

Here, in Part II, we present four solutions that do not assume any underlying mutual exclusion, using the concurrently accessible registers defined in Part I [9]. They are increasingly stronger, in that they satisfy stronger conditions, and more expensive, in that they require more storage. The precise formulation of the mutual exclusion problem and of the various fairness and failure-tolerance assumptions, is based upon the formalism of Part I.

2 The Problem

We now formally state the mutual exclusion problem, including a number of different requirements that one might place upon a solution. We exclude from consideration only the following types of requirements.

- Efficiency requirements involving space and time complexity.
- Probabilistic requirements, stating that the algorithm need only work with probability one. (Solutions with this kind of requirement have recently been studied by Rabin [19].)
- Generalizations of the mutual exclusion problem, such as allowing more than one process in the critical section at once under certain conditions [4, 14], or giving the processes different priorities [14].

Except for these exclusions and one other omission (*r-bounded waiting*) mentioned below, we have included every requirement we could think of that one might reasonably want to place upon a solution.

2.1 Basic Requirements

We assume that each process's program contains a noncritical section statement and a critical section statement, which are executed alternately. These statements generate the following sequence of elementary operation executions in process i:

$$NCS_i^{[1]} \longrightarrow CS_i^{[1]} \longrightarrow NCS_i^{[2]} \longrightarrow CS_i^{[2]} \longrightarrow \cdots$$

where $NCS_i^{[k]}$ denotes the k^{th} execution of process i's noncritical section, $CS_i^{[k]}$ denotes the k^{th} execution of its critical section, and $\longrightarrow$ is the precedence relation introduced in Part I. Taking $NCS_i^{[k]}$ and $CS_i^{[k]}$ to be elementary operation executions simply means that we do not assume any knowledge of their internal structure, and does not imply that they are of short duration.

We assume that the $CS_i^{[k]}$ are terminating operation executions, which means that process i never "halts" in its critical section. However, $NCS_i^{[k]}$

may be nonterminating for some k, meaning that process i may halt in its noncritical section.

The most basic requirement for a solution is that it satisfy the following:

Mutual Exclusion Property: For any pair of distinct processes i and j, no pair of operation executions $CS_i^{[k]}$ and $CS_j^{[k']}$ are concurrent.

In order to implement mutual exclusion, we must add some synchronization operations to each process's program. We make the following requirement on these additional operations.

> No other operation execution of a process can be concurrent with that process's critical or noncritical section operation executions.

This requirement was implicit in Dijkstra's original statement of the problem, but has apparently never been stated explicitly before.

The above requirement implies that each process's program may be written as follows:

```
initial declaration;
repeat forever
    noncritical section ;
    trying ;
    critical section ;
    exit ;
end repeat
```

The *trying* statement is what generates all the operation executions between a noncritical section execution and the subsequent critical section execution, and the *exit* statement generates all the operation executions between a critical section execution and the subsequent noncritical section execution. The *initial declaration* describes the initial values of the variables. A solution consists of a specification of the *initial declaration*, *trying* and *exit* statements.

A process i therefore generates the following sequence of operation executions:

$$NCS_i^{[1]} \longrightarrow trying_i^{[1]} \longrightarrow CS_i^{[1]} \longrightarrow exit_i^{[1]} \longrightarrow NCS_i^{[2]} \longrightarrow \cdots$$

where $trying_i^{[1]}$ denotes the operation execution generated by the first execution of the *trying* statement, etc.

The second basic property that we require of a solution is that there be no deadlock. Deadlock occurs when one or more processes are "trying to enter" their critical sections, but no process ever does. To say that a process tries forever to enter its critical section means that it is performing a nonterminating execution of its *trying* statement. Since every critical section execution terminates, the absence of deadlock should mean that if some process's *trying* statement doesn't terminate, then other processes must be continually executing their critical sections. However, there is also the possibility that a deadlock occurs because all the processes are stuck in their *exit* statements. The possibility of a nonterminating *exit* execution complicates the statement of the properties and is of no interest here, since the *exit* statements in all our algorithms consist of a fixed number of terminating operations. We will therefore simply require of an algorithm that every *exit* execution terminates.

The absence of deadlock can now be expressed formally as follows:

Deadlock Freedom Property: If there exists a nonterminating *trying* operation execution, then there exist an infinite number of critical section operation executions.

These two properties, mutual exclusion and deadlock freedom, were the requirements for a mutual exclusion solution originally stated by Dijkstra in [2]. (Of course, he allowed mutually exclusive access to a shared variable in the solution.) They are the minimal requirements one might place on a solution.

2.2 Fairness Requirements

Deadlock freedom means that the entire system of processes can always continue to make progress. However, it does not preclude the possibility that some individual process may wait forever in its *trying* statement. The requirement that this cannot happen is expressed by:

Lockout Freedom Property: Every *trying* operation execution must terminate.

This requirement was first stated and satisfied by Knuth in [6].

Lockout freedom means that any process i trying to enter its critical section will eventually do so, but it does not guarantee when. In particular, it allows other processes to execute their critical sections arbitrarily many times before process i executes its critical section. We can strengthen the lockout freedom property by placing some kind of fairness condition on the order in which trying processes are allowed to execute their critical sections.

The strongest imaginable fairness condition is that if process i starts to execute its *trying* statement before process j does, then i must execute its critical section before j does. Such a condition is not expressible in our formalism because "starting to execute" is an instantaneous event, and such events are not part of the formalism. However, even if we were to allow atomic operations—including atomic reads and writes of communication variables—so our operations were actually instantaneous events, one can show that this condition cannot be satisfied by any algorithm. The reason is that with a single operation, a process can either tell the other processes that it is in its *trying* statement (by performing a write) or else check if other processes are in their *trying* statements (by performing a read), but not both. Hence, if two processes enter their *trying* statements at very nearly the same time, then there will be no way for them to decide which one entered first. This result can be proved formally, but we will not bother to do so.

The strongest fairness condition that can be satisfied is the following *first-come-first-served* (FCFS) condition. We assume that the *trying* statement consists of two substatements—a *doorway* whose execution requires only a bounded number of elementary operation executions (and hence always terminates), followed by a *waiting* statement. We can require that if process i finishes executing its *doorway* statement before process j begins executing its *doorway* statement, then i must execute its critical section before j does. Letting $doorway_i^{[k]}$ and $waiting_i^{[k]}$ denote the k^{th} execution of the *doorway* and *waiting* statements by process i, this condition can be expressed formally as follows.

First-Come-First-Served Property: For any pair of processes i and j and any execution $CS_j^{[m]}$: if $doorway_i^{[k]} \longrightarrow doorway_j^{[m]}$, then $CS_i^{[k]} \longrightarrow CS_j^{[m]}$.

(The conclusion means that $CS_i^{[k]}$ is actually executed.)

The FCFS property states that processes will not execute their critical sections "out of turn". However, it does not imply that any process ever actually executes its critical section. In particular, FCFS does not imply deadlock freedom. However, FCFS and deadlock freedom imply lockout freedom, as we now show.

