Data Requirements for Implementation of N-Process Mutual Exclusion Using a Single Shared Variable

JAMES E. BURNS, PAUL JACKSON, AND NANCY A. LYNCH

Georgia Institute of Technology, Atlanta, Georgia

MICHAEL J. FISCHER

University of Washington, Seattle, Washington

AND

GARY L. PETERSON

University of Rochester, Rochester, New York

ABSTRACT An analysis is made of the shared memory requirements for implementing mutual exclusion of N asynchronous parallel processes in a model where the only primitive communication mechanism is a general test-and-set operation on a single shared variable. While two variable values suffice to implement simple mutual exclusion without deadlock, it is shown that any solution which avoids possible lockout of processes requires at least $\sqrt{2N} + \frac{1}{2}$ values A technical restriction on the model increases this requirement to $N/2$ values, while achieving a fixed bound on waiting further increases the requirement to $N + 1$ values. These bounds are shown to be nearly optimal, for algorithms are exhibited for the last two cases which use $\lfloor N/2 \rfloor + 9$ and $N + 3$ values, respectively All of the lower bounds apply a fortiori to the space requirements for weaker primitives, such as P and V, using busy waiting

Categories and Subject Descriptors D 4 1 **[Operating Systems]**: Process Management—*mutual exclusion*; D 4 2 **[Operating Systems]**: Storage Management F 1 1 **[Computation by Abstract Devices]**: Models of Computation, F 1.2 **[Computation by Abstract Devices]**: Modes of Computation—*parallelism*, F 2 **[Theory of Computation]** Analysis of Algorithms and Problem Complexity

General Terms Algorithms, Performance, Theory

Additional Key Words and Phrases: critical section, test and set, asynchronous processes, synchronization

1. *Introduction*

Concurrent processing by several asynchronous parallel processes differs from sequential processing in that the order in which the elementary steps of the various processes are executed is not predetermined but may depend on difficult-to-predict variables such as the relative speeds of the processes and external events such as interrupts and operator intervention. To prevent interference among the various

This material is based upon work supported by National Science Foundation Grants MCS 77-02474, MCS 77-15628, MCS 77-28305, MCS 79-24370, and MCS 80-03337, and by U S Army Research Office Contract DAAG 29-79-C-0155

Authors' present addresses. J E. Burns, Indiana University, Bloomington, IN 47401; P Jackson and N A Lynch, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, M. J Fischer, Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195, G L Peterson, Department of Computer Science, University of Rochester, Rochester, NY 14627

processes, one often designates certain sensitive sections of code in the various processes as "critical sections" which are never to be executed simultaneously by two or more processes. Such mutual exclusion of access to critical sections is provided by means of entry protocols and exit protocols, sections of code which a process executes before entering and upon leaving a critical section, respectively. It is the job of the protocols to ensure that only one process at a time is in a critical section and that any other process trying to enter a critical section waits. In addition, the protocols play a scheduling role in determining which of several contending processes is allowed to proceed.

In order to provide mutual exclusion at all, there must be some primitive operations for interprocess communication. Examples of communication mechanisms are shared memory with elementary read and write operations [5, 7, 10], shared memory with test-and-set operations [3], message channels [8], and P and V operations [6]. Given a set of primitive operations, the "critical section problem" is to find entry and exit protocols using those operations which ensure mutual exclusion and at the same time have various desirable scheduling and other properties. Thus there is not a single critical section problem but many, and an extensive literature has developed around this class of problems (see [3–5, 7, 10, 11, 18, 19] and others).

Much work on the critical section problem has been concerned with finding protocols for a particular model and proving that they possess certain desired properties. More recently, there has been interest in finding out not only what can be done with a particular set of primitives but also what cannot [3, 12, 14, 15]. To prove a negative result of the sort "no protocol exists such that ...," it becomes necessary to define carefully the model of computation so that it is clear what solutions are allowed. In Section 2 we present a formal model based on a general test-and-set communication primitive which borrows ideas from the models of [3, 15, 19].

Section 3 presents algorithms which define upper limits on the amount of shared memory (measured by counting the number of distinct values which it can assume) for three critical section problems. Deadlock-free mutual exclusion of N processes can be achieved with only two shared memory values. Lockout-free mutual exclusion requires at most $\lfloor N/2 \rfloor + 9$ values. Finally, mutual exclusion with bounded waiting is solvable with $N + 3$ values.

Lower bounds for the above problems are given in Section 4. Any algorithm solving deadlock-free mutual exclusion must use at least two shared memory values. Bounded waiting and lockout-free mutual exclusion must use at least $N + 1$ and $\sqrt{2N} + \frac{1}{2}$ values, respectively. If lockout-free mutual exclusion is further constrained to be "memoryless" (i.e., each process always executes the same trying protocol whenever it attempts to enter a critical section), then at least $N/2$ states are required. (All of our upper bound algorithms are memoryless.)

Section 5 contains technical open questions and directions for further investigation.

This study is part of a larger effort to determine resource requirements for implementation of different kinds of "distributed computation" behavior using different process-variable configurations. The models studied consist generally of several processes communicating by means of test-and-set operations on several shared variables [13]. Behavior studied includes that of a simple arbiter system [13] and fair and maximally utilized access to multiple copies of a resource [9], as well as "failure-immune" mutual exclusion [9]. The problem of mutual exclusion using a simple shared variable is thus the simplest of many related problems; in fact, it seems to be closely related to the critical section which is inherent in the test-and-set operation itself. However, our lower bound results imply that the special-purpose

critical section inherent in the test-and-set operation does not automatically and easily solve the general critical section problem.

2. *A Formal Model for Exclusion Problems*

Our model is a hybrid of the models of [3] and [15]. It may also be regarded as a special case (with slight modification) of the general model of [13], tailored to the problems of this paper.

2.1 Systems of Processes. We consider a set of asynchronous parallel processes with a single shared communication variable. Processes access the variable using a general test-and-set instruction which, in one indivisible step, fetches the contents of the variable and stores a new value which depends on the value fetched. Intuitively, a process consists of a program, a program counter, and an internal memory, which together define the action of the process. In considering lower bounds, the internal details of the process are unimportant, so in our model a process is simply a set of states with a transition function. For presenting the upper bounds, we specify the transition function using an ALGOL-like notation.

The desired exclusion behavior of a set of processes is specified in terms of sets of states comprising "regions." The critical region of a process is a set of states which that process can only occupy while no other process is in its own critical region. The remainder region encompasses the rest of the process states. In order to solve synchronization problems, however, it appears necessary that new states other than those in the critical and remainder regions be introduced into each process. Thus we include two other sets of states in the basic definition as follows.

A *process* is a triple $P = (V, X, \delta)$, where V is a set of values; X is a (not necessarily finite) set of states partitioned into disjoint subsets R, T, C and E, where R is nonempty; and the *transition function* δ is a total function, $\delta: V \times X$ with the following properties:

(a) $x \in R$, $v \in V$ imply $\delta(v, x) \in V \times (T \cup C)$;
(b) $x \in T$, $v \in V$ imply $\delta(v, x) \in V \times (T \cup C)$;
(c) $x \in C$, $v \in V$ imply $\delta(v, x) \in V \times (E \cup R)$;
(d) $x \in E$, $v \in V$ imply $\delta(v, x) \in V \times (E \cup R)$.

The set V is referred to as the *message variable*, and X is the set of *local states of process P*. R, T, C, and E are the *remainder region*, *trying region*, *critical region*, and *exit region of P*, respectively. A transition from (v, x) to $\delta(v, x)$ is a *step of process P*. Transitions described in (a) and (b) are called *trying transitions*, while those described in (c) and (d) are called *exit transitions*.

The trying region describes a set of states wherein a process is seeking admission to its critical region, as in [3, 15]. The exit region describes a set of states wherein a process has just left its critical region but for purposes of synchronization must execute a protocol before being permitted to return to its own computing task. Although the exit protocols in many algorithms are very simple (such as a single "V" operation), we do not wish to exclude more sophisticated protocols from our model, for we wish our lower bounds to be as generally applicable as possible. To our knowledge, we are the first to include exit regions in a formal model, and our upper bound algorithms illustrate some ways in which exit regions can be used. Conditions (a) and (c) above indicate that the actual computing steps of the original process being modeled are suppressed. All steps of interest in the present paper involve attempts to enter the critical region or return to the remainder region.

Condition (b) indicates that the process, having once decided to attempt entry into its critical region, is thereafter committed to continue trying until it succeeds. Condition (d) indicates that the process, once in its exit region, must remain in its exit region until it reaches its remainder region.

For N any natural number, let $[N]$ denote $\{1, \ldots, N\}$. For N any natural number, a *system of N processes* is a $(2N + 1)$-tuple $S = (V, X_1, \ldots, X_N, \delta_1, \ldots, \delta_N)$, where for each $i \in [N]$, $P_i = (V, X_i, \delta_i)$ is a process. The remainder region, trying region, critical region, and exit region of process P_i are denoted by R_i, T_i, C_i, and E_i, respectively.

An *instantaneous description* (i.d.) of S is an $(N + 1)$-tuple $q = (v, x_1, \ldots, x_N)$, where $v \in V$ and $x_i \in X_i$ for all $i \in [N]$; in this case we define $V(q) = v$. The functions δ_i of the individual processes have natural extensions to the set of i.d.'s of S, defined by $\delta_i(v, x_1, \ldots, x_N) = (v', x_1, \ldots, x_{i-1}, x', x_{i+1}, \ldots, x_N)$, where $\delta_i(v, x_i) = (v', x')$. We also use (ambiguously) the notation R_i, T_i, C_i, and E_i for the natural extensions of the denoted sets of states to corresponding sets of i.d.'s. For example, $(v, x_1, \ldots, x_N) \in R_i$ if and only if $x_i \in R_i$.

