LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-84

THE MUTUAL EXCLUSION PROBLEM FOR UNRELIABLE PROCESSES

RONALD L. RIVEST
VAUGHAN R. PRATT

APRIL 1977

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TM-84

THE MUTUAL EXCLUSION PROBLEM FOR UNRELIABLE PROCESSES

Ronald L. Rivest
Vaughan R. Pratt

April 1977

MIT/LCS/TM-84

The Mutual Exclusion Problem for Unreliable Processes

Ronald L. Rivest
Vaughan R. Pratt

April 1977

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science
(formerly Project MAC)
CAMBRIDGE MASSACHUSETTS 02139

TM-84

The Mutual Exclusion Problem for Unreliable Processes

Ronald L. Rivest and Vaughan R. Pratt
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Consider n processes operating asynchronously in parallel, each of which maintains a single "public" variable which can be read (but not written) by the other processes. We show that the processes can synchronize their actions by the basic operations of (1) reading each other's public variables, and (2) setting their own public variable to some value. A process may "die" (fail) at any time, when its public variable is (automatically) set to a special "dead" value. A dead process may revive. Reading a public variable which is being simultaneously updated returns either the old or the new value.

Each process may be in a certain "critical" state (which it leaves if it dies). We present a synchronization scheme with the following properties.

(1) At most one process is ever in its critical state at a time.

(2) If a process wants to enter its critical state, it may do so before any other process enters its critical state more than once.

(3) The public variables assume only a finite number of values.

(4) A process wanting to enter its critical state can always make progress towards that goal.

(5) The various processes may run at arbitrary speeds relative to one another.

By the definition of the problem, no process can prevent another from entering its critical state by repeatedly failing and restarting.

In the case of two processes, what makes our solution of particular interest is its remarkable simplicity when compared with the extant solutions to this problem. Our n-process solution uses the two-process solution as a subroutine, and is not quite as elegant as the two-process solution.

This research was supported by the National Science Foundation under contracts MCS76-14294, MCS76-18461 and MCS74-12997 A04.

I. Introduction

In this introduction we first make precise the model of parallel computation that we are using, and then describe the problem to be solved.

We consider n processes P_i, for $0 \le i < n$, operating asynchronously in parallel. The number n fixed and is known to every process. Therefore processes may not leave the system, although they may "die" (fail) as explained later. The processes act independently in that any action possible for a process may be performed at any time; none of a process's basic operations depend on the states or actions of other processes for their successful completion. (Thus our model differs from one in which Dijkstra's P and V coordinating primitives are used.)

Each process P_i conveys information to the other processes by means of a single "public" variable S_i which may be changed only by P_i, but it may be read at any time by any other process. If S_i is simultaneously changed by P_i and read by some other process P_j, then P_j will see either the old or the new value. More precisely, we assume that any set of reads and writes of a single public variable (and more generally the entire set of reads and writes of all the public variables during a particular computation) have the effect of being executed sequentially in some order. All communication between processes will be accomplished by means of the public variables; the only information that P_i can obtain about P_j is the value of S_j.

We make no assumptions about the relative speeds of the various processes, but we do require that every active process make progress, no matter how quickly or slowly. A process may run at an arbitrary positive (even time-varying) speed, but it may only halt by "dying" as explained later.

The most remarkable feature of the coordination scheme to be presented is that it continues to work correctly in spite of the failure (or even repeated failure and restarting) of any subset of the processes. The scheme is therefore "ultrareliable" in that

those processes still functioning reliable are able to effectively coordinate their activities.

The preceding specifies the nature of the parallel system with which we are working. Now we turn our attention to the specification of the problem to be solved.

A solution to Dijkstra's mutual exclusion problem for n processes [2,4] is a set of n protocols, one per process, which may be executed by their owners at any time. Each process is assumed to contain a critical section, which is a piece of the program that requires for its correct execution that no other processes be simultaneously in their critical section. Typically the critical section involves some resource (e.g a printer, operating systems table, or satellite communications channel), which can only be meaningfully used by one user at a time. Whenever the process desires to use this nonsharable resource it first must execute its protocol in order to then enter its critical section with the assurance that no other process is also in its critical section. The only action we require of a process when it leaves its critical section is that it set is public variable to the special value D (for "done" or "dead") before reexecuting its protocol. Thus a solution must have the following property.

> Mutual exclusion: No two processes may be in their critical sections at the same time.

Dijkstra's solution [2] also satisfies the obviously desirable criterion:

> No deadlock: It is not possible for all processes to become simultaneously blocked in such a fashion that none of them will be able to enter their critical sections.

Unfortunately, Dijkstra's solution does not have the following property (as pointed out by Knuth [6]):

> No lockout: It is not possible for an individual process to be kept forever from entering its critical section by some (perhaps highly improbable) sequence of actions by the other processes.

In other words, a particular way of interleaving the basic operations was able to "lock out" an individual process. Note that "no lockout" implies "no deadlock". Knuth presented a new solution which also has "no lockout": a process can always enter its critical section before the other n-1 processes can execute their critical sections (collectively) more than 2^{n-1} times. A modified procedure by de Bruijn [1] reduced this figure to $\binom{n}{2}$. Finally, Eisenberg and McGuire [5] constructed a coordination scheme satisfying "mutual exclusion,""no lockout," and

> Linear waiting: There is a constant ℓ_0 such that a process that is waiting to enter its critical section will at worst have to wait for every other process to take ℓ_0 turns (in its critical section). (In fact, Eisenberg and Mcguire's solution had $\ell_0=1$.)