Theorem 1 *FCFS and deadlock freedom imply lockout freedom.*

Proof: Suppose $trying_i^{[k]}$ is nonterminating. Since there are a finite number of processes, the deadlock freedom property implies that some process j performs an infinite number of $CS_j^{[m]}$ executions, and therefore an infinite number of $doorway_j^{[m]}$ executions. It then follows from Axiom A5 of Part I that $doorway_i^{[k]} \longrightarrow doorway_j^{[m]}$ for some m. The FCFS property then implies the required contradiction. ∎

The requirement that executing the doorway take only a bounded number of elementary operation executions means that a process does not have to wait inside its *doorway* statement. Formally, the requirement is that there be some *a priori* bound—the same bound for any possible execution of the algorithm—on the number of elementary operation executions in each $doorway_i^{[k]}$. Had we only assumed that the *doorway* executions always terminate, then any lockout free solution is always FCFS, where the *doorway* is defined to be essentially the entire *trying* statement. This requirement seems to capture the intuitive meaning of "first-come-first-served". A weaker notion of FCFS was introduced in [17], where it was only required that a process in its *doorway* should not have to wait for a process in its critical or noncritical section. However, we find that definition rather arbitrary.

Michael Fischer has also observed that a FCFS algorithm should not force a process to wait in its *exit* statement. Once a process has finished executing its critical section, it may execute a very short noncritical section and immediately enter its *trying* statement. In this case, the *exit* statement is effectively part of the next execution of the *doorway*, so it should involve no waiting. Hence, any $exit_i^{[k]}$ execution should consist of only a bounded number of elementary operation executions for a FCFS solution. As we mentioned above, this is true of all the solutions described here.

An additional fairness property intermediate between lockout freedom and FCFS, called *r-bounded waiting*, has also been proposed [20]. It states that after process i has executed its *doorway*, any other process can enter its critical section at most r times before i does. Its formal statement is the same as the above statement of the FCFS property, except with $CS_j^{[m]}$ replaced by $CS_j^{[m+r]}$.

2.3 Premature Termination

Thus far, all our properties have been constraints upon what the processes may do. We now state some properties that give processes the freedom to behave in certain ways not explicitly indicated by their programs. We have already required one such property by allowing nonterminating executions of the noncritical section—i.e., we give the process the freedom to halt in its noncritical section. It is this requirement that distinguishes the mutual exclusion problem from a large class of synchronization problems known as "producer/consumer" problems [1]. For example, it prohibits solutions in which processes must take turns entering their critical section.

We now consider two kinds of behavior in which a process can return to its noncritical section from any arbitrary point in its program. In the first, a process stops the execution of its algorithm by setting its communication variables to certain default values and halting. Formally, this means that anywhere in its algorithm, a process may execute the following operation:

begin
  set all communication variables to their default values;
  halt
end

For convenience, we consider the final halting operation execution to be a nonterminating noncritical section execution. The default values are specified as part of the algorithm. For all our algorithms, the default value of every communication variable is the same as its initial value.

This type of behavior has been called "failure" in previous papers on the mutual exclusion problem. However, we reserve the term "failure" for a more insidious kind of behavior, and call the above behavior *shutdown.* If the algorithm satisfies a property under this type of behavior, then it is said to be *shutdown safe* for that property.

Shutdown could represent the physical situation of "unplugging" a processor. Whenever a processor discovers that another processor is unplugged, it does not try to actually read that processor's variables, but instead uses their default values. We require that the processor never be "plugged back in" after it has been unplugged. We show below that this is really equivalent to requiring that the processor remain unplugged for a sufficiently long time.

The second kind of behavior is one in which a process deliberately *aborts* the execution of its algorithm. Abortion is the same as shutdown except for three things:

- The process returns to its noncritical section instead of halting.
- Some of its communication variables are left unchanged. (Which ones are specified as part of the algorithm.)

- A communication variable is not set to its default value if it already has that value.[1]

Formally, an abortion is an operation execution consisting of a collection of writes that set certain of the process's communication variables to their default values, followed by ($\longrightarrow$) a noncritical section execution. (The noncritical section execution may then be followed by a *trying* statement execution—or by another abortion.) For our algorithms, the value of a communication variable is set by an abortion if there is an explicitly declared initial value for the variable, otherwise it is left unchanged by the abortion. If an algorithm satisfies a property with this type of behavior, then it is said to be *abortion safe* for that property.

2.4 Failure

Shutdown and abortion describe fairly reasonable kinds of behavior. We now consider unreasonable kinds of behavior, such as might occur in the event of process failure. There are two kinds of faulty behavior that a failed process could exhibit.

- *Unannounced death*, in which it halts undetectably.
- *Malfunctioning*, in which it keeps setting its state, including the values of its communication variables, to arbitrary values.

An algorithm that can handle the first type of faulty behavior must use real-time clocks, otherwise there is no way to distinguish between a process that has died and one that is simply pausing for a long time between execution steps. An example of an algorithm (not a solution to our mutual exclusion problem) that works in the presence of such faulty behavior can be found in [8]. Consideration of this kind of behavior is beyond the scope of this paper.

A malfunctioning process obviously cannot be prevented from executing its critical section while another process's critical section execution is in progress. However, we may still want to guarantee mutual exclusion among the nonfaulty processes. We therefore assume that a malfunctioning process does not execute its critical section. (A malfunctioning process that executes

[1] Remember that setting a variable to its old value is not a "no-op", since a read that is concurrent with that operation may get the wrong value. If communication variables were set every time the process aborted, repeated abortions would be indistinguishable from the "malfunctioning" behavior considered below.

its *critical section* code is simply defined not to be executing its critical section.)

A malfunctioning process can also disrupt things by preventing nonfaulty processes from entering their critical sections. This is unavoidable, since a process that malfunctions after entering its critical section could leave its communication variables in a state indicating that it is still in the critical section. What we can hope to guarantee is that if the process stops malfunctioning, then the algorithm will resume its normal operation. This leaves two types of behavior to be considered, which differ in how a process stops malfunctioning.

The first type of failure allows a failed process to execute the following sequence of actions.

- It malfunctions for a while, arbitrarily changing the values of its communication variables.
- It then aborts—setting all its communication variables to some default values.
- It then resumes normal behavior, never again malfunctioning.

This behavior represents a situation in which a process fails, its failure is eventually detected and it is shut down, and the process is repaired and restored to service. The assumption that it never again malfunctions is discussed below.

Formally, this means that each process may perform at most one operation execution composed of the following sequence of executions (ordered by the $\longrightarrow$ relation):

- A *malfunction* execution, consisting of a finite collection of writes to its communication variables.
- A collection of writes that sets each communication variable to its default value.
- A noncritical section execution.

The above operation execution will be called a *failure*. If a property of a solution remains satisfied under this kind of behavior, then the solution is said to be *fail safe* for that property. Note that we do not assume failure to be detectable; one process cannot tell that another has failed (unless it can infer from the values of the other process's variables that a failure must have occurred).

The second type of failure we consider is one in which a process malfunctions, but eventually stops malfunctioning and resumes forever its normal behavior, starting in any arbitrary state. This behavior represents a transient fault.