If S is a system of N processes, then any finite or infinite sequence of elements of $[N]$ is called a *schedule* for S. In a natural way, each schedule defines a "computation" of system S when applied to any i.d. q of S; namely, if $h = h_1, \ldots, h_k$ is a finite schedule for S, then

$$r(q, h) = \delta_{h_k}(\delta_{h_{k-1}}(\ldots \delta_{h_1}(q) \ldots))$$

is the *result of applying schedule h to i.d. q*. We say i.d. q' is *reachable* from i.d. q in S if for some schedule h, $r(q, h) = q'$. Process $i \in [N]$ *halts* in schedule h for S if i appears only finitely often in h. If i halts in h and q is an i.d., we define final(i, q, h) to be the internal state of process i when it halts. Formally, *final*$(i, q, h) = y$ if there exist an i.d. $q' = (v, y_1, \ldots, y_N)$ and schedules h_1, h_2, with h_1 finite, and $h = h_1h_2$ such than h_2 contains no occurrence of i, $r(q, h_1) = q'$, and $y_i = y$.

The correctness of our algorithms depends on certain assumptions about the scheduling of processes, namely, the assumption that no process halts anywhere except possibly in its remainder region. Schedules with this property are called admissible and are defined below.

Let S be a system of process and q an i.d. A schedule h is *admissible from q* if for all $i \in [N]$, if i halts in h, then final $(i, q, h) \in R_i$.

2.2 SYNCHRONIZATION PROBLEMS. We are now ready to provide careful definitions for synchronization problems. We list formal conditions that may be combined to make precise some of the informal synchronization problems found in the literature. In the remainder of this section, S denotes a system of N processes and q an i.d.

C1: *Mutual Exclusion.* q "violates mutual exclusion" if $q \in C_i \cap C_j$ for some $i \neq j$, $i, j \in [N]$. *S satisfies mutual exclusion starting at q* if no i.d. reachable from q in S violates mutual exclusion.

The next three properties refer to a process' progress through its protocols. P_i is *stuck* for q and h if for all (finite) prefixes h_1 and h_2 of h, $r(q, h_1)$ and $r(q, h_2)$ are in the same region of P_i.

C2: *Deadlock-free.* S is *deadlock-free starting from q* if for all reachable i.d.'s $q' \notin \bigcap_{i=1}^{N} R_i$ and all schedules h admissible from q', there is some P_i which is not stuck for q' and h.

This property ensures that the introduction of synchronization protocols does not cause the entire system to stop computing. In particular, the processes cannot all loop indefinitely in their trying or exit regions.

In Section 3.1 there is presented a description of a system which satisfies mutual exclusion and no deadlock, having two values for its message variable. It is also shown, in Section 4.1, that two values are required for any system satisfying deadlock-free mutual exclusion. Both the algorithm and the lower bound are proved using the model as presented so far, and the reader may wish at this point to read those sections.

Other properties of interest involve fairness of the system from the point of view of each individual process. We consider two such properties.

C3: *Lockout-free.* P_i can be "locked out" starting from q if there exist $q' \notin R_i$ reachable from q and a schedule h admissible from q' such that P_i is stuck for q' and h. S is *lockout-free* starting from q if no P_i can be locked out starting from q.

C4: *Bounded Waiting.* P_j "goes from remainder to critical at least k times" for q and $h = h_1 \cdots h_m$ if there are indices $0 \leq i_1 < j_1 < i_2 < \cdots < j_k \leq m$ such that $r(q, h_1 h_2 \cdots h_{i_l}) \in R_j$ and $r(q, h_1 h_2 \cdots h_{j_l}) \in C_j$, $1 \leq l \leq k$. P_i "k-waits" starting from q if there exists $q' \notin R_i$ reachable from q and a schedule h such that P_i is stuck for q' and h and for some $j \in [N]$, $j \neq i$, P_j goes from remainder to critical at least k times for q' and h. S satisfies *k-bounded waiting starting from q* if no P_i $(k+1)$-waits starting from q. S satisfies *bounded waiting starting from q* if S satisfies k-bounded waiting starting from q for some value of k.

In other words, in a system which satisfies bounded waiting, if a given process is not in its remainder region, there is a bound on the number of times any other process is able to enter its critical region before the given process changes regions.

Note that the schedule h is not required to be admissible. Given any schedule h which causes a violation of k-bounded waiting, we can find a new schedule h' which is admissible and also causes a violation of k-bounded waiting. This follows because a violation of k-bounded waiting (unlike a violation of lockout) occurs after a finite amount of time.

Note also that if S and q satisfy C2 and C4, then they satisfy C3 as well. Also, C3 implies C2.

Finally, the following property does not represent a requirement one would necessarily care to impose on a system of processes but is nevertheless a property shared by practically all known exclusion algorithms. Intuitively, a process does not use its past computation history to alter its synchronization protocols.

C5: *No Memory.* S satisfies *no memory* if for all $i \in [N]$, $|R_i| = 1$.

3. *Upper Bounds*

This section presents the upper bound results on the number of states of the shared variable required to solve the problems of deadlock-free, bounded waiting, and lockout-free mutual exclusion. The correctness of the algorithms is argued informally.

3.1 Deadlock-Free Mutual Exclusion. In order to illustrate the model, we give a detailed description of a very simple system satisfying C1, C2, and C5 and having two values for its message variable. (Here and in the corresponding lower bound in Section 4.1 we are merely formalizing well-known results.)

Algorithm

$$S = (V, \underbrace{X, X, \ldots, X}_{N}, \underbrace{\delta, \delta, \ldots, \delta}_{N}),$$

where $X = R \cup T \cup C \cup E$ as above. Here, $R = \{R_0\}$, $T = \{T_0\}$, $C = \{C_0\}$, $E = \emptyset$, and $V = \{0, 1\}$. q, the initial i.d., is

$$(0, \underbrace{R_0, R_0, \ldots, R_0}_{N}).$$

Transitions are

$$\begin{aligned} &\delta(0, R_0) = (1, C_0), \quad &&\delta(1, R_0) = (1, T_0), \\ &\delta(0, T_0) = (1, C_0), \quad &&\delta(1, T_0) = (1, T_0), \\ &\delta(0, C_0) = (0, C_0), \quad &&\delta(1, C_0) = (0, R_0) \end{aligned}$$

Verification by induction is straightforward. Note that S and q do not satisfy C3, since the schedule $(121)^\infty$ locks the second process out.

Thus, we have proved

THEOREM 3.1. *For each $N \geq 1$ there is a system S of N processes and an i.d. q such that S, q satisfy mutual exclusion (C1), are deadlock-free (C2), and use no memory (C5), and $|V| = 2$.*

3.2 HIGHER-LEVEL NOTATION FOR BOUNDED-WAITING AND LOCKOUT-FREE MUTUAL EXCLUSION ALGORITHMS. The remaining upper bounds will be shown by giving algorithms in an ALGOL-like notation, for understandability. States can be thought of as having components corresponding to internal variables and program instruction counters. Some state transformations are expressible implicitly by the usual flow of control of ALGOL programs; others (branching and alteration of values of internal variables) must be expressed explicitly.

Access to the shared variable V is allowed only with the test-and-set primitive, which has the following syntax.

⟨test-and-set⟩ ::= test ⟨variable⟩ until ⟨set⟩ {; ⟨set⟩} endtest|
test ⟨variable⟩ while ⟨set⟩ {; ⟨set⟩} endtest
⟨set⟩ ::= ⟨value$_1$⟩ setto ⟨value$_2$⟩ [: ⟨statement⟩]

The intended semantics of the first statement is to compare the ⟨variable⟩ to the ⟨value$_1$⟩ values, all of which must be distinct. If a match is found, the ⟨variable⟩ is set to the corresponding ⟨value$_2$⟩ value, the corresponding statement (which represents a state change) is executed, and control passes to the next test-and-set. If no match is found, the test-and-set is reexecuted from the beginning (busy-waiting). Similarly, the semantics of the second statement is to compare the ⟨variable⟩ to the (disjoint) ⟨value$_1$⟩ values. If a match is found, the ⟨variable⟩ is set to the corresponding ⟨value$_2$⟩ value, the corresponding statement is executed, and control passes back to the beginning of the same test-and-set. If no match is found, control passes to the next test-and-set.

There are other features in the algorithm that are not present in standard ALGOL, but these should be transparent to the reader. For example, the symbols "⌊ ⌋" are used for the floor function. The "exit" statement is used to escape from the closest enclosing "while" loop.

It should now be straightforward to translate the next two algorithms into the basic model. Statements and tests not involving V will be absorbed into internal state changes in the basic test-and-sets in the translated algorithm.

3.3 MUTUAL EXCLUSION WITH BOUNDED WAITING. In this section we prove the following theorem by exhibiting Algorithm B. We first present Algorithm A which is somewhat simpler but uses a few more states.