All of the above solutions employ a global variable that may be set by every process. This has the drawback that if the memory unit containing the global variable should fail, the entire system breaks down. While this is not serious if the processes themselves are relying on this common memory for storage of program and data, as in a typical multiprocessing system, it is undesirable in a multicomputer (distributed) system. Leslie Lamport [7] invented a "mutual exclusion,"no lockout,"linear waiting" system using only variables local to each process (the public variables of our model); each process may set only its own public variable, but may read the public variables of any other process. He assumes that if a process (or the memory unit containing its public variable) should fail, its public variable may assume arbitrary values for a period of time, but eventually must give a value of zero (until of course the process is restarted). The failure of any given process does not disable the system. We extract from the above discussion a property that our solution will satisfy:

> Public variables: Coordination is achieved by use of public variables only. (No global variables will be used.)

A related "distributed control" problem is studied by Dijkstra in [3].

Lamport's solution still suffers from two drawbacks, in that it does not have the following properties:

Finite ranges: The public variables can only assume a finite number of values.

Ultrareliability: A process (or set of processes) cannot deadlock the rest of the system by repeatedly failing and restarting.

In order to make the definition of "ultrareliability" precise, we need to specify what "failure" means. In the real world a device may fail, or die, in any one of three fashions. For example, it may simply cease to function in such a manner that the only way to tell that it has failed is to notice that one can not obtain a response from it. Or it may fail by continuing to act but in such a manner that it behaves as if it were some different device; it ceases to lie within its design parameters. Or, finally, it may fail in a disciplined fashion by simply stopping and posting a flag that it has failed. We show that only the third mode of failure permits the solution of our problem.

The first mode of failure (dying without any symptoms) is incompatible with our model. Since we make no assumptions about the relative speeds of the various processes, P_i cannot conclude that P_j has died, merely by noticing that S_j has not changed value in a long time. (P_i can not distinguish between the death of P_j and a delay due to P_j executing a very long internal computation.)

Similarly, the second mode of failure creates problems. In particular, if processes may misbehave in an arbitrary fashion, the mutual exclusion problem becomes insoluble since a process could refuse to alter its public variable when it leaves its critical section, causing the other processes to be locked out.

Therefore, we require that a process P_i fail only in the following disciplined manner (the third mode of failure): it must leave its critical section if it is in it, set its public variable S_i to the "dead" value D, and then stop. This is the only way a

process may stop or fail. (It is necessary to assume that when a process fails its public variable does not assume arbitrary values for a period of time, but rather changes directly from its previous value to the special "dead" value D. Otherwise a process that repeatedly failed and restarted could have a random value in its public variable whenever another process was reading it.) A process which is not dead (i.e. has $S_i \neq D$) is said to be active.

Although we do not explore the possibility in this paper, it seems plausible that the solutions we present could be modified to incorporate "symptomless dying" by changing our assumption that the processes run at arbitrary speeds. In this model the equivalent of reading the value D in a public variable could be obtained by "timing out" another process (i.e. waiting a fixed amount of time for a change in the value of its public variable - if none occurs the process is assumed to be dead).

It is worth pointing out here that while we are free to specify the protocol a process must execute before it can execute its critical section, it is not possible for us to require that a process execute some corresponding "exit protocol" after executing its critical section which is different from the procedure for failing (setting S_i to D). This is because a process may fail at any time, such as after executing its critical section but before executing this "exit protocol." Therefore we only require that a process P_i set S_i to D when exiting its critical section.

A solution using only public variables permits even rather widely separated systems to coordinate their activities in a simple fashion. Assuming that each pair of processes are directly connected, a process that changes its public variable need delay just enough to let the new value propagate to each other process. In a message-passing network a read could be implemented as a message sent from the process which is doing the reading to the process which owns the public variable, who in turn replies with a message giving the value of his variable (or an automatically generated message indicating that he has failed).

For the sake of precision we give a formal description of on the model of

parallelism we employ. The assignments, functions and tests used in the processes and their protocols will be abstracted as functions from system states to system states. Although our final solutions impose considerable structure on the state of each process, we shall be content here to stipulate structure of two "registers," separating the public and private components of a process state.

A process state $\sigma_i = (R_i, S_i)$ is a pair of values chosen from finite sets $\underline{R}$ and $\underline{S}$ respectively. Here S_i denotes P_i's public variable, and R_i denotes the other "invisible" components of P_i's state, such as P_i's program counter, local variables, etc. Let $R(\sigma)$ and $S(\sigma)$ denote the components of a process state σ. A system state is an n-tuple $\Sigma=(\sigma_0,\sigma_1,\ldots,\sigma_{n-1})$ of process states; we let $R(\Sigma)$ denote $(R_0,R_1,\ldots,R_{(n-1)})$ and likewise for $S(\Sigma)$. Let T denote the set $\{\delta_0,\delta_1,\ldots,\delta_{n-1},\kappa_0,\kappa_1,\ldots,\kappa_{n-1}\}$ of n state transition functions and n failure functions; the δ_i's and κ_i's map system states to system states. A solution to the mutual exclusion problem may specify only the δ_i's ; the κ_i's are given. Note that $\delta_i(p) = p$ may be possible; this corresponds the notion of "busy waiting."

The dead states of a process P_i have $S_i = D$, the critical states have $R_i = C$. We define L to be the set of all infinite sequences $(p^{(0)},p^{(1)},\ldots)$ of system states such that $p^{(0)} = (0,0)^n$ is the system start state and for all $i \geq 0$ there is a function $\alpha_i T$ such that $\alpha_i(p^{(i)}) = p^{(i+1)}$, and such that for all iN and jn if $S(p_j^{(i)}) \neq D$ then there exists an $i' > i$ such that $\alpha_{i'}=\delta_j$ or $\alpha_{i'}=\kappa_j$. Thus L is the set of legal sequences, embodying our assumption that each process must make progress or fail. Let $A = \{q \mid (\exists(p^{(0)},\ldots)L)(\exists iN)(q = p^{(i)})\}$ denote the set of accessible system states.

The constraints that any solution must meet are as follows. The notation in means that i is "of type" n (i.e. $0 \leq i < n$), and kN means that k is a natural number.)