If such a failure occurs, we cannot expect the system immediately to resume its normal operation. For example, the malfunctioning process might resume its normal operation just at the point where it is about to enter its critical section—while another process is executing its critical section. The most we can require is that after the process stops malfunctioning, the system *eventually* resumes its correct operation.

Since we are interested in the eventual operation, we need only consider what happens after every process has stopped malfunctioning. The state of the system at that time can be duplicated by starting all processes at arbitrary points in their program, with their variables having arbitrary values. In other words, we need only consider the behavior obtained by having each process do the following:

- Execute a *malfunction* operation.
- Then begin normal execution at any point in its program.

This kind of behavior will be called a *transient malfunction.* Any operation execution that is not part of the malfunction execution will be called a *normal* operation execution.

Unfortunately, deadlock freedom and lockout freedom cannot be achieved under this kind of transient malfunction behavior without a further assumption. To see why, suppose a malfunctioning process sets its communication variables to the values they should have while executing its critical section, and then begins normal execution with a nonterminating noncritical section execution. The process will always appear to the rest of the system as if it is executing its critical section, so no other process can ever execute its critical section.

To handle this kind of behavior, we must assume that a process executing its noncritical section will eventually set its communication variables to their default values. Therefore, we assume that instead of being elementary, the noncritical section executions are generated by the following program:

while ?
  do *abort* ;
    noncritical operation **od**

where the "?" denotes some unknown condition, which could cause the **while** to be executed forever, and every execution of the *noncritical operation* terminates. Recall that an *abort* execution sets certain communication variables to their default values if they are not already set to those values.

We now consider what it means for a property to hold "eventually". Intuitively, by "eventually" we mean "after some bounded period of time" following all the malfunctions. However, we have not introduced any concept of physical time. The only unit of time implicit in our formalism is the time needed to perform an operation execution. Therefore, we must define "after some bounded period of time" to mean "after some bounded number of operation executions". The definition we need is the following.

Definition 1 *A* system step *is an operation execution consisting of one normal elementary operation execution from every process. An operation execution A is said to occur* after t system steps *if there exist system steps $S_1, \ldots, S_t$ such that $S_1 \longrightarrow \cdots \longrightarrow S_t \longrightarrow A$.*

It is interesting to note that we could introduce a notion of time by defining the "time" at which an operation occurs to be the maximum t such that the operation occurs after t system steps (or 0 if there is no such t). Axioms A5 and A6 of Part I imply that this maximum always exists. Axiom A5 and the assumption that there are no nonterminating elementary operation executions imply that "time" increases without bound—i.e., there are operations occurring at arbitrarily large "times". Since we only need the concept of eventuality, we will not consider this way of defining "time".

We can now define what it means for a property to hold "eventually". Deadlock freedom and lockout freedom state that something eventually happens—for example, deadlock freedom states that so long as some process is executing its *trying* operation, then some process eventually executes its critical section. Since "eventually X eventually happens" is equivalent to "X eventually happens", requiring that these two properties eventually hold is the same as simply requiring that they hold.

We say that the mutual exclusion and FCFS properties eventually hold if they can be violated only for a bounded "length of time". Thus, the mutual exclusion property eventually holds if there is some t such that any two critical section executions $CS_i^{[k]}$ and $CS_j^{[m]}$ that both occur after t system steps are not concurrent. Similarly, the FCFS property holds eventually if it holds whenever both the *doorway* executions occur after t system steps. The value of t must be independent of the particular execution of the algorithm, but it may depend upon the number N of processes.

If a property eventually holds under the above type of transient malfunction behavior, then we say that the algorithm is *self-stabilizing* for that property. The concept of self-stabilization is due to Dijkstra [3].

Remarks On "Forever"

In our definition of failure, we could not allow a malfunctioning process to fail again after it had resumed its normal behavior, since repeated malfunctioning and recovery can be indistinguishable from continuous malfunctioning. However, if an algorithm satisfies any of our properties under the assumption that a process may malfunction only once, then it will also satisfy the property under repeated malfunctioning and recovery—so long as the process waits long enough before malfunctioning again.

The reason for this is that all our properties require that something either be true at all times (mutual exclusion, FCFS) or that something happen in the future (deadlock freedom, lockout freedom). If something remains true during a malfunction, then it will also remain true under repeated malfunctioning. If something must happen eventually, then because there is no "crystal ball" operation that can tell if a process will abort in the future,[2] another malfunction can occur after the required action has taken place. Therefore, an algorithm that is fail safe for such a property must also satisfy the property under repeated failure, if a failed process waits long enough before executing its *trying* statement again. Similar remarks apply to shutdown and transient malfunction.

3 The Solutions

We now present four solutions to the mutual exclusion problem. Each one is stronger than the preceding one in the sense that it satisfies more properties, and is more expensive in that it requires more communication variables.

[2] Such operations lead to logical contradictions—e.g., if one process executes "set x true if process i will abort", and process i executes "abort if x is never set true".

3.1 The Mutual Exclusion Protocol

We first describe the fundamental method for achieving mutual exclusion upon which all the solutions are based. Each process has a communication variable that acts as a synchronizing "flag". Mutual exclusion is guaranteed by the following protocol: in order to enter its critical section, a process must first set its flag true and then find every other process's flag to be false. The following result shows that this protocol does indeed ensure mutual exclusion, where v and w are communication variables, as defined in Part I, that represent the flags of two processes, and A and B represent executions of those processes' critical sections.

Theorem 2 *Let v and w be communication variables, and suppose that for some operation executions A and B and some k and m:*

- $V^{[k]} \longrightarrow \mathit{read}\ w = \mathit{false} \longrightarrow A$.
- $W^{[m]} \longrightarrow \mathit{read}\ v = \mathit{false} \longrightarrow B$.
- $v^{[k]} = w^{[m]} = \mathit{true}$.
- *If* $V^{[k+1]}$ *exists then* $A \longrightarrow V^{[k+1]}$.
- *If* $W^{[m+1]}$ *exists then* $B \longrightarrow W^{[m+1]}$.

Then A and B are not concurrent.

We first prove the following result, which will be used in the proof of the theorem. Its statement and proof use the formalism developed in Part I.

Lemma 1 *Let v be a communication variable and R a $\mathit{read}\ v = \mathit{false}$ operation such that:*

1. $v^{[k]} = \mathit{true}$
2. $V^{[k]} \dashrightarrow R$
3. $R \not\dashrightarrow V^{[k]}$

Then $V^{[k+1]}$ must exist and $V^{[k+1]} \dashrightarrow R$.

Proof: Intuitively, the assumptions mean that $V^{[k]}$ "effectively precedes" R, so R can't see any value written by a write that precedes $V^{[k]}$. Since R does not obtain the value written by $V^{[k]}$, it must be causally affected by a latter write operation $V^{[k+1]}$. We now formalize this reasoning.

By A3 and the assumption that the writes of v are totally ordered, hypothesis 2 implies that $V^{[i]} \dashrightarrow R$ for all $i \leq k$. If $R \dashrightarrow V^{[i]}$ for some $i < k$, then A3 would imply $R \dashrightarrow V^{[k]}$, contrary to hypothesis 3. Hence, we conclude that R is effectively nonconcurrent with $V^{[i]}$ for all $i \leq k$. If $V^{[k]}$ were the last write to v, hypothesis 2 and C2 would imply that R has to obtain the value *true*, contrary to hypothesis 1. Therefore, the operation $V^{[k+1]}$ must exist.