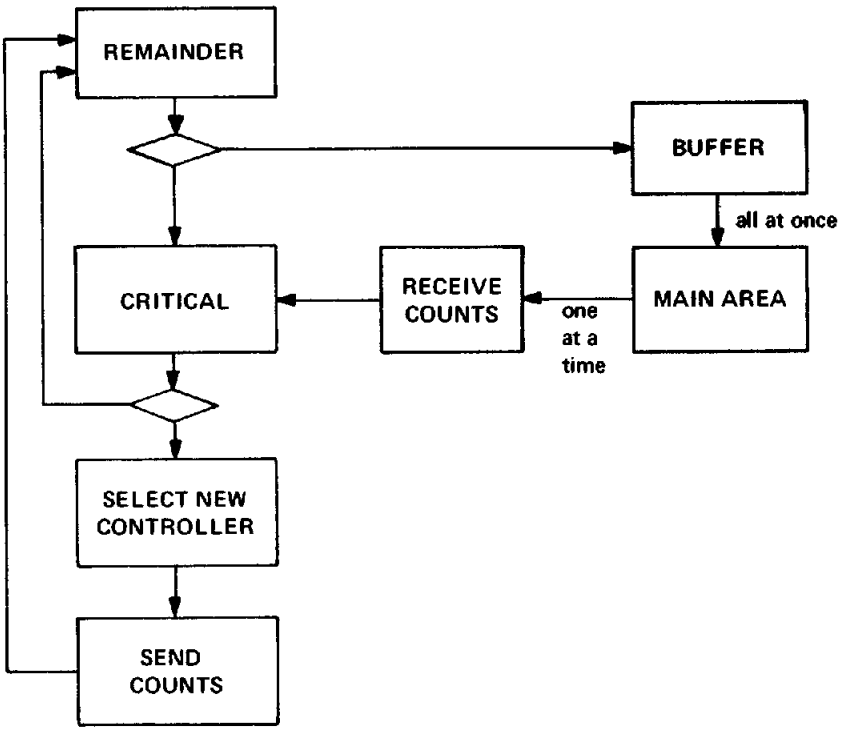

FIGURE 1

THEOREM 3.2. *For each $N \geq 1$ there is a system S of N processes and an i.d. q such that S, q satisfy mutual exclusion ($C1$), are deadlock-free ($C2$), have bounded waiting ($C4$), and no memory ($C5$), and $|V| = N + 3$. Moreover, $k = 1$ in the bound of $C4$.*

Algorithm A below satisfies the conditions of the theorem except that there are $N + 6$ values of the shared variable. We later indicate how to reduce this to $N + 3$.

The basic structure of the algorithm is the same as that in [4]. A process desiring to enter its critical region goes in immediately if there are no other active processes; otherwise it waits in the "buffer." Eventually, all processes waiting in the buffer are moved into the "main area." Processes are chosen one at a time from the main area to go to their critical regions (see Figure 1). Since no process may enter the main area until the main area is emptied, this procedure gives 1-bounded waiting.

The above procedure requires a mechanism for controlling the movement of processes through the buffer and main area. As each process leaves its critical region (i.e., while it is in its exit region), it is temporarily designated the "controller." The controller has the responsibility of keeping track of the number of processes in the buffer and main area, sending messages to cause processes to move, and passing on the necessary control information to the next designated controller. All this is done through the single shared variable V which takes on the values {$S0$, $S1$, ..., $SN - 1$, FREE, ENTER, ELECT, COUNT, ACK, BYE}. The last five values are called "messages."

A process desiring to enter its critical region examines V. If V = FREE (indicating that the system is empty), then the process sets V to $S0$ and enters its critical region. If $V = Sj$, the process sets V to $Sj + 1$. The S-values of V are thus used by the controller to keep track of the count of the number of processes in the buffer. (This count is kept in the controller's local variable, BUFF.)

The controller loads the main area, when empty, by sending one ENTER message through V for each process in the buffer. If additional processes come into the buffer during this time, they too are moved into the main area. The controller selects the process to become the next controller by sending a single ELECT message, which will be picked up by some process in the main area. The controller then sends the current counts of the number of processes in the buffer and main area to the controller-to-be before signing off with the BYE message. (Note: In the special case in which there are no processes in the buffer or main area and the process leaving its critical region sees $V = S0$, the process simply sets V to FREE and leaves; the system has been returned to the empty state.)

An apparent problem with the above scheme is possible interference between processes entering their trying regions and processes attempting to communicate using messages. A process entering its trying region should alter V; otherwise, since the other processes would be unaware of it, they could execute any number of critical regions before the first process could get in. Thus, processes entering their trying regions might hinder communication between the controller and the other processes. In [4], about $2N$ values of V are used to allow message communication to go on concurrently with the counting function. We solve the problem in the following way. Every message requires a response (usually ACK). While awaiting the response, the controller "normalizes" V. That is, it continually examines V, resetting it to $S0$ whenever Sj is detected (and keeping track of the number of new processes in the buffer by setting BUFF to BUFF + j). If an entering process sees a message in V, it holds this value and sets V to $S1$, thus announcing its presence to the controller. It then waits until V takes on the value $S0$, at which point it resets V to the held message. V will eventually "settle down" at $S0$ since the controller continues to normalize V to this value, and the value can only be changed a finite number of times by entering processes.

The key to the correctness of the algorithm is the ability to communicate information among the processes reliably. All communication (except sending the buffer count) is initiated by a message sent by the controller, which then waits for an ACK response. Only a process of the appropriate type will respond, and the controller knows of the existence of such a process by the controller's local state. The existence of a normalizing process (a process which continues to normalize V) ensures that messages always get through. Note that the controller itself is always a normalizing process unless it is in its last test-and-set loop. But at this point, a process has already been elected and moved to RECEIVECOUNTS, where it becomes a normalizing process.

For simplicity, the counts of the number of processes in the buffer and main area are sent to the new controller in unary. The main area count is sent with the COUNT message, while the buffer count is sent by incrementing the $S0$ value of V. It should be clear that more "time efficient" methods could be used for passing this information. We are primarily interested in presenting a clear algorithm which is compatible with the more complex algorithm given in Section 3.4.

Algorithm A

Shared variable msgvar V initial FREE,

Algorithm for each process

```
begin integer MAIN, BUFF initial 0;
      msgvar M                initial S0;
```

```
REMAINDER    ' remainder region '
' trying protocol follows '
 test V until
    Free  setto S0      goto CRITICAL,
    Sj    setto Sj + 1  goto BUFFER,   ' j < N - 1 '
    other setto S1      M = other
 endtest,
HOLDING
  test V until S0 setto M  M := S0 endtest,
BUFFER:
  test V until ENTER setto ACK endtest,
MAINAREA
  test V until ELECT setto ACK endtest,
RECEIVECOUNTS
  while true do
    test V until
      Sj      setto S0    BUFF = BUFF + j,
      COUNT setto ACK  MAIN = MAIN + 1,
      BYE     setto S0    exit
    endtest,
CRITICAL    ' critical region '
' exit protocol follows !
 if (MAIN = 0 and BUFF = 0) then
    test V until
      S0 setto FREE  goto REMAINDER,
      Sj setto S0      BUFF = BUFF + j   ' 1 ≤ j ≤ N - 1 !
    endtest,
SELECT NEW CONTROLLER
  if MAIN = 0 then   ! move processes from buffer to main area '
    while BUFF > 0 do begin
      test V until
        Sj setto ENTER: (BUFF = BUFF + j - 1, MAIN = MAIN + 1)
      endtest,
      while true do
        test V until
          Sj    setto S0. BUFF = BUFF + j,
          ACK setto S0  exit
        endtest
    end,
  test V until
    Sj setto ELECT  (BUFF = BUFF + j, MAIN = MAIN - 1)
  endtest,
  while true do
    test V until
      Sj    setto S0: BUFF = BUFF + j,
      ACK setto S0  exit
    endtest,
SENDCOUNTS
  while MAIN > 0 do begin
    test V until S0    setto COUNT  MAIN = MAIN - 1 endtest,
    test V until ACK setto S0 endtest
  end,
  while BUFF > 0 do
    test V until S0 setto S1  BUFF = BUFF - 1 endtest,
  test V until S0 setto BYE endtest,
  go to REMAINDER
end
```

We now sketch how to modify Algorithm A to use only $N + 3$ shared values. One value is saved by equating FREE with $SN - 1$. These two values can never be confused, since $SN - 1$ can only occur with no process in its remainder region, while FREE can only occur when all processes are in their remainder regions. The values COUNT and BYE can be eliminated by modifying the sections of code labeled "RECEIVECOUNTS" and "SENDCOUNTS," as shown below. Both counts are sent as a single coded integer by using the S-values. Note that the first waiting loop in the section "RECEIVECOUNTS" is required in Algorithm B to be certain that the ACK response to the ELECT message has been seen by the controller. Otherwise the receiving process might mistake this ACK for the final ACK sent by the controller.

Algorithm B

Replace the indicated sections of Algorithm A by the following code Note that COUNT and BYE are no longer used so that the number of shared values is reduced to $N + 3$ (FREE is equated with $SN - 1$)

```
RECEIVECOUNTS
  while BUFF < N do
    test V until Sj setto S0. BUFF := BUFF + j endtest,
  while true do
    test V until
      Sj    setto S0  BUFF = BUFF + j,
      ACK setto S0  exit
    endtest,
  MAIN := ⌊BUFF/N⌋ − 1,
  BUFF = BUFF − (MAIN + 1)*N,
SENDCOUNTS
  BUFF = BUFF + (MAIN + 1)*N,
  while BUFF > 0 do
    test V until S0 setto S1  BUFF = BUFF − 1 endtest;
  MAIN := 0;
  test V until S0 setto ACK endtest;
  goto REMAINDER
```

3.4 LOCKOUT-FREE MUTUAL EXCLUSION. If we drop the requirement of bounded waiting and ask only for a lockout-free solution, the number of states needed to achieve mutual exclusion can be cut roughly in half, as shown by Algorithm C. We thus obtain

THEOREM 3.3. *For each $N \geq 1$ there is a system S of N processes and an i.d. q such that S, q satisfy mutual exclusion ($C1$), are lockout-free ($C3$), and have no memory ($C5$), and $|V| = \lfloor N/2 \rfloor + 9$.*

In Algorithm C the shared variable V takes on the $\lfloor N/2 \rfloor + 9$ values $S0$, $S1$, ..., Sk, FREE, ENTER, ELECT, COUNT, ACK, BYE, STOP, GO, where $k = \lfloor N/2 \rfloor$. Since there are fewer values of V than processes (for sufficiently large N), the count of entering processes cannot be kept unambiguously in V. In particular, more than k processes entering their trying regions closely together will cause the transition of V from Sk to $S0$. We call this transition "wraparound." The process causing this transition is called the "executive." Since only the executive knows that wraparound has occurred, it has the responsibility to see that those processes which were not able to announce their presence unambiguously will eventually get to their critical regions.