1. A is finite.
2. $(\forall pA)(\forall in)\ S((\kappa_i(p))_i) = D$.
   (κ_i kills the i-th process.)

3. $(\forall pA)(\forall i,jn)((i\neq j) \Rightarrow (\kappa_i(p))_j = p_j)$.

   (κ_i can't affect other processes.)

4. $(\forall pA)(\forall i,jn)((i\neq j) \Rightarrow (\delta_i(p))_j = p_j)$.

   (δ_i can't affect other processes.)

5. $(\forall p,qA)(\forall in)((p_i=q_i \wedge (\delta_i(p))_i \neq (\delta_i(q))_i) \Rightarrow S(\delta_i(p))=S(p))$

   (δ_i can't store while fetching.)

6. $(\forall pA)(\forall in)(\exists j)(\forall qA)[(p_i=q_i \wedge S(p)_j=S(q)_j) \Rightarrow (\delta_i(p))_i=(\delta_i(q))_i]$.

   (δ_i can fetch only p_i and S_j, where j may depend on p_i.)

7. $(\forall pA)(\forall i,jn)((i\neq j) \Rightarrow (R(p)_i = C \Rightarrow R(p)_j \neq C)$.

   (Mutual exclusion.)

8. $(\forall kN)(\forall(\ldots p^{(k)},\ldots)L)(\forall jn)[(S(p_j^{(k)}) \neq D \Rightarrow (\exists iN, i\geq k)((\alpha_i=\kappa_j) \vee (R(p_j^{(i)})=C))]$

   (No lockout)

9. $(\forall jn)(\forall in,\ i\neq j)(\forall k,k',k''N,\ k<k'<k'')(\forall mN)(\forall(p^{(\emptyset)},\ldots)L)[(S(p_j^{(m)})\neq D) \Rightarrow$
   $[R(p_i^{(k)})=C \wedge R(p_i^{(k')})\neq C \wedge R(p_i^{(k'')})=C \Rightarrow$
   $(\exists \ell, \ell<k'')(\alpha_\ell=\kappa_j \vee R(p_j(\ell))=C]]$.

   (Linear waiting.)

We remark that while our formal specification permits a dying process P_i to remember its internal state (that is, κ_i affects S_i but not R_i), any such information can clearly not be used to restart P_i where it left off in the synchronization protocol, since constraints 8 and 9 only require the other processes to pay attention to P_i when it is active.

Though we shall present our solution in the vernacular of the programming milieu, it should be clear how to translate our solution into a system of δ_i's satisfying the above constraints. It should also be possible to translate any system of δ_i's satisfying the constraints into a system of programs satisfying our intuitive understanding of the problem's constraints.

The research presented here was motivated by an unpublished solution by A. Meyer and M. Fischer which used global variables and did not have ultrareliability.

II. The Two-Process Solution

In this section we present a solution for the two-process problem satisfying constraints with a proof of its correctness. This solution will be used later as a subroutine in the n-process solution.

The public variable S_i will be equal to D whenever P_i is either dead or uninterested in entering its critical section. The public variables also acts as a counter modulo 3; they may assume the values 0, 1 and 2 in addition to the value D. The public variables may be easily implemented using only two bits each. We assume from now on that all arithmetic performed on them will be performed modulo 3.

The protocol is given in Figure 1. To execute its critical section, process i executes the block, "begin twoprocess(i,j); critical section; S_i := D end", where j is the name of the other process.

```
procedure twoprocess (integer i,j);
begin
        if Sj≠D then Si := 1+Sj else Si := 0 ;
        if Sj≠D then Si := 1+Sj ;
        wait until (Sj=D ∨ Sj=1+Si ∨ (i=0 ∧ Sj=Si))
end
```

Figure 1. The Two Process Solution

The evaluation of either of the first two statements is assumed to involve at most one fetch of the value of S_j (that is, the value of S_j used in the conditional is the value used in its associated assignment statement.) The wait statement is assumed to involve the repeated fetching of S_j and testing until the condition becomes true (i.e. busy waiting). The conditions for the two processes can obviously be simplified when the values of i and j are instantiated, since the truth of i=0 is fixed once i is known.

To clear up any possible ambiguities, Figure 2 presents a rewritten version of the body of our protocol using only indivisible statements. We use the primitive operations of "fetch" (an assignment of S_j to the temporary variable U, a register inaccessible to P_j), "store" (an assignment to S_i whose right hand side names no variables visible to P_j), =, if and goto. Statements labelled A-E either fetch S_j or change S_i; these are the crucial statements when considering alternative interleavings of statements executed by the two processors. Since the if's and go to's of P_i's protocol commute with all P_j's statements in that their relative order does not affect the outcome, we include the loop implementing the wait condition together with its corresponding fetch in step E.

A: $U := S_j$;

B: $S_i := (\underline{\text{if}}\ U = 0\ \underline{\text{then}}\ 0\ \underline{\text{else}}\ 1+U)$;

C: $U := S_j$;

D: $S_i := (\underline{\text{if}}\ U = 0\ \underline{\text{then}}\ S_i\ \underline{\text{else}}\ 1+U)$;

E: $U := S_j$;

$\underline{\text{if}}\ \neg(U=0 \vee U=1+S_i \vee (i=0 \wedge U=S_i))\ \underline{\text{then}}\ \underline{\text{go to}}\ E$;

Figure 2. Protocol using only indivisible statements

The semantics of a programming language do not normally take into account the possibility of processor failure. Rather than change the semantics we will change the protocol to reflect the possibility that after any instruction the indivisible statement

F: $(S_i := 0;\ \underline{\text{go to}}\ A)$

may non-deterministically be executed. It is more convenient to describe this addition to the protocol using a nondeterministic state transition diagram than to stick to the ALGOL-like notation. For our purposes, the only relevant state information is the contents of S_i and U. Hence the whole critical section collapses to a single state. All code executed before the protocol is entered or after the critical section is exited (via an execution of a statement equivalent to F). similarly collapses to a single state, labelled "entry" in Figure 3, on the assumption that eventually the protocol will be used again. This assumption is inessential to any correctness argument using it, since a process may wait arbitrarily long

before re-entering. We have for convenience collapsed together as state 4 all states after step D but not satisfying the wait condition at step E . This will not affect the correctness proof.