We now prove by contradiction that $V^{[k+1]} \dashrightarrow R$. Suppose to the contrary that $V^{[k+1]} \not\dashrightarrow R$. C3 then implies that $R \dashrightarrow V^{[k+1]}$ which, by A3 implies $R \dashrightarrow V^{[i]}$ for all $i \geq k+1$. A3 and the assumption that $V^{[k+1]} \not\dashrightarrow R$, implies that $V^{[i]} \not\dashrightarrow R$ for all $i \geq k+1$. Since we have already concluded that $V^{[i]} \dashrightarrow R$ for all $i \leq k$, C2 implies that R must obtain the value *true*, which contradicts hypothesis 1. This completes the proof that $V^{[k+1]} \dashrightarrow R$. ∎

Proof of Theorem: By C3, we have the following two possibilities:

1. $\textit{read } w = \textit{false} \dashrightarrow W^{[m]}$.
2. $W^{[m]} \dashrightarrow \textit{read } w = \textit{false}$.

(These are not disjoint possibilities.) We consider case 1 first. Combining 1 with the first two hypotheses of the theorem, we have

$$V^{[k]} \longrightarrow \textit{read } w = \textit{false} \dashrightarrow W^{[m]} \longrightarrow \textit{read } v = \textit{false}$$

By A4, this implies $V^{[k]} \longrightarrow \textit{read } v = \textit{false}$. A2 and the lemma then imply that $V^{[k+1]}$ exists and $V^{[k+1]} \dashrightarrow \textit{read } v = \textit{false}$. Combining this with the fourth and second hypotheses gives

$$A \longrightarrow V^{[k+1]} \dashrightarrow \textit{read } v = \textit{false} \longrightarrow B$$

By A4, this implies $A \longrightarrow B$, completing the proof in case 1.

We now consider case 2. Having already proved the theorem in case 1, we can make the additional assumption that case 1 does not hold, so $\textit{read } w = \textit{false} \not\dashrightarrow W^{[m]}$. We can then apply the lemma (substituting w for v and m for k) to conclude that $W^{[m+1]}$ exists and $W^{[m+1]} \dashrightarrow \textit{read } w = \textit{false}$. Combining this with the first and last hypotheses gives

