The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report

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

Abstract

Consider n processes operating asynchronously in parallel, each of which maintains a single "special" variable which can be read (but not written) by the other processes.

All coordination between processes is to be accomplished by means of the execution of the primitive operations of a process (1) reading another process's special variable, and (2) setting its own special variable to some value. A process may "die" at any time, when its special variable is (automatically) set to a special "dead" value. A dead process may revive. Reading a special variable which is being simultaneously written 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 coordination 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 special variables are bounded in value.

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

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.

Introduction

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

We assume the existence of n processes operating asynchronously in parallel. The number n is known to every process. Furthermore, new processes may not be added to the system; the number n may not change over time. Therefore, processes may not leave the system either, although they may "die" in a sense to be explained later. The processes act independently in the sense that any action possible for a process may be performed at any time; none of the basic operations that a process can perform require for their successful completion any sort of delay or any dependence on the state or actions of other processes. (Thus our model differs from one in which the P and V coordinating primitives are used.)

Each process maintains a single "special" variable which it uses to convey information to the other processes. A special variable may be changed only by the process that owns it, although it may be inspected at any time by any of the other processes. If a special variable is being simultaneously changed by its owner and read by some other process, the other process will see either the old or the new value. More precisely, the effect of any set of read and writes of a special variable will always be the same as if they were executed sequentially in some order. Later on, we shall assume that the entire set of reads and writes of all the special variables during a particular computation can be arranged in a sequence without changing the net effect. That this may be done follows directly from the above assumptions.

All communication between processes will be accomplished by means of the special variables. The only information that process i can obtain about process j is the value of process j's special variable. A process's local variables, program counter, internal state, etc., are invisible to the other processes. Note that for one process to request the value of the special variable of another process, it must have a name for that other process. We will assume that the names are the integers 0 to $n-1$.

The most outstanding 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. In the real world a device may fail, or die, in any one of many possible fashions. For example, it may simply cease to function but 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 very disciplined fashion by simply posting a flag that it has failed and then stopping.

The first mode of failure (dying without any symptoms) cannot be incorporated into our model. Since we do not wish to make any assumptions about the absolute or relative speeds of the various processes, process i cannot, merely by noticing that process j's special variable has not changed in a long time, conclude that process j has died. Therefore process i could not distinguish between the death of process j and a long delay due to (say) process j executing a very long internal computation. We do require that each process continues to make progress, however slowly or quickly. Thus no process is permitted to stop during its computation with no evidence visible to the other processes that it has stopped, since this could hang up the system (the other processes might think that the stopped process was in its critical section, for example).

Similarly, the second mode of failure could cause problems. In particular, if processes were allowed to misbehave in an arbitrary fashion, a solution to the mutual exclusion problem would become impossible, since a process could enter its critical state and then refuse to alter its special variable when it leaves it, causing the other processes to be locked out.

Therefore, we require that a process may fail only in the following disciplined manner (the third mode of failure): it must

This research was supported by the National Science Foundation under contracts MCS76-14294, MCS76-18461 and DCR74-12997.

leave its critical section if it is in it, set its special variable to the value "dead" (or some value defined to be equivalent), and then stop. This is the only way a process may stop or fail.

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 programs, one per process, which may be executed by their owners at any time. Each program contains 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. This is our first requirement of a solution:

(C1) *Mutual exclusion*: No two processes may be in their critical sections at the same time.

Dijkstra's solution [2] satisfies (C1) and also the obviously desirable criterion:

(C2) *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 (this was pointed out by Knuth [6]):

(C3) *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. Knuth presented a new solution which satisfied all of (C1)-(C3). In fact, in an n-process system, a process would be able to enter its critical section before the other n-1 processes were able to execute their critical sections (collectively) 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 (C1)-(C3) and:

(C4) *Linear waiting*: A process that is waiting to enter its critical section will at worst have to wait for every other process to take one turn (in its critical section).

Thus no process can execute its critical section twice while some other process is kept waiting.