Note that the occurrence of wraparound is what causes the loss of the 1-bounded waiting property. For example, suppose one process goes critical from an i.d. at

which all processes are in remainder, setting V to $S0$. Next, exactly k other processes take one step each, entering their trying regions and returning the value of V to $S0$. Now the first process may leave the critical region and, since it cannot detect that any process is waiting, cycle from remainder to critical any number of times. Thus the algorithm can violate bounded waiting for any bound.

The executive knows that there are k processes in the buffer which are unknown to the controller. These processes (and possibly some others which are incidentally detected by the executive) are suspended by sending STOP signals to each. If the executive sees a controller message during this process, it merely holds the message value until the stopping procedure is complete and then restores the held value. The executive then announces its presence to the controller in the normal way (incrementing the Si value) and enters the buffer. (If there is no controller, the executive goes directly to its critical region.)

Note that the sending of STOP signals by the executive does not interfere with the operation of the controller, since the controller ignores these signals. The only possible interference occurs when the executive "picks up" a controller message. However, by holding onto this message the executive effectively suspends the operation of the controller until the executive finishes its task. The executive thus never needs to hold more than one controller message.

Once at least k processes have been suspended by the executive, Algorithm C behaves identically to Algorithm A. Thus the executive eventually reaches its critical region. When leaving its critical region, the former executive (now a controller) sends a GO message to each process which was suspended, causing it to go to the main area. Assuming that no additional wraparounds (and hence no additional executives) can have occurred at this time, all the processes in the main area must get to their critical regions before any other processes can enter the main area. This guarantees that lockout is prevented.

Difficulties could arise if two executives were present at the same time, because their messages would be indistinguishable, which could lead to lockout for one of the executives. This problem cannot arise in our algorithm because there are insufficient processes to cause another wraparound until the current executive finishes its task, goes critical, and reawakens the idling processes with GO messages.

There is an apparent danger of lockout for processes which are already in MAINAREA when the executive begins moving idling processes to MAINAREA. However, since a new executive must go through BUFFER, and since all the processes in the MAINAREA must reach their critical regions before any process moves from BUFFER to MAINAREA, no process can be stuck in MAINAREA.

Algorithm C

Replace the first sections of Algorithm A, (up to MAINAREA) with the following code.

```
begin integer MAIN, BUFF, IDLERS initial 0,
      msgvar M                    initial S0,
REMAINDER   ' remainder region '
' trying protocol follows '
 test V until
    FREE setto S0      goto CRITICAL,
   Sj     setto Sj + 1 goto BUFFER,      ' 0 ≤ j < ⌊N/2⌋ '
   Sk     setto S0     goto EXECUTIVE,   ' k = ⌊N/2⌋ '
   STOP   setto S1     goto IDLE,
   other  setto S1     (M = other, goto HOLDING)
 endtest,
```

```
EXECUTIVE:
  BUFF := ⌊N/2⌋;
  while BUFF > 0 do
    test V until
      Sj     setto STOP. (BUFF := BUFF + j − 1; IDLERS := IDLERS + 1);
      STOP setto STOP,
      FREE setto S0: goto CRITICAL;
      other  setto S0  M := other
    endtest,
  test V while STOP setto STOP endtest,
  if M ≠ S0 then
    test V until S0 setto M: M = S0 endtest,
  test V until
    FREE setto S0      goto CRITICAL,
    Sj     setto Sj + 1 goto BUFFER,
    other  setto S1    . M := other
  endtest,
HOLDING:
  test V until
    S0     setto M  (M := S0, goto BUFFER),
    STOP setto M  (M = S0, goto IDLE)
  endtest,
BUFFER
  test V until
    ENTER setto ACK: goto MAINAREA,
    STOP   setto S0    goto IDLE
  endtest,
IDLE:
  test V until GO setto ACK  goto MAINAREA endtest,
Insert the following code after "1 exit protocol follows !"
while IDLERS > 0 do begin
  test V until Sj setto GO; BUFF := BUFF + j endtest;
  IDLERS = IDLERS − 1,
  MAIN   := MAIN   + 1,
  while true do
    test V until
      Sj     setto S0  BUFF := BUFF + j,
      ACK setto S0: exit
    endtest
  end;
```

4. *Lower Bounds*

We present five lower bound theorems, using a series of lemmas along the way. Dependence on properties C1–C5 is described explicitly for each result. Proofs are usually by contradiction; assuming there are too few values of V, a schedule is constructed which violates one of the needed conditions.

4.1 Deadlock-Free Mutual Exclusion. We give three lemmas leading to a lower bound which complements Theorem 3.1. First we show that it is always possible, from any i.d., to "drive" all processes into their remainder regions.

Lemma 4.1. *Let S be a system of N processes, $N \geq 1$, and q any i.d. Assume that S, q are deadlock-free (C2). Let $L = \{i \in [N]: q \notin R_i\}$. Then there exists a schedule $h \in L^*$ such that $r(q, h) \in \bigcap_{i=1}^{N} R_i$.*

Proof. By induction on $|L|$. If $|L| = 0$, there is nothing to prove. If $|L| = k + 1$, let $l_1, \ldots, l_{k+1}$ be some enumeration of L. $h_1 = (l_1 \cdots l_{k+1})^\infty$ is admissible from

q, so by C2 there is some prefix h_2 of h_1 with $q' = r(q, h_2) \in R_l$ for some $l \in L$. Then let $M = L - \{l\}$. By the inductive hypothesis there exists schedule $h_3 \in M^*$ such that $r(q', h_3) \in \bigcap_{i=1}^{N} R_i$. Then $h = h_2 h_3$ suffices. □

Next, we show that if all processes are in their remainder regions and a single process acts alone, then that process will eventually reach its critical region.

LEMMA 4.2. *Let S be a system of N processes, $N \geq 1$, $q \in \bigcap_{i=1}^{N} R_i$. Assume that S, q are deadlock-free (C2). Let $i \in [N]$. Then for some $k \geq 1$, $r(q, i^k) \in C_i$.*

PROOF. $q' = r(q, i) \in T_i \cup C_i$. If $q' \in C_i$, we are done. Otherwise, i^∞ is admissible from q'. The conclusion follows by C2. □

The next lemma says that if a process can, on its own, enter both its remainder and its critical region, then it must indicate the distinction to the other processes by means of distinct values of V. We say that i.d. q *looks like* i.d. q' *to process* i if and only if $V(q) = V(q')$ and the state of process i is identical in q and q'.

LEMMA 4.3. *Let S be a system of N processes, $N \geq 2$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1) and are deadlock-free (C2). Let $i \in [N]$. Assume $q' = r(q, i^k) \in R_i$ and $q'' = r(q, i^l) \in C_i$ for some $k, l \geq 0$. Then $V(q') \neq V(q'')$.*

PROOF. Assume that $V(q') = V(q'')$. $h = (1\ 2 \cdots (i-1)(i+1) \cdots N)^\infty$ is admissible from q', so by C2 there is some prefix h_1 of h with $r(q', h_1) \in C_j$ for some $j \neq i$. But q'' looks like q' to processes in $[N] - \{i\}$, so $r(q'', h_1) \in C_j$ also. But then $r(q'', h_1) \in C_i \cap C_j$, contradicting C1. □

We combine the preceding lemmas to obtain the lower bound result corresponding to Theorem 3.1.

THEOREM 4.4. *Let S be a system of N processes, $N \geq 2$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1) and are deadlock-free (C2). Then $|V| \geq 2$.*

PROOF. Obtain $q' = r(q, h) \in \bigcap_{i=1}^{N} R_i$. Obtain k with $q'' = r(q', 1^k) \in C_1$. Then $V(q'') \neq V(q')$ (by Lemma 4.3 applied to S, q'). □

Later we will require explicit names for the schedules whose existence is asserted in Lemmas 4.1 and 4.2. In the applications the system S will generally be considered fixed. Thus we define the following. Let S be a system of N processes, $N \geq 1$. Let q be any i.d. such that S, q are deadlock-free. Let $L = \{i \in [N] : q \notin R_i\}$. Then exit($q$) denotes a schedule $h \in L^*$ such that $r(q, h) \in \bigcap_{i=1}^{N} R_i$. Also, if S is a system of N processes, $N \geq 1$, $q \in \bigcap_{i=1}^{N} R_i$ with S, q deadlock-free, and if $i \in [N]$, then enter(q, i) denotes a schedule i^k, $k \geq 1$, such that $r(q, i^k) \in C_i$.

4.2 MUTUAL EXCLUSION WITH BOUNDED WAITING. Next we turn to the proof of a lower bound to complement Theorem 3.2. Although our best lower bound is $N + 1$ values (Theorem 4.9), we first give a much simpler proof for a lower bound of N values. The small strengthening to $N + 1$ is then carried out, partly because the remaining gap between upper and lower bounds is extremely small, but principally because the proofs use interesting ideas which recur in proofs of later results in Section 4.3.

THEOREM 4.5. *Let S be a system of N processes, $N \geq 1$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1), are deadlock-free (C2), and satisfy bounded waiting (C4). Then $|V| \geq N$.*

PROOF. Construct $\{q_i\}_{i=0}^{N}$ a sequence of i.d.'s as follows. Let $q_0 = r(q, \text{exit}(q))$. (That is, let all processes exit.) Let $q_1 = r(q_0, \text{enter}(q_0, 1))$. (That is, run P_1 until it enters its critical region.) For each i, $2 \le i \le N$, let $q_i = r(q_{i-1}, i) \in T_i$. (That is, let each process $P_2, \ldots, P_N$ in turn enter its trying region.) We show that $V(q_i) \neq V(q_j)$ for all $0 < i < j \le N$.

