SHARED DATA REQUIREMENTS FOR IMPLEMENTATION OF MUTUAL EXCLUSION USING A TEST-AND-SET PRIMITIVE†

James E. Burns[1], Michael J. Fischer[2], Paul Jackson[1], Nancy A. Lynch[1], and Gary L. Peterson[2]

[1]School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332

[2]Department of Computer Science, FR-35
University of Washington
Seattle, WA 98195

Abstract

We analyze the shared memory requirements for implementing mutual exclusion of N asynchronous parallel processes in a model where the only primitive communication mechanism is a generalized test-and-set operation. While two memory states suffice to implement mutual exclusion, we show that any solution which avoids possible lockout of processes requires at least $\sqrt{2N} - \frac{1}{2}$ states. A technical restriction on the model increases this requirement to (N+1)/2 states, while achieving bounded waiting ("fairness") further increases the requirement to N+1 states. These bounds are nearly optimal, for we exhibit algorithms for the last two cases using $\lfloor N/2 \rfloor + 9$ and N+3 states, respectively. All our lower bounds apply *a fortiori* to the space requirements for weaker primitives such as P and V using busy-waiting.

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 processes, one often designates certain sensitive sections of code in the various processes "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 insure 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 [Di1, EM, Kn], shared memory with test-and-set operations [CH1], message channels [F], P and V operations [Di2], etc. Given a set of primitive operations, the "critical section problem" is to find entry and exit protocols using those operations which insure 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 [CH1, CH2, Di1, EM, Kn, La, PF, RP, 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 [Li, LZS, CH1, MY, etc]. 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 [CH1, MY, RP].

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 contain) for three critical section problems. Deadlock-free mutual exclusion of N processes can be achieved with only two shared memory values. Lock out-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 $\sqrt{2N} - \frac{1}{2}$ shared 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+1)/2 states are required. (All of our upper bound algorithms are memoryless.)

† This material is based upon work supported by the National Science Foundation under Grants No. MCS77-02474, MCS77-15628, and MCS77-28305.

2. A FORMAL MODEL FOR EXCLUSION PROBLEMS

Our model is a hybrid of the models of [CH1] and [MY], 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 $P = (V, X, p)$ 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* p is a total function, $p: V\times X \to V\times X$ with the following properties:

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

The set V is referred to as the *message variable*. 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 p(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 [CH1, MY]. 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 modelled 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 exit from the critical region requires returning to the remainder region before attempting to reenter the critical region.

Notation: For $N \in \mathbb{N}$, $[N] = \{1,\ldots,N\}$.

For $N \in \mathbb{N}$, a *system of N processes* is a $2N+1$-tuple $S = (V,X_1,\ldots,X_N,p_1,\ldots p_N)$ where for each $i \in [N]$, $P_i = (V,X_i,p_i)$ 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 $i \in [N]$. We define $V(q) = v$. The functions p_i of the individual processes have natural extensions to the set of i.d.'s of S, defined by $p_i(v,x_1,\ldots x_N) = (v',x_1,\ldots x_{i-1},x',x_{i+1},\ldots,x_N)$, where $p_i(v,x_i) = (v',x')$. 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] will be 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

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

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, $\text{final}(i,q,h) = y$ if there exists an i.d. $q' = (v,y_1,\ldots,y_N)$, schedules h_1,h_2 with h_1 finite and $h = h_1h_2$ such that 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, we assume 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, 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 define carefully 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 subsection, 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 insures 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 a description of a system is presented which satisfies mutual exclusion and no deadlock, having two values for its message variable. It is also shown in section 4 that two values are required for any system satisfying deadlock-free mutual exclusion. Both the algorithm and the lower bound are proved directly 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 exists a $q' \notin R_i$ 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 $h = h_1 \ldots h_m$ if there are indices $0 \leq i_1 < j_1 < i_2 < \ldots < j_k \leq m$ such that $r(q,h_1h_2\ldots h_{i_\ell}) \in R_j$, and $r(q,h_1h_2\ldots h_{j_\ell}) \in C_j$, $1 \leq \ell \leq k$. 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 it is unnecessary for us to require the schedule h to be admissible, for 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 does not seem to rule out any known exclusion algorithm. 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 will present 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 will be 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) having two values for its message variable.

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

$X = R \cup T \cup C \cup E$ as above. Here, $R = \{R_0\}$, $T = \{T_0\}$, $C = \{C_0\}$, $E = \phi$, and $V = \{0,1\}$.

q, the initial i.d., is $(0, \underbrace{R_0, R_0, \ldots R_0}_{N})$.

Transitions are:

$p(0,R_0) = (1,C_0)$
$p(1,R_0) = (1,T_0)$
$p(0,T_0) = (1,C_0)$
$p(1,T_0) = (1,T_0)$
$p(0,C_0) = \text{arbitrary}$
$p(1,C_0) = (0,R_0)$.