All of the above solutions have the property that they employ a global variable (one that is 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 system which satisfies (C1)-(C4) using variables local to each process (the special variables of our model); each process may set only its own variable, but may read the special variables of any other process. He assumes that if a process (or the memory unit containing its variable) should fail, its 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 constraint that our solution will satisfy:

(C5) *No global variables*; Coordination is achieved by use of special variables only.

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

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

(C6) *Finite ranges*: The special variables can only assume a finite number of values.

(C7) *Impotence of repeated failing*: A process cannot deadlock the system by repeatedly failing and restarting.

The solution to be presented here will satisfy (C6) and (C7), although in order to satisfy (C7) it is necessary, as we remarked earlier, to assume in (C5) that when a process fails its special variable does not assume arbitrary values for a period of time, but rather changes directly from its previous value to zero. Otherwise a process that repeatedly failed and restarted could have a random value in its special variable whenever another process was reading it.

A solution satisfying (C5) 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 special variable need delay just enough to let the new value propagate to each other process. Or 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 special variable, who in turn replies with a message giving the value of his variable (or a message indicating that he has failed).

For the sake of precision we give formal constraints on the model of parallelism we have in mind. The various assignments, functions and tests we use in our programs may all be abstracted as functions from system states to system states. Further, though our solutions impose considerable structure on the state of each process, in the form of various registers, together with the program counter, we shall be content to stipulate minimum structure of two "registers," separating the visible and invisible components of a process state.

A *process state* (s_a, s_b) is a pair of natural numbers, (respectively the invisible and visible components of its state). A *system state* is an n-tuple $(v_0, v_1, \ldots, v_{n-1})$ of process states; we let v_a denote $(v_{0a}, v_{1a}, \ldots, v_{(n-1)a})$ and likewise for v_b. D is a 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. The solution may specify only the δ_i's; the κ_i's are given.

The *rest states* of a process are those with first component 0; the *critical states* have first component 1. (This permits processes to remember things, albeit publicly, when critical or at rest; we are being generous here, since most solutions in the literature, including ours, will work even if only one rest state and one critical state is permitted each process, i.e. no memory while in those states.)

Let D^*, the reflexive transitive closure of D, be the smallest set of functions satisfying $I \cup D \cup D^* \cup D^* \circ D^* = D^*$. That is, $D^* = \{I, \delta_0, \ldots, \delta_0\delta_0, \delta_0\delta_1, \ldots, \delta_2\kappa_3\delta_7\delta_0\kappa_2, \ldots\}$, all possible compositions of elements of D . We take $\delta_i\delta_j$ to be the function $(\delta_i\delta_j)(x) = \delta_j(\delta_i(x))$. Let $V = D^*((0,0)^n)$, the set of system states accessible from the <u>system start state</u> $(0,0)^n$.

The constraints that any solution must meet are as follows. (Assume every constraint is prefixed by "$\forall i,j{:}n\ \forall v,w{:}V\ i \neq j \supset$" , where $i{:}n$ means that i is "of type" n (i.e. $0 \leq i < n$), $v{:}V$ means that v is an accessible system state, $k{:}N$ means that k is a natural number, etc.)

1. V is finite.
2. $\kappa_i(v)_i = (0,0)$. κ_i kills the i-th process.
3. $\kappa_i(v)_j = v_j$. κ_i can't affect other processes.
4. $\delta_i(v)_j = v_j$. δ_i can't affect other processes.
5. $(v_i = w_i \wedge \delta_i(v)_i \neq \delta_i(w)_i) \supset \delta_i(v)_b \doteq v_b$ δ_i can't store while fetching.
6. $\exists k[(v_i = w_i \wedge v_{kb} = w_{kb}) \supset \delta_i(v)_i = \delta_i(w)_i]$. δ_i can fetch only v_i and v_{kb} .
7. $v_{i_a} \neq 1 \vee v_{j_a} \neq 1$. Mutual exclusion.
8. $\exists k{:}N\ \forall_{i,j}{:}n\ \forall v{:}V\ \forall D' \subseteq D - \{\delta_i\}\ \forall P{:}(D - \{\kappa_i\} - D')^{\&k}\ \exists f{:}P$ $[(\forall_j[\delta_j \in D' \supset v_{j_a} = 0]) \supset f(v)_{i_a} = 1]$. No lockout (& hence no deadlock).