Assume the contrary, so $0 < i < j \le N$ and $V(q_i) = V(q_j)$. Then q_j looks like q_i to processes $P_1, \ldots, P_i$. Since there is an admissible schedule h from q_i which involves $P_1, \ldots, P_i$ only and which causes some process to enter its critical region an infinite number of times, it follows that h (although not admissible from q_j) causes the same effect when applied from q_j. But this violates C4, since P_j remains in its trying region during the application of h from q_j. □

Next we develop the new ideas needed to raise the lower bound to $N + 1$ values. We require a nontrivial lemma giving a lower bound of 3 on the number of values needed for synchronization of two processes. The needed lemma is a slight strengthening of a similar result in [3] and is proved by a very similar case analysis argument. We break the lemma into two parts. The first part says that a bound of 2 on the number of values taken on by V when some process is not in its remainder region imposes some strong restrictions on the behavior of the two processes. Namely, if one process is unable, on its own, to reach its remainder region and is unable to signal reliably to the other process, then the other process is similarly unable, on its own, to reach its remainder region.

We say that a process P_i is *blocked* for q if for all $k \ge 0$, $r(q, i^k) \notin R_i$. That is, a blocked process cannot reach its remainder region on its own.

LEMMA 4.6 (CREMERS AND HIBBARD). *Let S be a system of two processes and q any i.d. Assume that S, q satisfy mutual exclusion (C1) and no lockout (C3). Assume there exist $v_1, v_2 \in V$ such that $V(q') \in \{v_1, v_2\}$ for all $q' \notin R_1 \cap R_2$ reachable from q. Assume P_1 is blocked for q and for infinitely many k it is the case that $V(r(q, 1^k)) = V(q)$. Then P_2 is blocked for q.*

PROOF. Assume the contrary, that $q' = r(q, 2^k) \in R_2$ for some fixed $k \ge 0$. Then $V(q') \in \{v_1, v_2\}$ since $q' \notin R_1$. Assume without loss of generality that $V(q) = v_1$.

Case 1. $V(q') = v_1$. Since 1^∞ is admissible from q', C2 implies that $r(q', 1^l) \in R_1$ for some $l \ge 0$. But then since q' looks like q to P_1, it follows that $r(q, 1^l) \in R_1$, a contradiction.

Case 2. $V(q') = v_2$.

Case 2.1. There are infinitely many l for which $V(r(q', 2^l)) = v_1$. Then an admissible schedule from q' can be constructed by alternating groups of steps of P_1 and P_2 which leave V at v_1; this schedule locks out P_1, contradicting C3.

Case 2.2. $V(r(q', 2^l)) = v_2$ for all but finitely many l. Choose l^* so that P_2 is blocked for $q'' = r(q', 2^{l^*})$ and $V(r(q'', 2^l)) = v_2$ for all $l \ge 0$. (P_2 must become blocked at some point, by Lemma 4.3, once the value of V stops changing.) Since q'' looks like q' to P_1, it is the case that $V(r(q'', 1^m)) = V(r(q', 1^m))$ for all $m \ge 0$ and thus $V(r(q', 1^m)) \in \{v_1, v_2\}$ for all $m \ge 0$. Since P_1 can cycle through its critical section infinitely many times on its own from q' (by C2), there must be infinitely many m such that $V(r(q', 1^m)) = v_2$, by Lemma 4.3. But then we can construct an admissible schedule from q'' composed of alternating groups of steps of P_1 and P_2 which leave V at v_2 and locks out P_2, thus contradicting C3. □

In two cases in the preceding proof (2.1 and 2.2) a technique of "piecing together" infinite schedules of several processes was introduced. This technique will be useful in later proofs (in Section 4.3) as well for construction of admissible schedules exhibiting lockout.

Using the preceding lemma repeatedly, we show the needed lower bound on the number of values needed for synchronization of two processes.

LEMMA 4.7 (CREMERS AND HIBBARD). *Let S be a system of two processes and q any i.d. Assume that S, q satisfy mutual exclusion (C1) and are lockout-free (C3). Then there do not exist v_1, v_2 with $V(q') \in \{v_1, v_2\}$ for all $q' \notin R_1 \cap R_2$ reachable from q.*

PROOF. Assume the contrary. Obtain $q' \in C_2$ with $r(q', 1^k) \in T_1$ for all $k \geq 0$ and with $V(r(q', 1^k)) = V(q')$ for infinitely many k. By Lemma 4.6, it is the case that P_2 is blocked for q'. Fix $k \geq 1$ with $q'' = r(q', 2^k)$ satisfying $V(r(q'', 2^l)) = V(q'')$ for infinitely many l. (That is, move P_2 at least one step, until it sets V to a value which P_2 can reproduce infinitely often.) Then by Lemma 4.6 (applied with P_1 and P_2 interchanged), it is the case that P_1 is blocked for q''. Alternately applying Lemma 4.6 to the two processes in this way, we construct an admissible schedule which contradicts C2. □

COROLLARY 4.8 (CREMERS AND HIBBARD). *Let S be a system of two processes and q any i.d. Assume that S, q satisfy C1 and C3. Then $|V| \geq 3$.*

PROOF. Immediate. □

We can now prove the lower bound of $N + 1$.

THEOREM 4.9. *Let S be a system of N processes, $N \geq 2$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1), are deadlock-free (C2), and satisfy bounded waiting (C4). Then $|V| \geq N + 1$.*

PROOF. Since C2 and C4 together imply C3, Corollary 4.8 gives the result for $N = 2$. Assume $N \geq 3$. Construct $\{q_i\}_{i=0}^{N}$ as in the proof of Theorem 4.5. Assuming $|V| \leq N$, one of the following cases must hold.

Case 1. $V(q_i) = V(q_j)$ for some $0 < i < j \leq N$. Then the proof of Theorem 4.5 provides the needed contradiction.

Case 2. $V(q_0) = V(q_i)$ for some $0 < i < N$. Since $r(q_0, N^m) \in C_N$ for some $m \geq 1$ (by C2), it follows that $r(q_i, N^m) \in C_N$. But $r(q_i, N^m) \in C_1$, violating C1.

Case 3. $V(q_0) = V(q_N)$ and cases 1 and 2 do not hold. By Lemma 4.7 there is some schedule h involving P_1 and P_2 only with $q' = r(q_0, h) \notin R_1 \cap R_2$ and $V(q') \notin \{V(q_1), V(q_2)\}$. There are two possibilities.

Case 3.1. $V(q') = V(q_0)$. Then q' looks like q_0 to P_3. Since the schedule 3^∞ causes P_3 to enter its critical region infinitely often when applied from q_0 (by C2), it does the same when applied from q'. This violates C4, since one of (P_1, P_2) remains meanwhile in some region other than its remainder region.

Case 3.2. $V(q') = V(q_i)$ for some i, $3 \leq i \leq N$. Then q' looks like q_i to processes $P_{i+1}, \ldots, P_n$. Let $q'' = r(q', (i + 1)(i + 2) \cdots (N))$. (That is, allow each of $P_{i+1}, \ldots, P_N$ in turn to enter its trying region.) Then q'' looks like q_0 to P_3, since $V(q'') = V(q_N) = V(q_0)$. Thus the schedule 3^∞ causes P_3 to enter its critical region infinitely often when applied from q'', violating C4 since $q'' \notin R_1 \cap R_2$. □

Note that although this section of the paper was aimed at a lower bound for mutual exclusion with bounded waiting, the two main lemmas, 4.6 and 4.7, make statements involving mutual exclusion with no lockout.

4.3 Lockout-Free Mutual Exclusion. We have two lower bound results corresponding to Theorem 3.3. The first does not depend on any extra assumptions but leaves a gap open. The second depends on the introduction of the technical assumption C5 but essentially closes the gap.

Theorem 4.10. *Let S be a system of N processes, $N \geq 2$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1) and are lockout-free (C3). Then $|V| \geq \sqrt{2N} + \frac{1}{2}$.*

Proof. We show by induction on k that for $k \geq 3$, if S is any system of $(k^2 - k)/2 - 1$ or more processes and q any i.d. such that S, q satisfy C1 and C3, then $|V| \geq k$. The theorem then follows immediately.

For $k = 3$, Corollary 4.8 gives the result. For the induction step, let

$$N \geq \frac{(k+1)^2 - (k+1)}{2} - 1,$$

let S be a system of N processes, and let q be an i.d. such that S, q satisfy C1 and C3, and assume contrary to the induction hypothesis for $k+1$ that $|V| < k+1$. We proceed to derive a contradiction.

Construct $\{q_i\}_{i=0}^{N}$ as follows. Let $q_0 = r(q, \text{exit}(q))$. Let $q_1 = r(q_0, \text{enter}(q_0, 1))$. (These are as for Theorem 4.5.) For each i, $2 \leq i \leq N$, let $q_i = r(q_{i-1}, i^{l_i}) \in T_i$, $l_i \geq 1$, and assume (without loss of generality) that each q_i is such that there are infinitely many m with $V(r(q_i, i^m)) = V(q_i)$. (That is, let each process $P_2, \ldots, P_N$ in turn enter its trying region to a point where it could, on its own, cause the current value of V to recur infinitely many times. This is possible since V is finite and C1 holds.)