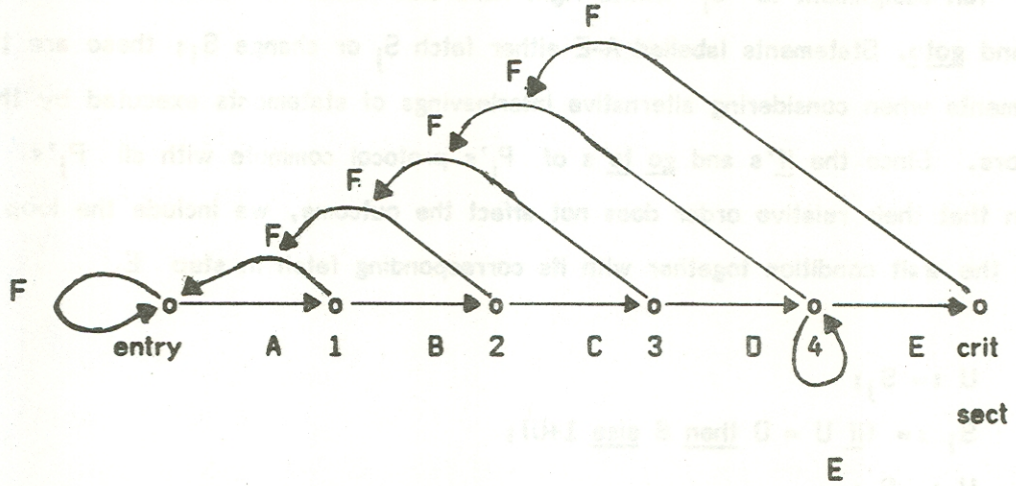

Figure 3. State transition diagram of the program

We observe that the procedure consists of a loop-free block of code (statements A-D) followed by a single loop which waits until the process may safely enter its critical section. The act of setting S_i to $1+S_j$ is equivalent to saying "After you" to process P_j since if both processes arrive at statement E simultaneously, that process whose S value is one less than the other process's S value may enter its critical section. The repetitive nature of A-D ("After you, after you") may appear redundant, but we doubt the existence of a solution with only one fetch from the other process's memory (not counting the fetch in the test).

Lemma 2.1. Deadlock is impossible .

Proof: A deadlock could only arise if both processes were looping at statement E . In this case one of the two S values must be one less than the other (since we are working mod 3), or else the two values are identical. In either case exactly one process will find that it may proceed to enter its critical section. ∎

Lemma 2.2. The two processes are never simultaneously in their critical sections. (Mutual exclusion is effected.)

Proof. We exhaustively consider all computation sequences leading up to the possible violation of mutual exclusion, an initially forbidding prospect that fortunately collapses into four easy cases. Consider the final ABCD path each process took through its transition diagram just before the violation. These two paths can be merged in $\binom{4+4}{4} = 70$ ways. Assuming that P_i is the last process to execute its D transition reduces this to 35 cases. (Naturally we shall not assume that i is zero in the tie-breaking i=0 test.) The upper-case letters A,B,C,D,E,F will denote P_i's transitions, and the corresponding lower-case letters will denote P_j's transitions. Each of these 35 cases will in general include many variants on the basic merge of the two paths. Thus the case ABabcCDd represents such variant sequences as ...AafBabcCDdEe, ...ABabfabcCDEde, ...ABabcCDEde, and so on.

We have assumed that the catastrophic sequence ends in D; it must therefore end in either CD or dD. It could not end in CD since after D, $S_i = S_j+1$, so that P_i can not enter its critical section.

A sequence ending in dD must end in cdD or CdD; the catastrophic sequence could not end in cdD since after cd process P_j is not able to enter its critical section until the D, after which P_j can enter its critical section if and only if P_i can not.

A sequence ending in CdD must end in cCdD or BCdD; the former case can be excluded from consideration since c and C can be interchanged without affecting the outcome.

A sequence ending in BCdD must either be the sequence abcABCdD or it must end in cBCdD. The former sequence is not catastrophic since D and B store the

same value (A and C are not separated by any action by P_j). Therefore any decision P_j makes after d will not later be invalidated by D.

A sequence ending in cBCdD can not end in catastrophe since d does not change S_j (note that $S_i = D$ when c occurs). Therefore after D we have $S_i = S_j+1$, so that P_i can not proceed.

This completes the proof of Lemma 2.2. ∎

Lemma 2.3. The algorithm has "no lockout" and "linear waiting".

Proof: Once P_i has reached its "wait" statement either P_i can proceed or P_j must eventually set S_j to D (since deadlock is impossible). P_j can execute its critical section at most once before setting S_j to D. Thereafter, until S_i executes its critical section and sets S_i to D, S_j can only assume the values of D and $1+S_i$ (S_i is constant during this time). In either case P_i is enabled to proceed to its critical section. ∎

Ultrareliability follows from Lemma 2.3, whose proof does not depend on the integrity of P_j. We have now demonstrated that the solution has all the desired properties: "mutual exclusion," "no lockout" (or deadlock), "linear waiting," "public variables," "finite ranges," and "ultrareliability."

III. The n-Process Solution (First Version)

We now solve the mutual exclusion problem for n processes where $n>2$, still heeding all our constraints. A straightforward generalization of the two-process solution eludes us, and we content ourselves with a quite different solution that requires the foregoing two-process solution as a subroutine.