$$B \longrightarrow W^{[m+1]} \dashrightarrow \textit{read } w = \textit{false} \longrightarrow A$$

```
private variable: j with range 1 ... N;
communication variable: x_i initially 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: The One-Bit Algorithm: Process i

A4 now implies $B \longrightarrow A$, proving the theorem for this case. ∎

We have written the proof of this theorem in full detail to show how A1–A4 and C0–C3 are used. In the remaining proofs, we will be more terse, leaving many of the details to the reader.

3.2 The One-Bit Solution

We now use the above protocol to obtain a mutual exclusion solution that requires only the single (one-bit) communication variable x_i for each process i. Obviously, no solution can work with fewer communication variables. This solution was also discovered independently by J. Burns. The algorithm for process i is shown in Figure 1, and its correctness properties are given by the following result.

Theorem 3 *The One-Bit Algorithm satisfies the mutual exclusion and deadlock freedom properties, and is shutdown safe and fail safe for these properties.*

Proof: To prove the mutual exclusion property, we observe that the above protocol is followed by the processes. More precisely, the mutual exclusion property is proved using Theorem 2, substituting x_i for v, x_j for w, $CS_i^{[k]}$ for A and $CS_j^{[k']}$ for B. This protocol is followed even under shutdown and failure behavior, so the algorithm is shutdown safe and fail safe for mutual exclusion. To prove deadlock freedom, we first prove the following lemma.

Lemma 2 *Any execution of the second* **for** *loop must terminate, even under shutdown and failure behavior.*

Proof: The proof is by contradiction. Let i be any process that executes a nonterminating second **for** loop. Then before entering the loop, i performs a finite number of *write* x_i executions, with the final one setting x_i true. We now prove by contradiction that every other process can also execute only a finite number of writes to its communication variable. Let k be the lowest-numbered process that performs an infinite number of *write* x_k executions. Process k executes statement l infinitely many times. Since every lower-numbered process j executes only a finite number of writes to x_j, A5, A2 and C2 imply that all but a finite number of reads of x_j by k must obtain its final value. For k to execute statement l infinitely many times (and not get trapped during an execution of the first **for** loop's **while** statement), this final value must be *false* for every $j < k$. This implies that k can execute its first **for** loop only finitely many times before it enters its second **for** loop. But since the final value of x_i is *true*, this means that $k < i$, and that k can execute its second **for** loop only finitely many times before being trapped forever in the "**while** x_i" statement in its second **for** loop. This contradicts the assumption that k performs an infinite number of *write* x_k executions.

We have thus proved that if the execution of the second **for** loop of any process is nonterminating, then every process can execute only a finite number of writes to its communication variable. The final value of a process's communication variable can be *true* only if the process executes its second **for** loop forever. Letting i be the highest-numbered process executing a nonterminating second **for** loop, so the final value of x_j is *false* for every $j > i$, we easily see that i must eventually exit this **for** loop, providing the required contradiction. Hence, every execution of the second **for** loop must eventually terminate. ∎

Proof of Theorem (continued): We now complete the proof of the theorem by showing that the One-Bit Algorithm is deadlock free. Assume that some process performs a nonterminating *trying* execution. Let i be the lowest numbered process that does not execute a nonterminating noncritical section. (There is at least one such process—the one performing the nonterminating *trying* execution.) Each lower numbered process j performs a nonterminating noncritical section execution after setting its communication variable false. (This is true for shutdown and failure behavior too.) It follows from A5, A2 and C2 that if i performs an infinite number of reads of the variable x_j, then all but a finite number of them must return the value *false*. This implies that every execution of the first **for** loop of process i must terminate. But, by the above lemma, every execution of its second **for** loop must also terminate. Since we have assumed that every execution of its noncritical section terminates, this implies that process i performs an infinite number of critical section executions, completing the proof of the theorem. ∎

The One-Bit Algorithm as written in Figure 1 is not self-stabilizing for mutual exclusion or deadlock freedom. It is easy to see that it is not self-stabilizing for deadlock freedom, since we could start all the processes in the **while** statement of the first **for** loop with their communication variables all true. It isn't self-stabilizing for mutual exclusion because a process could be started in its second **for** loop with its communication variable false, remain there arbitrarily long, waiting as higher numbered processes repeatedly execute their critical sections, and then execute its critical section while another process is also executing its critical section.

The One-Bit Algorithm is made self-stabilizing for both mutual exclusion and deadlock freedom by modifying each of the **while** loops so they read the value of x_i and correct its value if necessary. In other words, we place

$$\textbf{if } x_i \textbf{ then } x_i := \mathit{false}$$

in the body of the first **for** loop's **while** statement, and likewise for the second **for** loop (except setting x_i true there). We now prove that this modification makes the One-Bit Algorithm self-stabilizing for mutual exclusion and deadlock freedom.

Theorem 4 *With the above modification, the One-Bit Algorithm is self-stabilizing for mutual exclusion and deadlock freedom.*

Proof: It is easy to check that the proof of the deadlock freedom property in Theorem 3 is valid under the behavior assumed for self-stabilization, so the algorithm is self-stabilizing for deadlock freedom. To prove that it is self-stabilizing for mutual exclusion, we have to show that the mutual exclusion protocol is followed after a bounded number of system steps. It is easy to verify that this is true so long as every process that is in its second **for** loop (or past the point where it has decided to enter its second **for** loop) exits from that loop within a bounded number of system steps.[3] We prove this by "backwards induction" on the process number.

To start the induction, we observe that since its second **for** loop is empty, process N must exit that **for** loop within some bounded number $t(N)$ of system steps. To complete the induction step, we assume that if $j > i$ then process j must exit its second **for** loop within $t(j)$ system steps of when it entered, and we prove that if process i is in its second **for** loop (after the *malfunction*), then it must eventually exit. We define the following sets:

S_1: The set of processes waiting in their second **for** loop for a process numbered less than or equal to i.

S_2: The set of processes waiting in their first **for** loop for a process in S_1.

S_3: The set of processes in their *trying* statement.

If process i is in its second **for** loop, then within a bounded number of steps it either leaves that loop or else sets x_i true. In the latter case, no process that then enters its *trying* section can leave it before process i does. Each higher-numbered process that is in its second **for** loop must leave it in a bounded number of system steps, whereupon any other process that is in its second **for** loop past the test of x_i must exit that loop within a bounded number of system steps. It follows that within a bounded number of steps, if process i is still in its second **for** loop, then the system execution reaches a point at which each of the three sets S_m cannot get smaller until i sets x_i false. It is then easy to see that once this has happened, within a bounded number of steps, one of the following must occur:

- Process i exits its second **for** loop.
- Another process joins the set S_3.
- A process in S_3 joins S_1 or S_2.

Since there are only N processes, there is a bound on how many times the second two can occur. Therefore, the first possibility must occur within a bounded number of system steps, completing the proof that process i must

[3] In this proof, we talk about where a process is in its program immediately before and after a system step. This makes sense because a system step contains an elementary operation from every process.

exit its second **for** loop within a bounded number of system steps. This in turn completes the proof of the theorem. ∎

3.3 A Digression

Suppose N processes are arranged in a circle, with process 1 followed by process 2 followed by ... followed by process N, which is followed by process 1. Each process communicates only with its two neighbors using an array v of boolean variables, each $v(i)$ being owned by process i and read by the following process. We want the processes to continue forever taking turns performing some action—first process 1, then process 2, and so on. Each process must be able to tell whether it is its turn by reading just its own variable $v(i)$ and that of the preceding process, and must pass the turn on to the next process by complementing the value of $v(i)$ (which is the only change it can make).

The basic idea is to let it be process i's turn if the circle of variables $v(1)$, ... , $v(N)$ changes value at i—that is, if $v(i) = \neg v(i-1)$. This doesn't quite work because a ring of values cannot change at only one point. However, we let process 1 be exceptional, letting it be 1's turn when $v(1) = v(N)$. The reader should convince himself that this works if all the $v(i)$ are initially equal.