Verification by induction is straightforward. Note that S_1 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 No Lockout Mutual Exclusion Algorithms

The latter two 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 to program instruction counters. Some state transformations are implicitly expressible by the usual flow of control of Algol programs; others (branching and alteration of values of internal variables) must be explicitly expressed.

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

<set> ::= <scalar$_1$> setto <scalar$_2$>
[: statement]

The intended semantics is to compare the <variable> to the <scalar$_1$> values, all of which must be distinct. If a match is found, the <variable> is set to the corresponding <scalar$_2$> value, the corresponding statement (which represents a state change) is executed and control passes to the point immediately following the test-and-set. If no match is found, the test-and-set is re-executed from the beginning (busy-waiting). Only the testing and setting actions, (up to the colon), are explicitly indivisible; the <statement> portion is local to the acting process.

There are other non-standard Algol features in the algorithm, 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 present first Algorithm A which is somewhat simpler but uses a few more states.

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 [CH]. 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 it 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 six 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.)

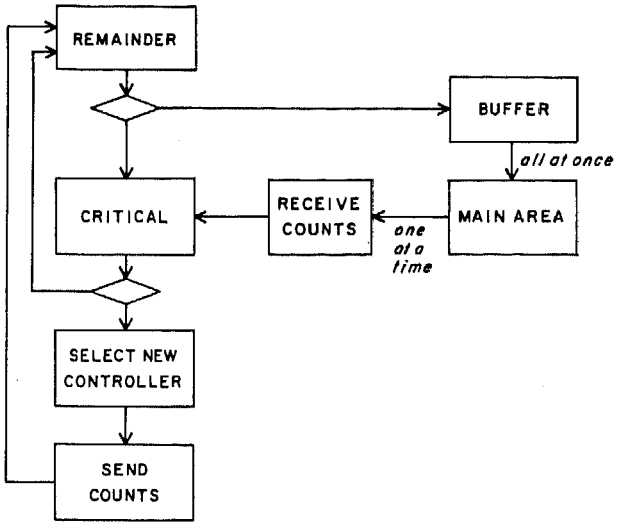

FIGURE 1

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 when 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.)

The only problem with the above scheme is that a process entering its trying region must 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 may hinder communication between the controller and the other processes. In [CH], about 2N values of V are used to allow 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 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 processes sees a message in V, it holds this value and sets V to S1, announcing its presence to the controller. It then waits until V takes on a value of S0, at which point it resets V to the held message. V will eventually "settle down" at S0 since the controller continues to reset V to this value, and the value can only be changed a finite number of times by entering processes. The messages are of course designed so that only those processes which should receive the message will act on it.

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;
  goto 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. It is easy to see that 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 labelled "RECEIVECOUNTS" and "SENDCOUNTS", as shown in Figure 3. Both counts are sent as a single coded integer by using the S-values. Note that the first waiting loop after "RECEIVECOUNTS" is required in Algorithm B to be certain that the ACK response to the ELECT message has been seen 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 to 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 following $\lfloor N/2 \rfloor + 9$ 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 occured, it has the responsibility to see that those processes which were not able to annouce their presence unambiguously will eventually get to their critical regions.

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 procedure is complete and then restores the held value. The executive then announces its presence to the controller in the normal way and enters the buffer. (If there is no controller, the executive goes directly to its critical region.)

Once at least k processes have been suspended by the executive, Algorithm C behaves identically to Algorithm A, since wraparound cannot occur again. 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. Since 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.

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;     ! j ≠ k !
     Sk    setto S0  : goto EXECUTIVE; ! k= ⌊N/2⌋!
     STOP  setto S1  : goto IDLE;
     other setto S1  : (M := other;
                        goto HOLDING)
   endtest;

EXECUTIVE:
  BUFF := k;
  while BUFF > 0 do
    test V until
      Sj    setto STOP: (BUFF := BUFF + j - 1;
                         IDLERS := IDLERS + 1)
      STOP  setto STOP:
      other setto S0  : M := other
    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 "! 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 state and sketch proofs for four lower bound theorems. More detailed versions of the proofs are deferred to a longer paper. In each case, proof is by contradiction; assuming there are too few values of V, we construct a schedule violating one of the needed conditions.

Corresponding to Theorem 3.1, we have the following.

Theorem 4.1. Let S be a system of N processes, $N \geq 2$, q any i.d. Assume S, q satisfy mutual exclusion (C1) and are deadlock-free (C2). Then $|V| \geq 2$.

Proof: Assume $|V| = 1$. Obtain $q' = r(q,h) \in \bigcap_{i=1}^{N} R_i$, (by (C2)). Again by (C2), obtain $k,\ell \geq 1$ with $r(q',1^k) \in C_1$ and $r(q',2^\ell) \in C_2$. Since $|V| = 1$, $r(q,h1^k2^\ell) \in C_1 \cap C_2$, contradicting (C1). □

To obtain a lower bound theorem corresponding to Theorem 3.2 we require a lemma giving a lower bound on the number of values needed for synchronization of 2 processes.