Since $|V| \leq k$, there exist i, j with $N - k \leq i < j \leq N$ and $V(q_i) = V(q_j)$. The processes $P_1, \ldots, P_i$, starting at q_i, comprise a system of at least $N - k \geq (k^2 - k)/2 - 1$ processes satisfying C1 and C3, so by the inductive hypothesis, $|V| \geq k$. Hence $|V| = k$. It follows that for every $v \in V$ and every q' reachable from q_i using only processes $P_1, \ldots, P_i$, there is a q'' reachable from q' using only processes $P_1, \ldots, P_i$ with $V(q'') = v$. (If not, then $P_1, \ldots, P_i$ starting from q' would be a system of processes satisfying C1 and C3 and using only values in $V - \{v\}$, contradicting the induction hypothesis.) In other words, $P_1, \ldots, P_i$ can be run in an admissible fashion, starting from q_i, so that V assumes every possible value infinitely often. Since $V(q_i) = V(q_j)$, the same is true starting from q_j.

We now construct a schedule admissible from q_j which locks out $P_{i+1}, \ldots, P_j$. We do this by running $P_1, \ldots, P_i$ to set V periodically to each $V(q_m)$, $i+1 \leq m \leq j$. Each time the value is set to some $V(q_m)$, P_m is run enough steps to return the value to $V(q_m)$. (Recall that by the choice of l_m this can be done infinitely often.) Repeating this process forever yields an infinite schedule admissible from q_j in which none of $P_{i+1}, \ldots, P_j$ ever leaves its trying region. This violates C3, a contradiction. We conclude that $|V| \geq k+1$. □

In the preceding proof, processes $P_1, \ldots, P_N$ were made to enter the system in a fixed order, and a counting argument was used to show repetition of values of V. We can do much better if we allow ourselves the freedom to select the order in which the processes initially enter. In order to obtain this improvement, we seem to be forced to introduce the technical assumption of "no memory" (C5), a property which is

possessed by all mutual exclusion algorithms we know of except for the 3-value 2-process algorithm of [4]; C5 allows guaranteed reproducibility of process entrance behavior.

Since the proof is more complicated than the others in this paper, it is helpful to decompose it by defining a digraph with vertices representing certain values of V, and with edges labeled by processes which cause the indicated changes in V when they make certain transitions upon entering the system. A purely graph-theoretic lemma can be used to show the existence of certain types of loops in such a labeled digraph, provided that the number of vertices is small. When this lemma is applied to the labeled digraph representing values and transitions, the resulting loops (representing sequences of transitions which begin and end with the same value of V) can be used to construct admissible lockout sequences.

We first present the needed graph-theoretic definitions and lemma. If L is a finite set, an *L-graph* is an edge-labeled finite digraph with labels in L. *vert*(G) denotes the set of vertices of G. If e is an edge, *label*(e) denotes the label of e, and *orig*(e) and *term*(e) denote the vertices at which e originates and terminates, respectively. (We permit multiple edges with the same originating and terminating vertices, provided they are distinctly labeled.) An L-graph is *full* provided for each $x \in \text{vert}(G)$ and each $l \in L$, there is at least one edge e with $\text{orig}(e) = x$ and $\text{label}(e) = l$.

A *path* in an L-graph G is a sequence $\pi = (x_0, e_1, x_1, \ldots, e_k, x_k)$ where $k \geq 0$, the x_i are vertices, and the e_i edges of G, with $\text{orig}(e_i) = x_{i-1}$ and $\text{term}(e_i) = x_i$ for all i, $1 \leq i \leq k$. If $k = 0$, π is *null*. *vert*(π) denotes $\{x_i : 0 \leq i \leq k\}$, *labels*($\pi$) denotes $\{\text{label}(e_i) : 1 \leq i \leq k\}$, *orig*($\pi$) $= x_0$, and *term*(π) $= x_k$. π is a *loop* if $x_0 = x_k$. π is a *highway* provided no two of its edges have the same label. If $\pi_1 = (x_0, e_1, x_1, \ldots, e_k, x_k)$ and $\pi_2 = (x_k, e_{k+1}, x_{k+1}, \ldots, e_l, x_l)$ are paths, then $\pi_1 \cdot \pi_2$ denotes the path $(x_0, e_1, x_1, \ldots, e_k, x_k, e_{k+1}, x_{k+1}, \ldots, e_l, x_l)$.

If $x \in \text{vert}(G)$ and π_1, π_2 are highways, then the pair (π_1, π_2) is a *loop trail from x* provided (a)–(c) hold:

(a) π_2 is a nonnull loop.
(b) $\text{orig}(\pi_1) = x$.
(c) $\text{vert}(\pi_2) \subseteq \text{vert}(\pi_1)$.

An L-graph G is *loopy* provided L can be partitioned into two sets, L_1 and L_2, and for every $x \in \text{vert}(G)$, there are two highways $\pi_1(x)$ and $\pi_2(x)$ such that (d) and (e) hold:

(d) $(\pi_1(x), \pi_2(x))$ is a loop trail from x.
(e) $\text{labels}(\pi_1(x)) \subseteq L_1$ and $\text{labels}(\pi_2(x)) \subseteq L_2$.

LEMMA 4.11. *Let G be a full L-graph, $|L| \geq 0$. Assume $|\text{vert}(G)| \leq |L|/2$. Then G is loopy.*

PROOF. See the appendix. □

Now we obtain the lower bound.

THEOREM 4.12. *Let S be a system of N processes, $N \geq 1$, and q any i.d. Assume that S, q satisfy mutual exclusion (C1), are lockout-free (C3), and have no memory (C5). Then $|V| \geq |\{V(q') : q' \in C_1 \text{ is reachable from } q\}| \geq N/2$.*

PROOF. Assume the contrary, so $A = \{V(q') : q' \in C_1 \text{ is reachable from } q\}$ satisfies $|A| \leq (N-1)/2$. Define an L-graph G, $L = \{2, \ldots, N\}$, with $\text{vert}(G) = A$, in order

to apply Lemma 4.11. The loops thereby obtained will be used to help construct a schedule locking out some processes.

Let r_i denote the (unique) remainder state of process i, $1 \le i \le N$. There will be two types of edges in G, called *normal edges* and *dummy edges.*

Normal Edges. For $i \in L$, $v, w \in A$, an edge e with $\text{orig}(e) = v$, $\text{term}(e) = w$ and $\text{label}(e) = i$ is included as a *normal edge* if and only if $\delta_i^k(v, r_i) \in w \times X_i$ for infinitely many values of k. (That is, if process i enters the system and sees value v, it can, on its own, cause value w to recur infinitely often. Note that a given v, i may give rise to more than one normal edge.) In this case, let $(\textit{reset}(e, j))_{j=1}^{\infty}$ denote a sequence of numbers, each ≥ 1, such that $\delta_i^{\sum_{j=1}^{l} \text{reset}(e,j)}(v, r_i) \in w \times X_i$ for all $l \ge 1$. Also, if $\alpha = (x_0, e_1, x_1, \ldots, e_m, x_m)$ is any highway in G all of whose edges are normal, then define $\textit{sched}(\alpha) = (\text{label}(e_1))^{\text{reset}(e_1,1)} \cdots (\text{label}(e_m))^{\text{reset}(e_m,1)}$. (That is, sched($\alpha$) is a schedule which causes the variable changes described by highway α.)

Dummy Edges. For $i \in L$, $v \in A$ having *no* normal edges e with $\text{orig}(e) = v$ and $\text{label}(e) = i$, an edge e with $\text{orig}(e) = \text{term}(e) = v$ and $\text{label}(e) = i$ is included as a *dummy edge.*

(Note that there might be no normal edge with label i from vertex v, because v might never occur when process i is in its remainder state r_i. Then the application of δ_i to v and r_i might not represent an event that could occur during the course of an actual computation from q. Thus all values w which are reached infinitely often by such application might fail to be in A.)

Clearly G is full, so by Lemma 4.11, G is loopy. Let $\{(\pi_1(x), \pi_2(x)) : x \in A\}$ be a set of loop trails from the vertices of G and $L_1 \cup L_2$ a partition of L with $\text{labels}(\pi_1(x)) \subseteq L_1$ and $\text{labels}(\pi_2(x)) \subseteq L_2$ for all x in A.

CLAIM 1. *If α is a highway all of whose edges are normal, and if $q' \in \bigcap_{i \in \textit{labels}(\alpha)} R_i$ is an i.d. reachable from q with $V(q') = \textit{orig}(\alpha)$, then $V(r(q', \textit{sched}(\alpha))) = \textit{term}(\alpha)$.*

PROOF. Straightforward. □

CLAIM 2. *If $i \in L$, $q' \in C_1 \cap R_i$ is reachable from q, and $V(q') = v$, then there is a normal edge e with $\textit{orig}(e) = v$ and $\textit{label}(e) = i$.*

PROOF. By the finiteness of A. □

Let $B \subseteq A$ denote $\{V(q') : q' \in C_1 \cap \bigcap_{i=2}^{N} R_i \text{ is reachable from } q\}$.

CLAIM 3. *If $v \in B$ and α is any highway with $\textit{orig}(\alpha) = v$, then α contains no dummy edges.*

PROOF. Let $q' \in C_1 \cap \bigcap_{i=2}^{N} R_i$ be reachable from q and such that $V(q') = v$. Let $\alpha = (v = x_0, e_1, x_1, \ldots, e_m, x_m)$, $m \ge 0$, and assume $e_1, \ldots, e_{i-1}$ are normal edges. We show e_i is a normal edge. Consider $\beta = (x_0, e_1, x_1, \ldots, e_{i-1}, x_{i-1})$. If $q'' = r(q', \text{sched}(\beta))$, then clearly q'' is reachable from q and $q'' \in C_1 \cap R_{\text{label}(e_i)}$. Moreover, $V(q'') = x_{i-1}$, by Claim 1. By Claim 2, there is a normal edge e with $\text{orig}(e) = x_{i-1}$ and $\text{label}(e) = \text{label}(e_i)$. Thus there is no dummy edge e' with $\text{orig}(e') = x_{i-1}$ and $\text{label}(e') = \text{label}(e_i)$. It follows that e_i is a normal edge.