It is obvious how this algorithm, which works for the cycle of all N processes arranged in order, is generalized to handle an arbitrary cycle of processes with one process singled out as the "first". To describe the general algorithm more formally, we need to introduce some notation. Recall that a *cycle* is an object of the form $\langle i_1, \ldots, i_m \rangle$, where the i_j are distinct integers between 1 and N. The i_j are called the *elements* of the cycle. Two cycles are the same if they are identical except for a cyclic permutation of their elements—e.g., $\langle 1, 3, 5, 7 \rangle$ and $\langle 5, 7, 1, 3 \rangle$ are two representations of the same cycle, which is not the same cycle as $\langle 1, 5, 3, 7 \rangle$. We define the *first element* of a cycle to be its smallest element i_j.

By a *boolean function on a cycle* we mean a boolean function on its set of elements. The following functions are used in the remaining algorithms. Note that $CG(v, \gamma, i)$ is the boolean function that has the value *true* if and only if it is process i's turn to go next in the general algorithm applied to the cycle γ of processes.

Definition 2 *Let v be a boolean function on the cycle $\gamma = \langle i_1, \ldots, i_m \rangle$, and* *let i_1 be the first element of γ. For each element i_j of the cycle we define:*

$$\begin{array}{lcll} CGV(v,\gamma,i_j) & \stackrel{\text{def}}{\equiv} & \neg v(i_{j-1}) & \text{if } j > 1 \\ & & v(i_m) & \text{if } j = 1. \\ CG(v,\gamma,i_j) & \stackrel{\text{def}}{\equiv} & v(i_j) \equiv CGV(v,\gamma,i_j) & \end{array}$$

If $CG(v,\gamma,i_j)$ is true, then we say that v changes value *at i_j.*

The turn-taking algorithm, in which process i takes its turn when $CG(v,\gamma,i)$ equals *true*, works right when it is started with all the $v(i)$ having the same value. If it is started with arbitrary initial values for the $v(i)$, then several processes may be able to go at the same time. However, deadlock is impossible; it is always at least one process's turn. This is expressed formally by the following result, whose proof is simple and is left to the reader.

Lemma 3 *Every boolean function on a cycle changes value at some element—i.e., for any boolean function v on a cycle γ, there is some element i of γ such that $CG(v,\gamma,i) = true$.*

We will also need the following definitions. A cycle $\langle i_1,\ldots,i_m\rangle$ is said to be *ordered* if, after a cyclic permutation of the i_j, $i_1 < \ldots < i_m$. For example, $\langle 5,7,2,3\rangle$ is an ordered cycle while $\langle 2,5,3,7\rangle$ is not. Any nonempty set of integers in the range 1 to N defines a unique ordered cycle. If S is such a set, then we let ORD S denote this ordered cycle.

3.4 The Three-Bit Algorithm

The One-Bit algorithm has the property that the lowest-numbered process that is trying to enter its critical section must eventually enter it—unless a still lower-numbered process enters the trying region before it can do so. However, a higher-numbered process can be locked out forever by lower-numbered processes repeatedly entering their critical sections. The basic idea of the Three-Bit Algorithm is for the processes' numbers to change in such a way that a waiting process must eventually become the lowest-numbered process.

Of course, we don't actually change a process's number. Rather, we modify the algorithm so that instead of process i's two **for** loops running from 1 up to (but excluding) i and $i+1$ to N, they run cyclically from f up to but excluding i and from $i \oplus 1$ up to but excluding f, where f is a function of the communication variables, and $\oplus$ denotes addition modulo N. As processes pass through their critical sections, they change the value of their communication variables in such a way as to make sure that f eventually equals the number of every waiting process.

The first problem that we face in doing this is that if we simply replaced the **for** loops as indicated above, a process could be waiting inside its first **for** loop without ever discovering that f should have been changed. Therefore, we modify the algorithm so that when it finds x_j true, instead of waiting for it to become false, process i recomputes f and restarts its cycle of examining all the processes' communication variables.

We add two new communication variables to each process i. The variable y_i is set true by process i immediately upon entering its trying section, and is not set false until after process i has left its critical section and set x_i false. The variable z_i is complemented when process i leaves its critical section.

Finally, it is necessary to guarantee that while process i is in its trying region, a "lower-numbered" process that enters after i cannot enter its critical section before i does. This is accomplished by having the "lower-numbered" process wait for y_i to become false instead of x_i. This will still insure mutual exclusion, since x_i is false whenever y_i is.

Putting these changes all together, we get the algorithm of Figure 2. The "**for** $j := \ldots$ **cyclically to** $\ldots$" denotes an iteration starting with j equal to the lower bound, and incrementing j by 1 modulo N up to but excluding the upper bound. We let $*\!:=\!*$ denote an assignment operation that performs a write only if it will change the value of the variable—i.e., the right-hand side is evaluated, compared with the current value of the variable on the left-hand side, and an assignment performed only if they are unequal. The $*\!:=\!*$ operation is introduced because we have assumed nothing about the value obtained by a read that is concurrent with a write, even if the write does not change the value. For example, executing $v := \mathit{true}$ when v has the value *true* can cause a concurrent read of v to obtain the value *false*. However, executing $v *\!:=\!* \mathit{true}$ has absolutely no effect if v has the value *true*.

We let z denote the function that assigns the value z_j to j—so evaluating it at j requires a read of the variable z_j. Thus, $CG(z, \langle 1, 3, 5\rangle, 3) = \mathit{true}$ if and only if $z_1 \neq z_3$. Note that i is always an element of the cycle γ computed by process i, so the cycle is nonempty and the argument of the minimum function is a nonempty set (by Lemma 3).

We now prove that this algorithm satisfies the desired properties. In this and subsequent proofs, we will reason informally about processes looping and things happening eventually. The reader can refer to the proof of Theorem 3

private variables: j, f **with range** $1 \ldots N$,
γ **with range cycles on** $1 \ldots N$;
communication variables: x_i, y_i **initially** *false*, z_i;
repeat forever
noncritical section;
$y_i := true$;
$l1$: $x_i := true$;
$l2$: $\gamma := \text{ORD}\{i : y_i = true\}$
$f :=$ minimum $\{j \in \gamma : CG(z, \gamma, j) = true\}$;
for $j := f$ **cyclically to** i
do **if** y_j **then** $x_i *:=* false$;
goto $l2$
fi
od;
if $\neg x_i$ **then goto** $l1$ **fi**;
for $j := i \oplus 1$ **cyclically to** f
do **if** x_j **then goto** $l2$ **fi** **od**;
critical section;
$z_i := \neg z_i$;
$x_i := false$;
$y_i := false$
end repeat

Figure 2: The Three-Bit Algorithm: Process i

to see how these arguments can be made more formal.

Theorem 5 *The Three-Bit Algorithm satisfies the mutual exclusion, deadlock freedom and lockout freedom properties, and is shutdown safe and fail safe for them.*

Proof: To verify the mutual exclusion property, we need only check that the basic mutual exclusion protocol is observed. This is not immediately obvious, since process i tests either x_j or y_j before entering its critical section, depending upon the value of f. However, a little thought will show that processes i and j do indeed follow the protocol before entering their critical sections, process i reading either x_j or y_j, and process j reading either x_i or y_i. This is true for the behavior allowed under shutdown and failure safety, so the algorithm is shutdown safe and fail safe for mutual exclusion.

To prove deadlock freedom, assume for the sake of contradiction that some process executes a nonterminating *trying* statement, and that no process performs an infinite number of critical section executions. Then eventually there must be some set of processes looping forever in their *trying* statements, and all other processes forever executing their noncritical sections with their x and y variables false. Moreover, all the "trying" processes will eventually obtain the same value for f. Ordering the process numbers cyclically starting with f, let i be the lowest-numbered trying process. It is easy to see that all trying processes other than i will eventually set their x variables false, and i will eventually enter its critical section, providing the necessary contradiction.

We now show that the algorithm is lockout free. There must be a time at which one of the following three conditions is true for every process:

- It will execute its critical section infinitely many times.
- It is and will remain forever in its *trying* statement.
- It is and will remain forever in its noncritical section.

Suppose that this time has been reached, and let $\beta = \langle j_1, \ldots, j_p \rangle$ be the ordered cycle formed from the set of processes for which one of the first two conditions holds. Note that we are not assuming j_1 to be the first element (smallest j_i) of β. We prove by contradiction that no process can remain forever in its trying section.

Suppose j_1 remains forever in its trying section. If j_2 were to execute its critical section infinitely many times, then it would eventually enter its *trying* section with z_{j_2} equal to $\neg CGV(z, \beta, j_2)$. When process j_2 then executes its statement $l2$, the cycle γ it computes will include the element j_1, and it will compute $CG(z, \gamma, j_2)$ to equal *false*. It is easy to see that the value of f that j_2 then computes will cause j_1 to lie in the index range of its first **for** loop, so it must wait forever for process j_1 to set y_{j_1} false.