This last condition require some explanation. Define a <u>path</u> of A (A any set of functions) to be a set of elements of A^* of the form $\{I, a_{i_0}, a_{i_0}a_{i_1}, \ldots, a_{i_0}a_{i_1}\ldots a_{i_m}\}$. We take $A^{\&k}$, where $A \subseteq D$, to be the set of all paths of minimal cardinality formed from elements in A such that some function in each path was formed using each δ_i in A at least k times. Thus this condition says that if all processes not at rest have run for k steps, we will have found that each of the non-failing ones among them will have entered its critical section at some time during our wait.

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. Barring oversights on our part, 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 satisfied all conditions but (C5) and (C7).

<u>The Two-Process Solution</u>

In this section we present a solution for the two-process problem satisfying (C1)-(C7). We also present a proof of its correctness, which turns out to be surprisingly complicated for such a short program. This solution will be used later as a subroutine in the n-process solution.

The special variable of process number i will be considered as an ordered pair of variables $(R,S)_i$. The R component may take on the values 0, 1, or 2. When $R = 0$ process i is either dead or uninterested in entering its critical section. The S component acts as counters modulo 3; we assume from now on that all arithmetic performed on it will be performed modulo 3. By the nature of our assumptions, both variables may be changed simultaneously by making a single assignment to the special variable. Even though each of R and S may assume one of three values, the special variable will only assume seven distinct values, since the combinations $(0,1)$ and $(0,2)$ will not occur.

The procedure for process i is given in Figure 1. The other process is named process j .

```
(R,S)_i := (1, 1+S_j);
(R,S)_i := (2, 1+S_j);
wait(R_j=0 v S_j=1+S_i v (i≤j ∧ (R,S)_j=(R,S)_i));
critical section;
(R,S)_i := (0,0);
```

<u>Figure 1. The Two Process Solution</u>

The "wait" statement on the third line merely waits until the specified condition is true. This might involve either repeatedly fetching $(R,S)_j$ and evaluating the test (busy waiting), or merely some sort of hardware circuitry for detecting the condition, if the variables on which the condition depends are continuously available. We shall interpret the statement "wait(P)" as "repeat until P" (i.e. busy waiting) in what follows. To ensure the correct behavior of the test, $(R,S)_j$ should be fetched just once and then the whole condition evaluated. In general, we will assume that the condition for a "wait" command is always evaluated by first fetching the special variable required for the evaluation of the condition, and then evaluating the condition based on this value. (A wait condition will always require the value of just one other special variable.)

The conditions for the two processes can obviously be simplified when the values of i and j are instantiated, since the truth of $i \leq j$ is fixed once i and j are known.

To clear up any possible ambiguities, we rewrite our program using only indivisible statements. We use fetch (represented as assignment to the temporary pair (U,V), a register inaccessible to the other process), store (an assignment whose right hand side names no variables visible to the other process), =, <u>if</u> and <u>goto</u>. We have labelled with A-E and H the statements that fetch $(R,S)_j$ or change $(R,S)_i$. These are the statements that matter when considering alternative interleavings of statements executed by the two processors. The <u>if</u>'s and <u>go to</u>'s commute with all other statements in that their order does not affect the state of any variables, whether visible or invisible to the other process.

```
A:      (U,V)   := (R,S)_j;
B:      (R,S)_i := (1,1+V);
C:      (U,V)   := (R,S)_j;
D:      (R,S)_i :=(2,1+V);
E:      (U,V)   := (R,S)_j;
F:      if ¬(U=0 v V=1+S_i v (i≤j ∧ (U,V)=(R,S)_i)
          then go to E;
G:      critical section;
H:      (R,S)_i := (0,0)
```

<u>Figure 2. Program using only indivisible statements</u>

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 program to reflect the possibility that after each instruction the indivisible statement

<u>go to</u> H

may non-deterministically be executed. It is more convenient to describe this addition to the program using a nondeterministic state transition diagram than to stick to the ALGOL-like notation. For discussion purposes, the only state information needed is the contents of $(R,S)_i$ and (U,V). Hence the whole critical section collapses to a single state. All code executed external to the program of Figure 2 similarly collapses to a single state, labelled "entry" in Figure 3, on the assumption that eventually the program will be used again. This assumption is inessential to any correctness argument using it, since a process may wait arbitrarily long before re-entering.