CLAIM 4. *For $v \in B$, it is the case that neither $\pi_1(v)$ nor $\pi_2(v)$ contains a dummy edge.*

PROOF. Let e be any edge of $\pi_1(v)$ or $\pi_2(v)$. In either case it is easy to see that there is a highway α with $\text{orig}(\alpha) = v$, containing e. Claim 3 then suffices. □

Let $q_0 = r(q, \text{exit}(q))$. Starting from q_0, we define a schedule h in $(L_1 \cup \{1\})^\infty$. This schedule will later have steps of L_2 processes inserted, thereby yielding a new schedule which locks out the new L_2 processes.

The definition of h involves simultaneous definition of three sequences of i.d.'s, $(q_i)_{i=0}^\infty$, $(s_i)_{i=0}^\infty$, and $(t_i)_{i=0}^\infty$. Each q_i is in $\bigcap_{j=1}^N R_j$ and each s_i in $C_1 \cap \bigcap_{j=2}^N R_j$, $i \geq 0$. q_0 has already been defined. For each $i \geq 0$, let $s_i = r(q_i, \text{enter}(q_i, 1))$. (That is, let P_1 enter and go to its critical region.) Then let $t_i = r(s_i, \text{sched}(\pi_1(V(s_i))))$. (That is, consider the π_1 highway corresponding to $V(s_i)$ and allow processes to enter the system in order to make the indicated changes. Claim 4 shows that this schedule is defined.) Finally, let $q_{i+1} = r(t_i, \text{exit}(t_i))$. (That is, let all processes return to their remainder regions.) Schedule h is then defined to be

$$\text{enter}(q_0, 1)\text{sched}(\pi_1(V(s_0)))\text{exit}(t_0)\text{enter}(q_1, 1)$$
$$\cdots \text{enter}(q_i, 1)\text{sched}(\pi_1(V(s_i)))\text{exit}(t_i) \cdots .$$

Since $|A|$ is finite, there is some fixed $v \in V$ such that $v = V(s_i)$ for infinitely many i. (That is, the same value of V will be reproduced at infinitely many of the steps when P_1 enters its critical region.) Let $(s_{i_j})_{j=0}^\infty$ be a subsequence of $(s_i)_{i=0}^\infty$ for which $v = V(s_{i_j})$ for all j. Write $h = h_{01}h_{10}h_{11}h_{20}h_{21}h_{30}h_{31} \cdots$, where the substrings are defined as follows:

(a) $h_{01} = \text{enter}(q_0, 1)\text{sched}(\pi_1(V(s_0)))\text{exit}(t_0)\text{enter}(q_1, 1) \cdots \text{enter}(q_{i_0}, 1)$.
(b) For each $j \geq 1$, $h_{j0} = \text{sched}(\pi_1(v))$.
(c) For each $j \geq 1$, $h_{j1} = \text{exit}(t_{i_{j-1}})\text{enter}(q_{i_{j-1}}, 1) \cdots \text{enter}(q_{i_j}, 1)$.

Now we modify h to obtain a new schedule h' exhibiting lockout. Let $h' = h'_{01}h'_{10}h'_{11}h'_{20}h'_{21}h'_{30}h'_{31} \cdots$, where the substrings are defined as follows:

(a) For each $j \geq 0$, $h'_{j1} = h_{j1}$.
(b) $h'_{10} = \text{sched}(x_0, e_1, x_1, \ldots, e_k, x_k)\text{sched}(\pi_2(v))\text{sched}(x_k, e_{k+1}, x_{k+1}, \ldots, e_m, x_m)$, where $\pi_1(v) = (x_0, e_1, x_1, \ldots, e_m, x_m)$ and $\text{orig}(\pi_2(v)) = x_k$.
(c) Assume $\pi_1(v) = (x_0, e_1, x_1, \ldots, e_m, x_m)$ and $\pi_2(v) = (x'_0, e'_1, x'_1, \ldots, e'_{m'}, x'_{m'})$. Since $\text{vert}(\pi_2(v)) \subseteq \text{vert}(\pi_1(v))$, we can define a function $f: [m'] \to [m]$ such that $x'_k = x_{f(k)}$ for all $k \in [m']$. Now consider any $j \geq 2$, and recall that

$$h_{j0} = \text{sched}(\pi_1(v)) = (\text{label}(e_1))^{\text{reset}(e_1,1)} \cdots (\text{label}(e_m))^{\text{reset}(e_m,1)}.$$

Define

$$h'_{j0} = (\text{label}(e_1))^{\text{reset}(e_1,1)}a_1(\text{label}(e_2))^{\text{reset}(e_2,1)}a_2 \cdots (\text{label}(e_m))^{\text{reset}(e_m,1)}a_m,$$

where each a_k, $1 \leq k \leq m$, is defined as follows. Let

$$a_k = (\text{label}(e'_1))^{\exp_1}(\text{label}(e'_2))^{\exp_2} \cdots (\text{label}(e'_{m'}))^{\exp_{m'}},$$

where $\exp_l = \text{reset}(e'_l, j)$ if $f(l) = k$ and 0 otherwise. Recall that the existence of edge e'_l from x'_{l-1} to x'_l means that process $\text{label}(e'_l)$ can reset V to value x'_l an infinite number of times. For each l the appropriate number of steps of process $\text{label}(e'_l)$ is spliced into h_{j0} at the place corresponding to the occurrence of $V = x'_l$.

Now consider execution of schedule h' starting from i.d. q_0. Claim 1 and the fact that $\pi_2(v)$ is a loop can be used to show that the insertion of $\text{sched}(\pi_2(v))$ into h_{10} affects nothing except for the internal states of L_2 processes. For each point of subsequent insertion of L_2 process steps, Claim 1 is again used, this time to show that the value of V immediately preceding the insertion is one which that process can reset; the number of steps spliced in is some number known to reset that value. Thus

all insertions of L_2 process steps affect nothing except for the internal state of L_2 processes.

Now it is easy to see that schedule h' executed from q_0 locks out the processes in labels($\pi_2(v)$). This is because every step that is executed by those processes occurs while process P_1 is in its critical region. Remaining details are left to the reader. □

5. *Open Questions and Directions for Further Investigation*

The principal interesting technical question left open by the present paper is the order-of-magnitude growth of the space bound for lockout-free mutual exclusion. That is, can Theorem 4.10 be strengthened to yield a lower bound linear in N, can Theorem 4.12 be strengthened to remove the assumption of "no memory," or is the true situation somewhere in between?

It would be interesting to consider the questions treated here using variations on the given general test-and-set primitive. In particular, what bounds are obtainable for a model in which only reading or writing of a variable, but not a combination of the two operations, is indivisible? What bounds are obtainable for a model having several shared 2-valued variables instead of one shared multivalued variable, assuming that the only indivisible access is a test-and-set on a single variable?

Various ways can be developed for measuring the "time" required for execution by systems of processes. Intuitively, some of our space-efficient system designs seem to extract a cost in additional computation time. Such trade-offs should be formalized and quantified.

Synchronization problems other than simple mutual exclusion should also be studied in the same framework. Some additional work in several of these directions appears in [1, 2, 9, 13, 16–18].

Appendix. Proof of Lemma 4.11

We construct the needed highways in a series of stages. At the beginning of each stage there is a set $X \subseteq \text{vert}(G)$ of vertices which have been processed, and sets M, M_1, $M_2 \subseteq L$, $|M| \leq 2|X|$, $M_1 \cup M_2$ a partition of M, of labels which have been used. For each $x \in X$, highways $\pi_1(x)$ and $\pi_2(x)$ have been defined, with (a) and (b) holding:

(a) $(\pi_1(x), \pi_2(x))$ is a loop trail from x.
(b) $\text{labels}(\pi_1(x)) \subseteq M_1$ and $\text{labels}(\pi_2(x)) \subseteq M_2$.

During each stage, X will have at least one new element added, and definitions of $\pi_1(x)$ and $\pi_2(x)$ will be provided for all new elements of X. Several new elements will also be added to M. The given conditions are preserved by these changes. At the end of the final stage, $X = \text{vert}(G)$, thus satisfying the requirements of the lemma.

At the start of the first stage, $X = M = \emptyset$.

Stage of Construction. (Two auxiliary highways, ϕ_1 and ϕ_2, are constructed in a series of steps. Then ϕ_1 and ϕ_2 are used to help define the required highways.) Choose $x \in \text{vert}(G) - X$, and initialize $\phi_1 = \phi_2 = (x)$.

(1) *Path Extension.* See if there exist edges e_1 and e_2 satisfying (1a)–(1d):

(1a) $\text{orig}(e_1) = \text{orig}(e_2) = \text{term}(\phi_1)$.
(1b) $\text{term}(e_1) = \text{term}(e_2) \in \text{vert}(G) - X - \text{vert}(\phi_1)$.
(1c) $\text{label}(e_1) \neq \text{label}(e_2)$.
(1d) $\{\text{label}(e_1), \text{label}(e_2)\} \subseteq L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)$.

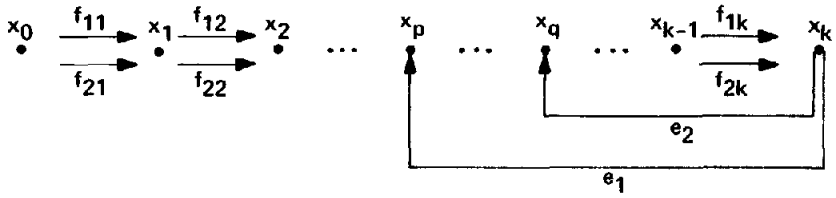

FIGURE 2

If not, then proceed to (2) below. If so, then let $\phi_i = \phi_i \cdot (\text{orig}(e_i), e_i, \text{term}(e_i))$, $i = 1, 2$, and return to (1) above. (That is, extend ϕ_1 and ϕ_2 as many times as possible to common previously unprocessed vertices, using pairs of edges with distinct previously unused labels.)