We first solve the n-process problem in the absence of the "linear waiting" constraint. This uses the two-process solution as a subroutine, and our n process solution here will itself be used as a subroutine to solve the n-process problem with all constraints in force. We say that process P_i is racing with process P_j when P_i is executing the two-process protocol, using S_j as the other public variable. When process P_i completes execution of that code we say it has won the race and beaten process P_j .

In the absence of the "linear waiting" constraint our solution is for each process to race with processes 0 through $n-1$ in turn, winning against itself. When it has won every race, it then executes its critical section. One way of implementing this scheme is to let each process have $n-1$ copies of the data structures used in the two-process solution, one for each of the other processes. When process P_i races with process P_j , process P_i fetches from copy i of process P_j's data, and stores into copy j of its own data. Then when it has won against process P_j , it leaves copy j of its data as it is, preventing process P_j from later racing with and beating process P_i, and proceeds to race with process P_{j+1}. When it has beaten all the other processes, it then executes its critical section; when done it sets all $n-1$ copies of its public variable to 0 , permitting processes waiting on process P_i to proceed.

A improvement to this scheme is for each process to have only one copy of its S register, but to have in addition a public R register which names the process currently being raced with. Then the test for whether process P_i can proceed at the end of its race with P_j becomes

$$R_j=0 \vee R_j<i \vee (R_j=i \wedge (S_j=0 \vee S_j=1+S_i \vee (S_i=S_j \wedge i\leq j))).$$

We arbitrarily favor the smaller-numbered process to break any ties. By making this test $i\leq j$ the tie that must happen when a process races with itself allows that process to proceed. A dead process is assumed to have $R=0$, while a critical process has $R=n-1$.

While we have postulated the use of two public variables per process here (and later on we will use three), our intention is to use a single "super" public variable, with

some standard pairing function providing the encoding. Thus the simultaneous assignment to two or more public of a process would actually be effected by a single assignment to the "super" variable. As before, we assume that any statement which involves the value of another process's (super) public variable will fetch that value once and use it consistently thoughout the statement.

Our first n-process solution is given in Figure 4.

```
procedure  nprocess (integer i);
begin procedure two process (interger j);
      begin if R_j ≠ 0 then (S,R)_i := (1+S_j, j) else (S,R)_i := (0, j);
          if R_j ≠ 0 then S_i := 1+S_j;
          wait until R_j = 0 ∧ R_j < i ∧
                  R_j = i ∨ (S_j = 0 ∧ S_j = 1+S_i ∧ (S_i = S_j ∧ i≤j)))
      end
      for  j := 0 to n-1 do twoprocess (j)
end
```

Figure 4. The n-process solution (first version)

To execute its critical section, process i executes the block of code, "begin nprocess (i); critical section; $R_i := 0$ end."

Lemma 3.1. This algorithm satisfies the "mutual exclusion," "no lockout," and "ultrareliability" constraints (i.e. all but "linear waiting").

Proof. Mutual Exclusion. Suppose P_i and P_j are simultaneously in their critical sections. Then process P_i beat process P_j and vice versa while the respective processes were leading up to their critical sections. Suppose without loss of generality that P_i beat P_j before P_j beat P_i. If P_j beat P_i before P_i increased R_i to $j+1$, this would contradict the correctness of the two-process solution. If P_j beat P_i after P_i increased R_i to $j+1$, this would contradict j's being stopped by the failure of both $R_i<j$ and $R_i=j$.

No Deadlock or Lockout. Assume there exists an active process in the system. We shall constructively prove the existence of a process that is able to make progress in the sense that it can complete the race it is presently engaged in. Since for fixed n the number of races a process engages in is fixed, it follows that some process will eventually reach its critical section.

The only way for a process not to be able to make progress is for it to be unable to pass the test when racing with some other process. Starting from the active process that we postulated to be in the system, we enumerate processes such that the next process enumerated is the one the last one enumerated is racing with and is beaten by. Eventually this enumeration must either terminate or cycle. If it terminates, the last process enumerated is free to proceed. We claim that it cannot cycle. If it does cycle the cycle is of length 1, 2 or greater. Length 1 is ruled out because the algorithm ensures that a process always beats itself. Length 2 is ruled out since the two-process solution has "no lockout." So assume the cycle is of length 3 or more. Let P_i, P_j, P_k be three consecutive processes in the enumeration such that P_k is the least-numbered process of the three. (Choosing P_k to be the least-numbered in the cycle will ensure this.) Then P_j has thus far raced only with processes P_1 through P_k. Since $i>k$, P_j cannot have raced with P_i yet, contradicting P_i's being beaten by P_j.

Ultrareliability follows from the above proof of "no lockout" since any failing process automatically forfeits any race it is engaged in.

IV. The n-Process Solution (Second Version)

We turn our attention now to effecting "linear waiting." The problem with the previous solution is that while a lower-numbered process is working its way up through the ranks as it competes with processes $0,1,2,\ldots$, a higher-numbered process may repeatedly enter the lists, win because no one has worked his way high enough to challenge him yet, and proceed to his critical section. To effect "linear waiting" each process P_i which has beaten every other process using the previous protocol then sets R_i to the value Q and performs additional coordination as decribed below before entering its

critical section.

Our solution is modelled on that of Eisenberg and McGuire [5], who use a global variable to point to processes $0,1,2,\ldots,n-1,0,\ldots$ in turn, permitting the indicated process to execute its critical section. The appropriate imagery is of a (one-handed) clock , the term we shall use for such a pointer.