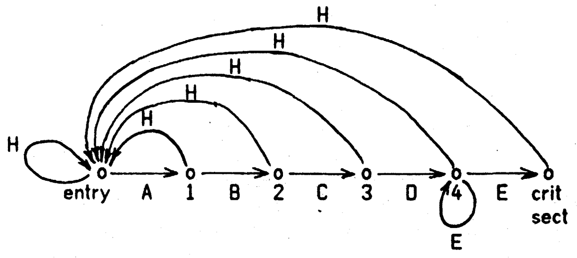

Figure 3. State transition diagram of the program

(We have for convenience collapsed together as state 4 all states satisfying $R=2$ but not satisfying the wait condition. This will not affect the correctness proof.)

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 variable R_i indicates how many times process i has set S_i to be $1+S_j$. The act of setting S_i to $1+S_j$ is equivalent to saying "After you" to process 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 (i.e. (C2) is satisfied).

Proof: Statement E contains the only waiting command, so a deadlock could only arise if both processes were looping at statement E . In this case both R values are 2 , and of the two S values one must be one less than the other (since we are working mod 3), or else the two special variables are identical. In either case exactly one process will find that it may proceed to enter its critical section. ▮

We next show that (C1) is not violated:

Lemma 2.2. The two processes are never simultaneously in their critical sections.

Proof. First observe that if (C1) can be violated, then it can be violated without using any unsuccessful E transitions. Hence we shall assume in this proof that all E transitions are successful.

We exhaustively consider all computation sequences leading up to such a catastrophe, an initially forbidding prospect that fortunately collapses into six easily disposed-of cases. To analyse how (C1) might have failed, we consider the final ABCD trajectory each process took through its transition diagram just before the catastrophe; these two trajectories can be merged in $\binom{4+4}{4} = 70$ ways; labelling as process i the first process to execute its A transition reduces this to 35 cases. (Naturally we shall not assume that i is the winner or loser in the tie-breaking "$i \le j$" test.) Each of these 35 cases will in general include many variants on the basic merge of the two ABCD's. Thus the case $A_iB_iA_jB_jC_jC_iD_iD_j$ may (if (C1) is violated) include such sequences as $\ldots A_iA_jH_jB_iA_jB_jC_jC_iD_iD_jE_iE_j$, $\ldots A_iB_iA_jB_jH_jA_jB_jC_jC_iD_iE_iD_jE_j$, $\ldots A_iB_iA_jB_jC_jC_iD_iE_iD_jE_j$, and so on.

We can ignore any sequence containing as a consecutive pair any of A_jC_i, C_jC_i, B_jB_i, B_jD_i, on the ground that the sequence derived from that sequence by reversing the order of the pair will have the same outcome. Note that D_jD_i and D_jB_i are not on the list because E_j may occur between them. The interchangeability of pairs is not as symmetric as it may appear; though the transformation from A_jC_i to C_iA_j is harmless, if we instead transform C_iA_j to A_jC_i we may overlook the possible sequence $\ldots C_iA_jB_jH_jA_j\ldots$.

The above reduces the 35 cases to 19 cases, which collapse further to six cases as follows. In cases I and VI we argue that whoever executes the last D cannot proceed. In cases II, III and V, we argue that no one can proceed until the last D has been executed. In case IV we argue that whatever happens after the D_i is unaffected by the following D_j .