(2) *Loop Trail Construction.* See if there exist edges e_1 and e_2 satisfying (2a)–(2d):

(2a) $\text{orig}(e_1) = \text{orig}(e_2) = \text{term}(\phi_1)$.
(2b) $\{\text{term}(e_1), \text{term}(e_2)\} \subseteq \text{vert}(\phi_1)$.
(2c) $\text{label}(e_1) \neq \text{label}(e_2)$.
(2d) $\{\text{label}(e_1), \text{label}(e_2)\} \subseteq L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)$.

If not, then proceed to (3) below. If so, then assume without loss of generality that $\phi_1 = (x_0, f_{11}, x_1, \ldots, f_{1k}, x_k)$, $\phi_2 = (x_0, f_{21}, x_1, \ldots, f_{2k}, x_k)$, $\text{term}(e_1) = x_p$, $\text{term}(e_2) = x_q$, and $p \leq q$. Define $\pi_1(x_i)$ and $\pi_2(x_i)$ for all i, $0 \leq i \leq k$, as follows. For $0 \leq i \leq k$, let $\pi_2(x_i) = (x_q, f_{2(q+1)}, x_{q+1}, \ldots, f_{2k}, x_k, e_2, x_q)$. For $0 \leq i \leq p$, let $\pi_1(x_i) = (x_i, f_{1(i+1)}, x_{i+1}, \ldots, f_{1p}, x_p, f_{1(p+1)}, x_{p+1}, \ldots, f_{1k}, x_k, e_1, x_p)$. For $p + 1 \leq i \leq k$, let $\pi_1(x_i) = (x_i, f_{1(i+1)}, x_{i+1}, \ldots, f_{1k}, x_k, e_1, x_p, f_{1(p+1)}, x_{p+1}, \ldots, f_{1i}, x_i)$. For $0 \leq i \leq k$, add x_i to X, and add $\text{labels}(\pi_1(x_i))$ to M_1 and $\text{labels}(\pi_2(x_i))$ to M_2. The stage is complete. (Figure 2 should be helpful. Here ϕ_1 represents the upper and ϕ_2 the lower path. Two edges, e_1 and e_2, branch back to prior vertices of ϕ_1 (and ϕ_2), thereby creating two loops. For each i, $\pi_2(x_i)$ represents the "inner loop" (from x_q to x_k and back by e_2 to x_q), while $\pi_1(x_i)$ represents a path from x_i to a vertex of the "outer loop" (from x_p to x_k and back by e_1 to x_p) followed by a complete circuit of the outer loop.)

(3) *Loop Trail Access.* See if there exists an edge e satisfying (3a)–(3c):

(3a) $\text{orig}(e) = \text{term}(\phi_1)$.
(3b) $\text{term}(e) \in X$.
(3c) $\text{label}(e) \in L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)$.

If not, then the stage halts with an error. If so, then let $\phi_1 = (x_0, f_{11}, x_1, \ldots, f_{1k}, x_k)$. Define $\pi_1(x_i)$ and $\pi_2(x_i)$ for all i, $0 \leq i \leq k$, as follows. $\pi_1(x_i) = (x_i, f_{1(i+1)}, x_{i+1}, \ldots, f_{1k}, x_k, e, \text{term}(e)) \cdot (\pi_1(\text{term}(e)))$. $\pi_2(x_i) = \pi_2(\text{term}(e))$. For $0 \leq i \leq k$, add x_i to X and $\text{labels}(\pi_1(x_i))$ to M_1. The stage is complete. (Intuitively, previously constructed loop trails are being re-used, with the newly-defined path ϕ_1 used to provide initial access.)

End of Stage.

It is easily seen that the complete execution of a stage preserves the needed conditions. Stages continue to be initiated as long as $X \neq \text{vert}(G)$. Since $\text{vert}(G)$ is finite and at least one element is added to X at each stage, we eventually obtain $X = \text{vert}(G)$, provided no stage ends in an infinite loop at step (1) or terminates with an error. An infinite loop is impossible, because at each execution of step (1) an element is added to $\text{vert}(\phi_1)$; thus (1b) must eventually fail.

We argue that no stage A having $|X| < |\text{vert}(G)|$ at its start terminates with an error. We know that $|M| \leq 2|X|$ at the start of stage A, and also that $2|\text{vert}(G)| \leq$

$|L|$. Thus $|L - M| \geq 2|\text{vert}(G) - X|$ at the start of stage A. Immediately after initialization of ϕ_1 and ϕ_2 we have

$$|L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)| \geq 2|\text{vert}(G) - X - \text{vert}(\phi_1) + 1|,$$

and this inequality is preserved by any number of executions of step (1). Thus, after any number of executions of step (1) we have

$$|L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)| \geq |\text{vert}(G) - X - \text{vert}(\phi_1) + 1| + 1.$$

We now apply the Pigeonhole Principle. Intuitively, the term on the left-hand side of the last inequality represents the number of unused labels, while the term on the right-hand side is one more than the number of "pigeonholes," where one pigeonhole is allotted to each unprocessed vertex and a single pigeonhole is allotted to *all* the previously processed vertices. By the Pigeonhole Principle and the fact that G is full, there exist two edges e_1 and e_2 satisfying (c)–(f):

(c) $\text{orig}(e_1) = \text{orig}(e_2) = \text{term}(\phi_1)$.
(d) Either (d1) or (d2) holds;

(d1) $\text{term}(e_1) = \text{term}(e_2) \in \text{vert}(G) - X - \text{vert}(\phi_1)$.
(d2) $\{\text{term}(e_1), \text{term}(e_2)\} \subseteq X \cup \text{vert}(\phi_1)$.

(e) $\text{label}(e_1) \neq \text{label}(e_2)$.
(f) $\{\text{label}(e_1), \text{label}(e_2)\} \subseteq L - M - \text{labels}(\phi_1) - \text{labels}(\phi_2)$.

If (d1) holds, then step (1) is executed, while if (d2) holds, then either step (2) or step (3) will be executed. Thus no error will result. □

REFERENCES

1. BURNS, J Mutual exclusion with linear waiting using binary shared variables *SIGACT News 10*, 2 (Summer 1978), 42–47
2. BURNS, J. Complexity of communication among asynchronous parallel processes Ph D. Dissertation, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Ga , 1981.
3. CREMERS, A., AND HIBBARD, T An algebraic approach to concurrent programming control and related complexity problems Tech Rep , Computer Science Dep., UCLA, Los Angeles, Calif , 1975
4. CREMERS, A , AND HIBBARD, T Mutual exclusion of N processors using an $O(N)$-valued message variable In *Lecture Notes in Computer Science 62*, Springer-Verlag, 1978, pp. 165–176
5. DIJKSTRA, E.W. Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (Sept 1965), 569
6. DIJKSTRA, E. Cooperating sequential processes In *Programming Languages*, F Genuys, Ed , Academic Press, New York, 1968.
7. EISENBERG, M.A., AND MCGUIRE, M R Further comments on Dijkstra's concurrent programming control problem *Commun ACM 15*, 11 (Nov. 1972), 999
8. FELDMAN, J. Synchronizing distant cooperating processes TR 26, Univ. of Rochester, Rochester, N Y., Oct 1977
9. FISCHER, M , LYNCH, N , BURNS, J , AND BORODIN, A. Resource allocation with immunity to limited process failure Proc. 20th Ann IEEE Symp on Foundations of Computer Science, Puerto Rico, 1979, pp. 234–254
10. KNUTH, D E Additional comments on a problem in concurrent control *Commun ACM 9*, 5 (May 1966), 321–322
11. LAMPORT, L A new solution of Dijkstra's concurrent programming problem *Commun. ACM 17*, 8 (Aug. 1974), 453–455
12. LIPTON, R Limitations of synchronization primitives with conditional branching and global variables Proc. 6th Ann ACM Symp on Theory of Computing, Seattle, Wash., 1974, pp 230–241
13. LYNCH, N , AND FISCHER, M. On describing the behavior and implementation of distributed systems *Theor. Comput Sci 13* (1981), 17–43
14. LIPTON, R , SNYDER, L , AND ZALCSTEIN, Y A comparative study of models of parallel computation Proc. 15th Ann Symp on Switching and Automata Theory, New Orleans, La , 1974, pp 145–155

15 MILLER, R., AND YAP, C. Formal specification and analysis of loosely connected processes. Res. Rep. RC 6716, IBM Thomas J. Watson Research Lab., Yorktown Heights, N.Y., Sept. 1977.

16 PETERSON, G. Time-space trade-offs for asynchronous parallel models—Reducibilities and equivalences. Proc. 11th Ann. ACM Symp. on Theory of Computing, Atlanta, Ga., April 1979, pp. 224–230.

17 PETERSON, G. The complexity of parallel algorithms. Ph.D. Dissertation, Computer Science Dep., Univ. of Washington, Seattle, Wash., 1979.

18. PETERSON, G., AND FISCHER, M. Economical solutions for the critical section problem in a distributed system. Proc. 9th ACM Symp. on Theory of Computing, Boulder, Colo., 1977, pp. 91–97.

19 RIVEST, R., AND PRATT, V. The mutual exclusion problem for unreliable processes: Preliminary report. Proc. 17th Ann. Symp. on Foundation of Computer Science, Houston, Texas, 1976, p. 1–8.

RECEIVED MAY 1979, REVISED JUNE 1980, ACCEPTED SEPTEMBER 1980