In the absence of global variables, each of the waiting processes P_i will maintain an up-to-date public copy c_i of a virtual clock. The public variables of process P_i are now S_i, R_i, and c_i. The absence of a single time-keeping authority complicates the Eisenberg-McGuire solution considerably. It is clearly impossible to have all the waiting processes maintain the same clock value. However, it is possible to have at most two (consecutive) values in the system at any moment. We adopt the notation $[a,b]$ to denote the set of integers $\{a,a+1,a+2,\ldots,b-1,b\}$, where the arithmetic is performed modulo n. Thus $[a,b-1]\cup[b,a-1] = \{0,1,\ldots,n-1\}$ while when $a\neq b$, $[a,b-1]\cap[b,a-1] = \{\}$. A system clock is defined to be a c_i for any i such that $R_i=Q$. A process P_i is in the system if $R_i=Q$. We formalize our inductive hypothosis thus.

Window Hypothesis (WH). At all times no two system clocks differ by more than 1 mod n. (Hence for each moment in time there must exist a k such that for every process P_i with $R_i=Q$, $c_i \in [k,k+1]$. This k is called the location of the window , or the value of the clock.)

The general principles for maintaining this state of affairs are:

(i) A process P_i may not tick (increment its clock by 1 modulo n) unless $R_i = Q$ and for every active process P_j (i.e. $R_j \neq 0$), $c_j \in [c_i,c_i+1]$. The condition for advancing one's clock is $(\forall j)[R_j=0 \vee c_j=c_i \vee c_j=c_i+1]$. Thus the main loop for a process P_i having $R_i = Q$ consists of waiting for this condition to come true, then doing $c_i := c_i+1$.

(ii) To join the system, a process sets its clock to an existing system clock, or to an arbitrary value if no other process is in the system, and then sets R_i to Q.

When $R_i=Q$ and $c_i=i$, process P_i proceeds to execute its critical section. The test for this condition is performed after the test for whether to advance, but before the actual advance. It is not hard to see how this guarantees mutual exclusion assuming WH, at least for the case when no processes are in the act of joining the system.

Now let us consider the implementation of (ii), joining the system. While the basic idea (set your clock to some system clock) seems plausible, the process is akin to boarding a rapidly spinning carousel. One may pick up a clock value, but find it out-of-date by the time one has stored it in one's own clock. The solution to this is to set $R_i \neq 0$ before selecting a clock value. In this way the virtual system clock cannot spin much further while your clock does not change. This raises the possibility of destroying ultrareliability . The solution is as for the two-process solution - set your clock initially to some system clock before you set $R \neq 0$ and independently select another clock value after setting $R \neq 0$. Thus although no guarantee is made that the first attempt to set your clock will bring you up to date, at least it avoids your being responsible for the system's making no further progress; each time you fail and re-enter, the system is no longer prevented by you from incrementing its clock one more time.

A second potential problem is that of the "phantom clock." A process P_i may commence joining the system by fetching a system clock, but not storing it immediately. In the meantime everyone in the system leaves it, one way or another. Then another process P_j attempts to join the system, finds that there are no system clocks in existance (i.e. all processes have an R value $\neq Q$), so P_j sets c_j to some arbitrary value and then sets R_j to Q. (The example can be made to work no matter what he chooses because of the inaccessibility of the "phantom clock" being held by our temporarily suspended process.) The first process now joins the system with its clock set to the "phantom clock," the value he saw originally, which may bear no relation to the new system clock.

The solution we adopt to this problem is to make the operation of joining the system an atomic (i.e. mutually exclusive) operation, using the previous n-process solution as the synchronization mechanism. This precludes any other process's joining the system

while a phantom clock exists.

A third potential problem arises as follows. Suppose that at some time $c_i=2$, $R_i=n-1$, $c_j=0$ and $R_j=0$ (possible because of potential delay between P_i's first fetch and store of a clock value, or by virtue of the entire system dying and being replaced by a new one with an unrelated system clock, as in the "phantom clock" example). Further, suppose that P_j is just about to increment its clock. Now suppose P_i fetches (but does not yet store) $c_j=0$, then c_j ticks twice to 2 as it is entitled to do, and then c_i is set to 0. If there are no other system clocks, P_i will proceed to join the system and (for $n>3$) destroy the window hypothesis WH.

Our solution to the third problem is simply to ensure that c_i cannot be so out-of-date as to be overtaken in this manner. We accomplish this with one more assignment of a system clock to c_i, and immediately before the final assignment after R_i ceases to be D (and hence after the first assignment, so altogether three similar assignments are made). To avoid the problem of the vanishing and reappearing system, we make the second and third assignments one atomic block of code, using the synchronizing mechanism proposed for the last assignment alone. After the middle assignment, no system clock will be able to advance more than four ahead of c_i (proved below), so for $n\geq 7$, the above problem cannot occur.

A fourth potential problem is the following. Suppose $c_j=1$ and $c_{j+1}=0$, and suppose c_i becomes 1 at the intermediate assignment. Now let P_j leave the system (say by dying) so that for the final assignment, P_i fetches P_{j+1}'s clock, which is still 0. But before P_i can store this value, P_{j+1} ticks twice to 2 (permitted since $c_i=1$). Now c_i becomes 0 and again we lose WH. This problem can actually be detected by P_i, since the final value it fetched was one less than the intermediate one. In this case, P_i can ignore the final value and keep the intermediate one.

A fifth potential problem is variant of the "phantom clock" problem: suppose that all system clocks are 0 or 1, but that all processes joining the system set their clocks to 0, with other system processes dying as needed to make room for new processes. In

this way the system can fail to make progress since there is always a system clock at 0. Our solution is to look at all system clocks in turn, ignoring those system clocks one less than the last distinct system clock we saw. In this way, if the system is not making progress, we will get the maximum system clock; if the system is making progress, the value we choose, while not guaranteed to be the maximum, at least will be no worse a choice than an arbitrary system clock.

The whole protocol is then:

1. c_i := "maximum" value of the visible system clocks (to prevent system deadlock due to the repeated failure and restarting of P_i).

2. Execute the equivalent of Dijkstra's "P" by using the procedure nprocess of the previous section to race against every P_j in turn, for $j = 0,...n-1$, which also sets R_i to a non-0 value.