(I) Sequences that end in the fetch-store sequence C_iD_i ($A_iB_iA_jB_jC_jD_jC_iD_i$, $A_iA_jB_iB_jC_jD_jC_iD_i$, $A_iA_jB_jC_jB_iD_jC_iD_i$, $A_iA_jB_jC_jD_jB_iC_iD_i$) forbid E_i , since after D_i , $S_i=S_j+1$. Similarly, sequences that end in C_jD_j ($A_iB_iC_iD_iA_jB_jC_jD_j$, $A_iB_iC_iA_jD_iB_jC_jD_j$, $A_iB_iA_jB_jC_iD_iC_jD_j$, $A_iA_jB_iC_iD_iB_jC_jD_j$, $A_iA_jB_iB_jC_iD_iC_jD_j$) forbid E_j .

(II) Sequences ending in $C_jD_jD_i$ ($A_iB_iC_iA_jB_jC_jD_jD_i$, $A_iB_iA_jB_jC_iC_jD_jD_i$, $A_iA_jB_iC_iB_jC_jD_jD_i$, $A_iA_jB_iB_jC_iC_jD_iD_j$) forbid E_j before D_i because the C_jD_j sequence sets S_j to $1+S_i$. After D_i , process j will be enabled to enter its critical section if and only if process i is not by Lemma 2.1. Similarly, $A_iA_jB_jC_jB_iC_iD_iD_j$ forbids E_i before D_j .

(III) Sequences that end in $C_iC_jD_iD_j$ ($A_iB_iA_jB_jC_iC_jD_iD_j$, $A_iA_jB_iB_jC_iC_jD_iD_j$) succumb to the argument in (II) since C_i and C_j commute.

(IV) In $A_iB_iC_iA_jB_jC_jD_iD_j$, the D_j stores the same value that B_j stored, since A_j and C_j are not separated by any stores due to process i . Thus the decision made by process i will not later be invalidated by the store D_j .

(V) In $A_iA_jB_iC_iB_jC_jD_iD_j$, the A_j and the C_i fetches each take place while the other process is between A and B . Hence each will fetch the same value, namely 0, and so store the same value, namely 1, via B_j and D_i respectively. This forbids E_i before D_j . (Notice that this argument breaks down if S is not constrained to be 0 when R is 0, and indeed the algorithm is incorrect without this precaution.)

(VI) In $A_iA_jB_jC_jB_iC_iD_jD_i$, the A_j and C_j fetches both take place while i is between A_i and B_i . Hence B_j and D_j both store 1. So C_i must fetch the 1 stored by B_j , so D_i will store 2. But after D_i , we have $S_i = S_j+1 = 1$, so i cannot proceed.

This completes the proof of Lemma 2.2. ▮

Lemma 2.3. Lockout of a non-failing process is impossible (i.e. (C3) holds).

Proof: We shall prove this for process P, letting P' denote the other process. Consider the transition diagram for P without its H transitions. The only loop is an E transition. Recall that we assumed that processes made progress. Hence if a process is locked out it must be continually taking that loop. Now eventually either P' makes an H transition or it settles down to taking its E' transition for ever. In the former case, after P' has taken the H transition, E (the condition evaluated when P takes transition E) becomes true, and now cannot be made false by any sequence of transitions allowed P' provided P stays in state 4. Thus the next transition P makes is to its critical section. In the latter case, once both processes are looping, as in the proof of Lemma 2.1, exactly one process will be permitted to enter its critical section. If P is the winner, we are done. Otherwise P' must eventually take an H transition, which reduces to the former case. ▮

(C4) holds:

Lemma 2.4. The algorithm has "linear waiting".

Proof: Suppose P is waiting at state 4. By the above proof, once P' has made an H transition, it cannot execute its critical section before P does. ▮

Trivially (C5) and (C6) hold by the construction. (C7) follows from Lemma 2.2, whose proof does not depend on the integrity of P'. We have now demonstrated (C1)-(C7).

The n-Process Solution.

We now solve the mutual exclusion problem for n processes where $n>2$, still heeding constraints (C1)-(C7). 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 constraint C4, which requires that no process be served twice while another is waiting. This uses the two-process solution as a subroutine, and in turn will itself be used as a subroutine to solve the n-process problem with all constraints in force. For notational convenience we say that process i is racing with process j when i and j are the two processes of the two-process solution and process i is executing the part of the code of that solution that precedes the critical section. When process i completes execution of that code we say it has won the race and beaten process j.