We therefore see that if j_1 remains forever in its trying section, then j_2 must also remain forever in its trying section. Since this is true for any element j_1 in the cycle β (we did not assume j_1 to be the first element), a simple induction argument shows that if any process remains forever in its trying section, then all the processes $j_1, \ldots, j_p$ must remain forever in their trying sections. But this means that the system is deadlocked, which we have shown to be impossible, giving the required contradiction.

The above argument remains valid under shutdown and failure behavior, so the algorithm is shutdown safe and fail safe for lockout freedom. ∎

As with the One-Bit Algorithm, we must modify the Three-Bit Algorithm in order to make it self-stabilizing. It is necessary to make sure that process i does not wait in its *trying* section with y_i false. We therefore need to add the statement y_i $*{:=}*$ *true* somewhere between the label $l2$ and the beginning of the **for** statement. It is not necessary to correct the value of x_i because that happens automatically, and the initial value of z_i does not matter. We then have the following result.

Theorem 6 *The Three-Bit Algorithm, with the above modification, is self-stabilizing for the mutual exclusion, deadlock freedom and lockout freedom properties.*

Proof: Within a bounded number of system steps, each process will either have passed through point $l2$ of its program twice, or entered its noncritical section and reset its x and y variables. (Remember that for self-stabilization, we must assume that these variables are reset in the noncritical section if they have the value *true*.) After that has happened, the system will be in a state it could have reached starting at the beginning from a normal initial state. ∎

3.5 FCFS Solutions

We now describe two FCFS solutions. Both of them combine a mutual exclusion algorithm ME that is deadlock free but not FCFS with an algorithm

FC that does not provide a mutual exclusion but does guarantee FCFS entry to its "critical section". The mutual exclusion algorithm is embedded within the FCFS algorithm as follows.

repeat forever
  noncritical section;
  FC trying;
  FC critical section: **begin**
    ME trying;
    ME critical section;
    ME exit
  end;
  FC exit
end repeat

It is obvious that the entire algorithm satisfies the FCFS and mutual exclusion properties, where its doorway is the FC algorithm's doorway. Moreover, if both FC and ME are deadlock free, then the entire algorithm is also deadlock free. This is also true under shutdown and failure . Hence, if FC is shutdown safe (or fail safe) for FCFS and deadlock freedom, and ME is shutdown safe (fail safe) for mutual exclusion and deadlock freedom, then the entire algorithm is shutdown safe (fail safe) for FCFS, mutual exclusion and deadlock freedom.

We can let ME be the One-Bit Algorithm, so we need only look for algorithms that are FCFS and deadlock free. The first one is the N-Bit Algorithm of Figure 3, which is a modification of an algorithm due to Katseff [5]. It uses N communication variables for each process. However, each of the $N-1$ variables $z_i[j]$ of process i is read only by process j. Hence, the complete mutual exclusion algorithm using it and the One-Bit Algorithm requires the same number of single-reader variables as the Three-Bit Algorithm. The "**for all** j" statement executes its body once for each of the indicated values of j, with the separate executions done in any order (or interleaved). The function z_{ij} on the cycle ORD$\{i,j\}$ is defined by:

$$
\begin{aligned}
z_{ij}(i) &\stackrel{\text{def}}{=} z_i[j] \\
z_{ij}(j) &\stackrel{\text{def}}{=} z_j[i]
\end{aligned}
$$

We now prove the following properties of this algorithm.

Lemma 4 *The N-Bit Algorithm satisfies the FCFS and Deadlock Freedom properties, and is shutdown safe, abortion safe and fail safe for them.*

communication variables:
  y_i **initially** *false*,
  array z_i **indexed by** $\{1 \ldots N\} - \{i\}$;
private variables:
  array *after* **indexed by** $\{1 \ldots N\} - \{i\}$ **of boolean**,
  j **with range** $1 \ldots N$;
repeat forever
  noncritical section;
  doorway: **for all** $j \neq i$
        do $z_i[j] *{:=}* \neg CGV(z_{ij}, \mathrm{ORD}\{i,j\}, i)$ **od**;
      for all $j \neq i$
        do $after[j] := y_j$ **od** ;
      $y_i := true$;
  waiting: **for all** $j \neq i$
        do while $after[j]$
          do if $CG(z_{ij}, \mathrm{ORD}\{i,j\}, i) \vee \neg y_j$
            then $after[j] := false$ **fi od**
        od;
  critical section;
  $y_i := false$
end repeat

Figure 3: The N-Bit FCFS Algorithm: Process i

Proof: Informally, the FCFS property is satisfied because if process i finishes its doorway before process j enters its doorway, but i has not yet exited, then j must see y_i true and wait for i to reset y_i or change $z_i[j]$. This argument is formalized as follows.

Assume that $\mathit{doorway}_i^{[k]} \longrightarrow \mathit{doorway}_j^{[m]}$. Let $Y_i^{[k']}$ denote the write operation of y_i performed during $\mathit{doorway}_i^{[k]}$, let $Z_i[j]^{[k'']}$ be the last write of $z_i[j]$ performed before $Y_i^{[k']}$.

We suppose that $CS_j^{[m]}$ exists, but that $CS_i^{[k]} \not\longrightarrow CS_j^{[m]}$, and derive a contradiction. Let R be any read of y_i performed by process j during $\mathit{trying}_j^{[m]}$. Since $\mathit{doorway}_i^{[k]} \longrightarrow \mathit{doorway}_j^{[m]}$, we have $Y_i^{[k']} \longrightarrow R$. Since $CS_i^{[k]} \not\longrightarrow CS_j^{[m]}$, A4 implies that $Y_i^{[k'+1]} \not\dashrightarrow R$. It then follows from C2 that the read R must obtain the value $y_i^{[k']}$, which equals *true*. A similar argument shows that every read of $z_i[j]$ during $\mathit{trying}_j^{[m]}$ obtains the value $z_i[j]^{[k'']}$. It is then easy to see that process j sets *after*$[i]$ true in its doorway and can never set it false because it always reads the same value of $z_i[j]$ in its *waiting* statement as it did in its *doorway*. Hence, j can never exit from its waiting section, which is the required contradiction.

We next prove deadlock freedom. The only way deadlock could occur is for there to be a cycle $\langle i_1, \ldots, i_m \rangle$ of processes, each one waiting for the preceding one—i.e., with each process $i_{j\oplus 1}$ having *after*$[i_j]$ true. We assume that this is the case and obtain a contradiction. Let Ry_j denote the read of $y_{i_{j\ominus 1}}$ and let Wy_j denote the write of y_{i_j} by process i_j in the last execution of its doorway. Since $Ry_j \longrightarrow Wy_j$ and the relation $\longrightarrow$ is acyclic, by A2 and A4 there must be some j such that $Wy_j \not\dashrightarrow Ry_{j\oplus 1}$. By C3, this implies that $Ry_{j\oplus 1} \dashrightarrow Wy_j$.

Let Wy' be the write of y_{i_j} that immediately precedes Wy_j, and thus sets its value false. If Wy' did not exist (because Wy_j was the first write of y_{i_j}) or $Ry_{j\oplus 1} \not\dashrightarrow Wy'$, it would follow from C2 and C3 that $Ry_{j\oplus 1}$ obtains the value *false*. But this is impossible because process $i_{j\oplus 1}$ has set *after*$[i_j]$ true. Hence, there is such a Wy' and $Ry_{j\oplus 1} \dashrightarrow Wy'$.

Using this result and A4, it is easy to check that the last write of $z_{i_{j\oplus 1}}[i_j]$ (during the last execution of the doorway of process $i_{j\oplus 1}$) must have preceded the reading of it by process i_j during the last execution of its doorway. It follows from this that in the deadlock state, $CG(z_{i_{j\oplus 1}}[i_j], \mathrm{ORD}\{i_j, i_{j\oplus 1}\}, i_{j\oplus 1})$ must be true, contradicting the assumption that $i_{j\oplus 1}$ is waiting forever with *after*$[i_j]$ true. This completes the proof of deadlock freedom.

We leave it to the reader to verify that the above proofs remain valid under shutdown, abortion and failure behavior. The only nontrivial part of the proof is showing that the algorithm is abortion safe for deadlock freedom. This property follows from the observation that if no process enters its critical section, then eventually all the values of $z_i[j]$ will stabilize and no more writes to those variables will occur—even if there are infinitely many abortions. ∎

Using this lemma and the preceding remarks about embedding a mutual exclusion algorithm inside a FCFS algorithm, we can prove the following result.

Theorem 7 *Embedding the One-Bit Algorithm inside the N-Bit Algorithm yields an algorithm that satisfies the mutual exclusion, FCFS, deadlock freedom and lockout freedom properties, and is shutdown safe, abortion safe and fail safe for these properties.*

Proof: As we remarked above, the proof of the mutual exclusion, FCFS and deadlock freedom properties is trivial. Lockout freedom follows from these by Theorem 1. The fact that it is shutdown safe and fail safe for these properties follows from the fact that the One-Bit and N-Bit algorithms are. The only thing left to show is that the entire algorithm is abortion safe for these properties even though the One-Bit algorithm is not. The FCFS property for the outer N-Bit algorithm implies that once a process has aborted, it cannot enter the One-Bit algorithm's *trying* statement until all the processes that were waiting there have either exited from the critical section or aborted. Hence, so far as the inner One-Bit algorithm is concerned, abortion is the same as shutdown until there are no more waiting processes. The shutdown safety of the One-Bit Algorithm therefore implies the abortion safety for the entire algorithm. ∎

The above algorithm satisfies all of our conditions except for self-stabilization. It is not self-stabilizing for deadlock freedom because it is possible to start the algorithm in a state with a cycle of processes each waiting for the next. (The fact that this cannot happen in normal operation is due to the precise order in which variables are read and written.) In our final algorithm, we modify the N-Bit Algorithm to eliminate this possibility.

In the N-Bit Algorithm, process i waits for process j so long as the function z_{ij} on the cycle $\text{ORD}\{i, j\}$ does not change value at i. Since a function must change value at some element of a cycle, this prevents i and j

communication variables:
y_i **initially** *false*,
array z_i **indexed by** $CYC(i)$;
private variables:
j **with range** $1 \ldots N$,
γ **with range** $CYC(i)$,
array *after* **indexed by** $1 \ldots N$ **of booleans**;
repeat forever