3. c_i := "maximum" value of the visible system clocks (to update c_i).

4. c_i := "maximum" value of the visible system clocks, if this doesn't decrement c_i by one, and do a "V" simultaneously (it suffices to set R_i to 0).

5. Wait till all clocks of active ($R \neq 0$) processes are in $[c_i, c_i+1]$.

6. If $c_i \neq i$, increment c_i by one (modulo n), and return to step 5.

Figure 5. The n-process solution (rough sketch of final version).

With the protocol before us we can observe one final problem. When a process leaves its critical section, it is possible for it to re-enter the system and find everything unchanged, with the system clock still pointing to it. In this way it may repeat its critical section arbitrarily often before other processes already in the system get a turn. To avoid this injustice we forbid a process from entering its critical section the first time it reaches

step 7 of the above algorithm. We do this with a flag ok which is set during step 4, and which is cleared during the first execution of step 6.

We now present the complete protocol for process P_i. We assume that S_i is of type integer mod 3 and c_i is of type integer mod n , implying that all arithmetic involving them is done modulo the appropriate quantity. Initially c_i is set to some random value, but its type forces the value to be in the range 0 to n-1 . R_i will successively take on values in some prefix of the sequence of D,0,1,2,...,n-1,Q , followed by D . R takes over some of the role of S, and the test S_j=D of the two process solution is replaced by the test R_j=D $\vee$ R_j<i $\vee$ R_j=Q .

```
Procedure  nprocess2(integer i)
begin
integer procedure sysclock;
   begin integer m,j;  m := D;
                  for j := 0 step 1 until n-1 do
                        if Rj=Q ∧ cj+1≠m then m := cj;
                  sysclock := if m≠D then m else 0
   end sysclock;

1: ci := sysclock;

2: for j := 0 until n-1 do nprocess (j);

3: ci := sysclock;

4: m  := sysclock;
   if m≠ci-1 then (R,c)i := (Q,m) else Ri := Q;
     ok := false;

5: for j := 0 until n-1 do wait(Rj=D ∨ cj=ci ∨ cj=ci+1);

6: if ci≠i ∨ ¬ok then begin ci := ci+1; ok := true; go to 5 end
```

end

Figure 6. The n process solution (final version).

When using WH as an induction hypothesis we shall strengthen it to include the following condition. For every k such that $R_k = Q$, when P_i is either in step 5 with $j>k$ or is between step 5 and the increment of step 6, the system clock c_k must be om $[c_i, c_i+1]$. This excludes the possibility that P_k might join the system with a valid clock (i.e. $c_k = c_i-1$), only to have it invalidated a moment later as P_i proceeds to increment its clock.

Lemma 4.1. After P_i has executed step 2, $R_i \neq D$.

Proof. Indeed, R_i is set to $\emptyset$ at the outset of step 2, and then increments up to n-1. ∎

Lemma 4.2. Assume the strengthened WH. If $R_j=R_k=Q$ but $c_k=c_i-1$, P_i cannot pass step 5 before c_k is incremented.

Proof. Evident from the code. (By the strenghtened WH, $c_k=c_i-1$ implies that P_i has not yet tested c_k in step 5. The strengthened form of WH excludes the case where P_i has already passed step 5 when P_k sets R_k to Q with $c_k+1=c_i$. (It is tempting here to argue that WH unstrengthened will suffice, since from the state described in the Lemma, P_i could legally proceed to invalidate WH, contradicting our assumption that WH held. The catch is that we want to use Lemma 4.2 in the inductive step of the proof of WH, and this sort of "looking into the future" would invalidate this use.)

∎

Lemma 4.3. Assume WH and that $n\geq 7$. After P_i reads some system clock c_j in the subroutine sysclock in step 3, no system clock may reach c_j+5 before c_i changes value.

(If 7 seems a little high, note that when fetching c_j, some process may have its clock at c_j-1, which is already c_j+5 when $n=6$.)

Proof. At the moment c_j is read no process P_k with $c_k=c_j+1$ can have passed the test at step 5 permitting it to increase c_k to c_j+2 (Lemma 4.2). (By WH, c_k is in $[c_j-1, c_j+1]$ at this time.) Hence to reach c_j+5 P_k must pass through step 5 four times, with four successively higher (modulo n) clock values. The last three times through step 6 must involve an examination of c_i. Because c_i remains constant (hypothesis) and $R_i \neq 0$ (Lemma 4.1), the test against c_i at step 5 could not have succeeded all three times. ∎

Lemma 4.4. After P_i completes the store into c_i of step 3 but before any other process executes an instruction, every system clock lies in $[c_i-1, c_i+4]$.

Proof. By WH, when c_j was fetched all system clocks lay in $[c_j-1, c_j+1]$. By Lemma 4.3 and the fact that system clocks can only increase, they can only move on to as far as c_j+4. Immediately after the assignment of this c_j to c_i, we can say the same of c_i. ∎

Lemma 4.5. After P_i completes step 3 but before c_i changes again, every system clock lies in $[c_i-1, c_i+4]$, and no system clock in $[c_i, c_i+4]$ will be able to increment more than once during this period.

Proof: By lemma 4.4 every system clock begins the period in $[c_i-1, c_i+4]$, and no process p_j with $c_j = c_i+4$ could have passed yet the test of step 5 (against c_i) permitting it to advance to c_i+5, by the proof of lemma 4.3. Therefore any process P_j which begins the period with $c_j = c_i+4$ or which sets c_j to c_i+4 during this period will be unable to pass the test at step 5 permitting it to advance c_j to c_i+5 before c_i is changed again.

Similarly any process P_j such that $c_j \in [c_i, c_i+4]$ which increments c_j once during this period will be unable to increment it again during this period since c_i will not be in

$[c_j, c_j+1]$, and therefore P_j will fail the test at step 5. ∎

Lemma 4.6. When P_i "joins the system" in step 4, if WH held just after P_i executed step 3, then WH holds just after P_i has joined the system.