In the absence of constraint (C4), 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 approach 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 i races with process j, process i fetches from copy i of process j's data, and stores into copy j of its own data. Then when it has won against process j, it leaves copy j of its data as it is, preventing process j from later racing with and beating process i, and proceeds to race with process $j+1$. When it has beaten all the other processes, all $n-1$ copies of its R register are 2. It then executes its critical section; when done it sets all $n-1$ copies of its R register to 0, permitting processes waiting on process i to proceed.

A slight improvement to this scheme is for each process to have only one copy of the R and S registers, but to have in addition a U register whose contents names the process currently being raced with. Then the test for whether process i can proceed at the end of its race with j becomes

$$U_j=D \lor U_j<i \lor$$
$$(U_j=i \land (R_j=0 \lor S_j=1+S_i \lor ((R,S)_i=(R,S)_j \land i\le j)))$$

We arbitrarily favor the smaller-numbered process to break any ties. By making this test $i\le j$ the tie that must happen when a process races with itself allows that process to proceed. A dead process is assumed to have $U=D$, while a critical process has $U=n-1$. Actually the test for $R_j=0$ is now redundant, since this condition is implied by $U_j=D \lor U_j<i$.

Lemma 3.1. This algorithm satisfies C1-C3 and C5-C7 (i.e. all but C4).

Proof.

(C1) Mutual Exclusion. Suppose processes i and j are simultaneously in their critical sections. Then process i beat process j and vice versa while the respective processes were leading up to their critical sections. Suppose without loss of generality that i beat j before j beat i. If j beat i before i increased U_i to $j+1$, this would contradict the correctness of the two-process solution. If j beat i after i increased U_i to $j+1$, this would contradict j's being stopped by the failure of both $U_i<j$ and $U_i=j$.

(C2,C3) No Deadlock or Lockout. Assume there exists a 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 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. For suppose it does cycle; then either 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 C2-C3 hold for the two-process solution. So assume the cycle is of length 3 or more. Let i,j,k be three consecutive processes in the enumeration such that k is the least-numbered process of the three. (Choosing k to be the least-numbered in the cycle will ensure this.) Then j has thus far raced only with processes 1 through k. Since $i>k$, j cannot have raced with i yet, contradicting i's being beaten by j.

(C5) No global variables. Self-evident.

(C6) Finite ranges. Self-evident.

(C7) Impotence of Repeated Failing. Follows from the proof of C2-C3. ▮

We turn our attention now to satisfying (C4). The problem is that while a lower-numbered process is working its way up through the ranks as it competes with 1,2,3,..., 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.

Our solution to this problem is modelled on that of Eisenberg and McGuire [5], who use a global variable to point to processes 0,1,2,...,n-1,0,... 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, we shall arrange for each of the waiting processes to maintain an up-to-date copy of a virtual clock. 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 value for the clock. However, it is possible to have at most two (consecutive) values in the system at any moment. We formalize this situation thus.

Window Hypothesis (WH). At all times no two system clocks (clocks owned by processes with U=Q) differ by more than 1 mod n. (Hence for each moment in time there must exist k such that for every process i with $U_i=Q$, $c_i \in [k,k+1]$.)

(Here and later, 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 $a \neq b$, $[a,b-1] \cap [b,a-1] = \{\}$.)

The general principles for maintaining this state of affairs are:

(i) Process i may not tick (increment its clock) unless for every live process j, $c_j \in [c_i,c_i+1]$.

(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.

Our implementation of (i) will have for each process not only a clock but a variable U_i just as in the previous algorithm. When process i is "hors de combat," $U_i=D$ (D is a constant denoting death). The condition for advancing one's clock is

$\forall j[U_j=D \vee c_j=c_i \vee c_j=c_i+1]$

Thus the main loop consists of waiting for this condition to come true, then doing $c_i:=c_i+1$.