```
    noncritical section;
    doorway: for all γ ∈ CYC(i) do
                 z_i[γ] ∗:=∗ ¬CGV(z_γ, γ, j) od;
             for all j ≠ i
                 do after[j] := y_j od;
    waiting: for all j ≠ i
                 do while after[j]
                        do after[j] := y_j;
                           for all γ ∈ CYC(j, i)
                               do if ¬CG(z_γ, γ, i)
                                      then after[j] := false fi od
                        od
                 od;
    critical section;
    y_i := false
end repeat
```

Figure 4: The N!-Bit FCFS Algorithm: Process i.

from waiting for each other. However, it does not prevent a cycle of waiting processes containing more than two elements. We therefore introduce a function z_γ for every cycle γ, and we require that i wait for j only if for every cycle γ in which j precedes i: z_γ does not change value at i. It is easy to see that for any state, there can be no cycle γ in which each process waits for the preceding one, since z_γ must change value at some element of γ.

This leads us to the N!-Bit Algorithm of Figure 4. We use the notation that $CYC(i)$ denotes the set of all cycles containing i and at least one other element, and $CYC(j, i)$ denotes the set of all those cycles in which j precedes i. We let z_γ denote the function on the cycle γ that assigns the value $z_i[\gamma]$ to the element i.

Using the $N!$-Bit FCFS Algorithm, we can construct the "ultimate" algorithm that satisfies every property we have ever wanted from a mutual exclusion solution, as stated by the following theorem. Unfortunately, as the reader has no doubt noticed, this solution requires approximately $N!$ communication variables for each process, making it of little practical interest except for very small values of N.

Theorem 8 *Embedding the One-Bit Algorithm inside the $N!$-Bit Algorithm yields an algorithm that satisfies the mutual exclusion, FCFS, deadlock freedom and lockout freedom properties, and is shutdown safe, abortion safe, fail safe and self-stabilizing for these properties.*

Proof: The proof of all but the self-stabilizing condition is the same as for the previous solution using the N-Bit Algorithm. It is easy to see that since the One-Bit algorithm is self-stabilizing for mutual exclusion and deadlock freedom, to prove self-stabilization for the entire algorithm it suffices to prove that the $N!$-Bit Algorithm is self-stabilizing for deadlock freedom. The proof of that is easily done using the above argument that there cannot be a cycle of processes each waiting endlessly for the preceding one. ∎

4 Conclusion

Using the formalism of Part I, we stated the mutual exclusion problem, as well as several additional properties we might want a solution to satisfy. We then gave four algorithms, ranging from the inexpensive One-Bit Algorithm that satisfies only the most basic requirements to the ridiculously costly $N!$-Bit Algorithm that satisfies every property we have ever wanted of a solution.

Our proofs have been done in the style of standard "journal mathematics", using informal reasoning that in principle can be reduced to very formal logic, but in practise never is. Our experience in years of devising synchronization algorithms has been that this style of proof is quite unreliable. We have on several occasions "proved" the correctness of synchronization algorithms only to discover later that they were incorrect. (Everyone working in this field seems to have the same experience.) This is especially true of algorithms using our nonatomic communication primitives.

This experience led us to develop a reliable method for proving properties of concurrent programs [7], [11], [16]. Instead of reasoning about a program's behavior, one reasons in terms of its state. When the first version of the present paper was written, it was not possible to apply this method to these mutual exclusion algorithms for the following reasons:

- The proof method required that the program be described in terms of atomic operations; we did not know how to reason about the nonatomic reads and writes used by the algorithms.
- Most of the correctness properties to be proved, as well as the properties assumed of the communication variables, were stated in terms of the program's behavior; we did not know how to apply our state-based reasoning to such behavioral properties.

Recent progress in reasoning about nonatomic operations [12] and in temporal logic specifications [13, 15] should make it possible to recast our definitions and proofs in this formalism. However, doing so would be a major undertaking, completely beyond the scope of this paper. We are therefore forced to leave these proofs in their current form as traditional, informal proofs. The behavioral reasoning used in our correctness proofs, and in most other published correctness proofs of concurrent algorithms, is inherently unreliable; we advise the reader to be skeptical of such proofs.

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, Irene Greif, and Michael Fischer. The impetus finally to write down the results came from discussions with Michael Rabin in 1980 that led to the discovery of the Three-Bit Algorithm.

References

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

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

[3] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[4] M. J. Fischer et al. *Resource Allocation with Immunity to Limited Process Failure*. Technical Report 79-09-01, Department of Computer Science, University of Washington, September 1979.

[5] H. P. Katseff. A solution to the critical section problem with a totally wait-free fifo doorway. 1978. Internal Memorandum, Computer Science Division, University of California, Berkeley.

[6] D. E. Knuth. Additional commments on a problem in concurrent program control. *Communications of the ACM*, 9(5):321, May 1966.

[7] Leslie Lamport. The 'Hoare logic' of concurrent programs. *Acta Informatica*, 14(1):21–37, 1980.

[8] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[9] Leslie Lamport. The mutual exclusion problem—part i: a theory of interprocess communication. To appear in *JACM*.

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

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

[12] Leslie Lamport. Reasoning about nonatomic operations. In *Proceedings of the Tenth Annual Symposium on Principles of Programming Languages*, pages 28–37, ACM SIGACT-SIGPLAN, January 1983.

[13] Leslie Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[14] Leslie Lamport. The synchronization of independent processes. *Acta Informatica*, 7(1):15–34, 1976.

[15] Leslie Lamport. What good is temporal logic? In R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, IFIP, North Holland, Paris, September 1983.

[16] Susan Owicki and Leslie Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[17] 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.

[18] Gary L. Peterson. A new solution to lamport's concurrent programming problem. *ACM Transactions on Programming Languages and Systems*, 5(1):56–65, January 1983.

[19] Michael Rabin. The choice coordination problem. *Acta Informatica*, 17:121–134, 1982.

[20] 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.