Proof. Lemma 4.2 implies that until P_i joins the system (or fails), WH will hold in that all system clocks will differ by at most one. Lemma 4.5 implies that no system clock c_j will increment more than once during this period, unless c_j was initially c_i-1, in which case it may tick twice. Let c_k be the largest system clock just before P_i joins the system, and let c_k' be the value that c_k had when P_i was beginning step 4. If $c_k' \in [c_i, c_i+4]$ then c_k has ticked at most once in this period, so c_i will be set either to c_k or $c_k' = c_k-1$, and so will be in the window. If $c_k' = c_i-1$ (the only other possibility, by lemma 4.4), then $c_k \in [c_i-1, c_i+1]$, and c_i will be set to a value in $[c_i, c_i+1]$, thereby preserving WH. (If c_i is not changed then WH is obviously preserved, and if P_i sees a $c_k = c_i+1$ then no system clock is at c_i-1 by lemma 4.2). ∎

Lemma 4.7. Provided the system starts with all processes dead, the Window Hypothesis will hold at all times.

Proof. We proceed by induction on the number of instructions executed by all processes since some time when all processes were dead. We only care about instructions that assign to c_i while $R_i=Q$, and instructions that set R_i to Q. All others cannot do any harm immediately. For each i there is only one instruction of each of these two kinds, $c_i := c_i+1$ at step 6, and $R_i := Q$ at step 4. The first of these preserves WH since i has passed the test at step 5 (whose intent is clearly to preserve WH), and the strengthening of WH assures us that no process has since then joined the system with a value other than c_i or c_i+1. The second preserves WH on account of lemma 4.6.

∎

Theorem 3.8. This algorithm has "mutual exclusion," "no lockout," "linear waiting," and "ultrareliability."

Proof.

Mutual Exclusion. Because of WH, only one process can simultaneously satisfy $c_i=i$ and have a minimal clock (one such that for no j is $c_j=c_i-1$). Because system clocks do not decrease, the minimal clock property guaranteed by step 6 must still hold at step 7, by WH as strengthened.

No Lockout or Deadlock. The only source of problems for the reliable process P_i are the wait at step 2, dealt with already by Lemma 3.1, the wait at step 5, and the goto at step 6. By WH, eventually all processes with $R\neq 0$ will have a clock value equal to c_i or c_i+1, by WH and the fact that the largest system clock is chosen. Moreover, any process with $c=c_i-1$ will pass the test at step 5, so eventually no such process will remain. Then P_i will pass the test at step 5. The goto at step 6 allows c_i to make progress, and this combined with setting ok means that eventually the test at step 6 will not be satisfied, and P_i will enter its critical section.

Linear Waiting. If we weaken this condition from its original statement to "each process need wait while at most kn other (not necessarily all distinct) processes execute their critical sections," then this is evident from the argument for C2-C3. In order to improve the waiting to "while $R\neq 0$, each other process may execute its critical section at most once," we need to change the algorithm to avoid processes i-1 through i-5 being served twice. A simple solution to this is to introduce about 4n "dummy" processes, renumbering the original processes so that they are 5 apart. This need not entail the actual construction of memories for the dummy processes, since the real processes will always know that the dummies are dead. The effect is to "slow down" the system clock a bit, at the cost of additional overhead in the execution of the algorithm. An additional benefit of such an approach is to solve the problem for the case when there are fewer than 6 processes. Unless the cost of the critical sections far outweighs that of our solution to their mutual exclusion, this "padding" of additional processes is probably of no practical value.

Ultrareliability. Follows from the proof of "no lockout," since a failing process must forfeit any priority it has earned.

V. Summary

We have demonstrated that the primitive operations of reading and writing public variables are sufficiently powerful to implement a solution to the mutual exclusion problem, even if the participating processes are permitted to fail arbitrarily often. The demonstrated solution does not make exorbitant demands on memory capacity (the public variables only assume a finite number of values), and it ensures that the processes are treated equitably (the solution has the "linear waiting" property).

Whether the solutions presented here will turn out to be useful in practice remains to be seen. Our assumption that the number of processes remains fixed is often violated in practice but appears to be essential to our model. We should also like to direct the reader's attention to reference [8], where Peterson and Fischer present alternative (and rather elegant) solutions to our problem in an effort to reduce the amount of computation, and the number of states of the public variables, required by our solutions.

Acknowledgments

We should like to thank Leslie Lamport, Michael Fischer, and Albert Meyer for their helpful comments on earlier versions of this algorithm and manuscript, and Michael Fischer and Gary Peterson for detecting and correcting some serious flaws in a later version.

VI. References

[1] de Bruijn, N. G., "Additional Comments on a Problem in Concurrent Control", CACM 10 (March 1967), 137-138.

[2] Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control",

CACM 8 (September 1965), 569.

[3] Dijkstra, E. W., "Self-Stabilizing Systems in Spite of Distributed Control", CACM 17 (November 1974), 643-644.

[4] Dijkstra, E. W., "Cooperating Sequential Processes", in Programming Languages. F Genuys, Ed., Academic Press, New York (1968).

[5] Eisenberg, M. A., and M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem", (CACM 15 November 1972), 999.

[6] Knuth, D. E., "Additional Comments on a Problem in Concurrent Control", CACM 9 (May 1966), 321-322.

[7] Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem", CACM 17 (August 1974), 453-455.

[8] Peterson G.L. and Fischer M.J., "Economical Solutions For The Critical Section Problem In A Distributed System," University of Washington, Department of Computer Science Technical Report, Technical Report No. 77-02-03. (To be presented at the Ninth ACM Symposium on Theory of Computing, Boulder, Colorado, May 2-4, 1977.)