When $c_i=i$, process 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 C1 (mutual exclusion), 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 $U_i \neq D$ before selecting a clock value. In this way the system cannot spin much further while your clock does not change. This raises the possibility of not meeting C7. The solution is as for the two-process solution - set your clock initially to some system clock before you set U, independently of selecting another clock value after setting U. 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 separate problem is that of the "phantom clock." A process may fetch a system clock, but not store it immediately. In the meantime everyone in the system leaves it, one way or another. Then another process attempts to join the system, finds no usable clock value, so chooses an arbitrary value. (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 final fetch-and-store one atomic 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.

Yet another problem exists. Suppose that at some time $c_i=2$, $U_i=n-1$, $c_j=0$ and $U_j=Q$ (possible because of potential delay between 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 process j is just about to increment its clock. Now suppose process 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, c_i will proceed to join the system and (for $n>3$) destroy the window hypothesis WH.

Our solution 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, immediately before the final assignment and after U_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 same synchronizing mechanism we proposed to use for the last assignment alone. After the middle assignment, no system clock will be able to advance more than three ahead of c_i (proved below), so for $n \geq 5$, the wrap-around problem we are trying to solve cannot occur.

Here is another problem. Suppose $c_j=1$ and $c_{j+1}=0$, and suppose c_i becomes 1 at the intermediate assignment. Now let j leave the system (say by dying) so that for the final assignment, i fetches j+1's clock, which is still 0. But before i can store this value, 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 process i, since the final value it fetched was one less than the intermediate one. In this case, process i can ignore the final value and keep the intermediate one.

One more problem, a 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 algorithm is then:

1. c_i := "max" system clock (to avoid hangup).

2. P. (Use previous algorithm, which inter alia sets U_i.)

3. c_i := "max" system clock (to update c_i).

4. c_i := "max" system clock, if this doesn't decrement c_i.

5. V. (Suffices to set U_i := Q - see below.)

6. Wait till all clocks of live ($U \neq D$) processes equal yours or are one larger than it.

7. If your clock points to you, execute your critical section, then set U to D and exit.

8. Increment your clock (mod n) and return to step 6.

With the algorithm 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 that is set during step 5, tested at step 7, and cleared during step 8.

We now present the complete algorithm. We assume that S is of type integer mod 3 and c_i 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 $n-1$. U_i takes on some prefix of the sequence of values $D, 0, 1, 2, \ldots, n-1, Q$, followed by D. U takes over some of the role of R, and the test $R_j=0$ is subsumed by the test $U_j=D \vee U_j<i \vee U_j=Q$. Hence we may assume that R takes on only two values, 1 and 2.

```
integer procedure sysclock;
  begin integer m,j;   m := D;
                  for j := 0 step 1 until n-1 do
                        if U_j=Q ∧ c_j+1≠m then m := c_j;
                  sysclock := if m≠D then m else 0
  end sysclock;

1: c_i := sysclock;

2: for j := 0 until n-1 do
   begin (R,S,U)_i := (1,1+S,j_j);
         (R,S)_i := (2,1+S_j);
         wait(U_j=D ∨ U_j<i ∨ U_j=Q ∨
            (U_j=i ∧ (S_j=1+S_i ∨ ((R,S)_i=(R,S)_j ∧ i≤j))))
   end;

3: c_i := sysclock;

4: m  := sysclock; if m≠c_i-1 then c_i := m;

5: U_i := Q; ok := false;

6: for j := 0 until n-1 do wait(U_j=D ∨ c_j=c_i ∨ c_j=c_i+1);

7: if c_i=i ∧ ok then
     begin <critical section>; U_i := D end
   else
8:   begin c_i := c_i+1; ok := true; go to 6 end
```

In the following proof of the correctness of this algorithm, we will do all arithmetic involving c_i's modulo n.

For technical reasons (namely when using WH as an induction hypothesis) we shall strengthen WH to include the following condition. No system clock c_k may be m when c_i is not in $[m-1, m]$ and process i is either in step 6 with $j>k$ or is between step 6 and the increment of step 8. This caters for the possibility that a process might join the system with a valid clock, only to have it invalidated a moment later as some process proceeds to increment its clock.

Lemma 3.2. After process i has executed step 2, $U_i \neq D$.

Proof. Indeed, U_i is set to 0 at the outset of step 2, and then increments up to $n-1$. ▮

Lemma 3.3. Assume WH. If $U_i=U_k=Q$ but $c_k=c_i-1$, cannot pass step 6 before c_i increments.

Proof. Evident from the code. The strengthened form of WH is needed here to avoid the case where j has already passed step 6 when i arrives in the system with $c_i+1=c_j$. (It is tempting here to argue that WH unstrengthened will suffice, since from the state described in the Lemma, j could legally proceed to invalidate WH, contradicting our assumption that WH held. The catch is that we want to use Lemma 3.3 in the inductive step of the proof of WH, and this sort of "looking into the future" would invalidate this use.) ▮

Lemma 3.4. Assume WH and that $n \geq 6$. After process i fetches some c_j in step 3, no system clock may reach c_j+4 before c_i changes again.

(If 6 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+4 when $n=5$..)

Proof. At the moment c_j is fetched, no process can have passed the test at step 6 permitting it to increase to c_j+2 (Lemma 3.3). Hence to reach c_j+4 a process must pass through step 6 three times, with three successively higher (modulo n) clock values. Because c_i remains constant (hypothesis) and $U_i \neq D$ (Lemma 3.2), the test at step 6 could not have succeeded all three times. ▮

Corollary 3.5. After process i completes step 3 and before any process can execute another instruction, every system clock lies in $[c_i-1, c_i+3]$.

Proof. By WH, when c_j was fetched all system clocks lay in $[c_j-1, c_j+1]$. By Lemma 3.4 and the fact that system clocks can only increase, they can only move on to as far as c_j+3. Immediately after the assignment of this c_j to c_i, we can say the same of c_i. ▮

Corollary 3.6. After process i completes step 3 but before c_i changes again, every system clock lies in $[c_i-1, c_i+3]$, and any system clock in $[c_i, c_i+3]$ that increments during this time will not subsequently be able to pass the test at step 6 during this time and so cannot increment more than once.

Proof. During step 3, after c_j has been fetched, every system clock is either in $[c_j-1, c_j+2]$ or cannot pass the test at step 6 on account of (at least) c_i. Assigning c_j to c_i then leaves us with every process either in $[c_i-1, c_i+2]$ or still unable to pass the test on account of c_i. The former can clearly now not increment past c_i+3 before being stopped at 6 on account of c_i. Moreover, any that increment to c_i+1 or beyond (and hence were initially in $[c_i, c_i+2]$) cannot pass the test at the next encounter of 6, again on account of c_i, and so can increment at most once. ▮

Lemma 3.7. 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 $U_i=Q$, and instructions that set U_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 8, and $U_i := Q$ at step 5. The first of these preserves WH since i has passed the test at step 4 (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 Corollary 3.6, which promises not only that the joining process is within the window but ensures that any process with a greater clock will be stuck at step 6 on account of the joining processbefore it can invalidate WH. ∎

Theorem 3.8. This algorithm satisfies C1-C7.

Proof.

(C1) 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.

(C2,C3) No Lockout or Deadlock. The only source of problems for the reliable process i are the wait at step 2, dealt with already by Lemma 3.1, the wait at step 6, and the goto at step 8. By WH, eventually all processes with U≠D 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 6, so eventually no such process will remain. Then i will pass the test at step 6. The goto at step 8 allows c_i to make progress, and this combined with setting ok means that eventually the test at step 7 will be satisfied, and i will enter its critical section.

(C4) 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 U≠D, 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.

(C5) No Global Variables. Self-evident.

(C6) Finite range. Self-evident.

(C7) Impotence of Repeated Failing. Follows from the proof of C2-C3.

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.

References

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

[2] Dijkstra, E. W., "Solution of a Problem in Concurrent Programming Control", CACM 8 (September 1965), p 569.

[3] Dijkstra, E. W., "Self-Stabilizing Systems in Spite of Distributed Control", CACM 17 (November 1974), p 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), p 999

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

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