Lemma 4.2. Let S be a system of 2 processes, q any i.d. Assume 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 $q' \in R_1 \cap R_2$ reachable from q.

Proof: By a detailed case analysis following the ideas of [CH1]. □

Theorem 4.3. Let S be a system of N processes, $N \geq 2$, q any i.d. Assume 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), Lemma 4.2 gives the result for N=2. Assume N≥3. Construct $\{q_i\}_{i=0}^{N}$ a sequence of i.d.'s as follows.

Let $q_0 \in \bigcap_{i=1}^{N} R_i$ be reachable from q. Let $q_1 = r(q_0, 1^{\ell})$ $\in C_1$, $\ell \geq 1$. (That is, run P_1 alone until it enters its critical region.) For each i, $2 \leq i \leq N$, 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.) 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 q_j looks exactly 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.

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, 2 do not hold.

By Lemma 4.2, there is some schedule h involving 1 and 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 exactly 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 exactly like q_i to processes $P_{i+1}, \ldots, P_N$. Let $q'' = r(q', (i+1)(i+2)\ldots(N))$. (That is, allow each of $P_{i+1}, \ldots, P_N$ in turn to enter its trying region.) Then q" looks just 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 $q'' \notin R_1 \cap R_2$. □

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 4.4. Let S be a system of N processes and q any i.d. Assume 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 $\frac{k^2-k}{2} - 1$ 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, Lemma 4.2 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, q 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 $q_0 \in \bigcap_{i=1}^{N} R_i$ be reachable from q. Let $q_1 = r(q_0, 1^{\ell_1}) \in C_1$, $\ell_1 \geq 1$. (These are as for Theorem 4.3.) For each i, $2 \leq i \leq N$, let $q_i = r(q_{i-1}, i^{\ell_i}) \in T_i$, $\ell_i \geq 1$, 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 $N-k \geq \frac{k^2-k}{2} - 1$ 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 periodically set V to each $V(q_m)$, $i+1 \le m \le j$. 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 by the choice of ℓ_m that 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 $|V| \ge k+1$. □

Theorem 4.5. Let S be a system of N processes and q any i.d. Assume S, q satisfy mutual exclusion (C1), are lockout-free (C3) and have no memory (C5). Then $|V| \ge |\{V(q'): q' \in C_1$ is reachable from $q\}| \ge \frac{N+1}{2}$.

Proof. The complete proof is too lengthy for this paper, so we present a simplified version which assumes that all processes are identically programmed and that there is only one "free" value v_0, i.e. if $q' \in \bigcap_{i=1}^{N} R_i$ is reachable from q, then $V(q') = v_0$. (In this case, we also get a slightly better bound.)

Let $k = N/2 + 1$ and assume $|\{V(q'): q' \in C_1$ is reachable from $q\}| < k$. Construct $\{q_i\}_{i=0}^{N}$ as in the proof of Theorem 4.4, choosing each ℓ_i to be as small as possible. Since $q_i \in C_1$, $i \ge 1$, there exist i', j' with $1 \le i' < j' \le k$, and $V(q_{i'})=V(q_{j'})$. Since all the processes are identically programmed, $V(q_{i'+m}) = V(q_{j'+m})$, $0 \le m \le N-j'$. Let $i = j'-1$, $j = i+(j'-i') \le N$. We have $V(q_i) = V(q_j)$ and $\{V(q_0),\ldots,V(q_i)\} \supseteq \{V(q_{i+1}),\ldots,V(q_j)\}$. By condition (C5) and the assumption of only one free value, processes $P_1,\ldots,P_i$ can be run in an admissible fashion so as to drive V to each value in $V(q_0),\ldots,V(q_i)$ infinitely often. Thus, $P_{i+1},\ldots,P_j$ can be locked out exactly as in the proof of theorem 4.4, contradicting (C3). □

References

[CH1] Cremers, A. and T. Hibbard. "An Algebraic Approach to Concurrent Programming Control and Related Complexity Problems." University of Southern California Computer Science Department technical report, Nov., 1975.

[CH2] Cremers, A. and T. Hibbard. "Mutual Exclusion of N Processors using an O(N)-valued Message Variable." (extended abstract). University of Southern California, 1977.

[Di1] Dijkstra, E. "Solution of a Problem in Concurrent Programming Control." CACM 9, 9 (1965), p. 569.

[Di2] Dijkstra, E. "Cooperating Sequential Processes." In Programming Languages, F. Genuys, ed., Academic Press, New York, N.Y., 1968.

[EM] Eisenberg, M. and M. McGuire. "Further Comments on Dijkstra's Concurrent Programming Control Problem." CACM 15, 11 (1972), p. 999.

[F] Feldman, J. "A Programming Methodology for Distributed Computing (among other things)." Technical report TR9, Dept. of Computer Science, University of Rochester (1977), 51pp.

[Kn] Knuth, D. "Additional Comments on a Problem in Concurrent Control." CACM 9 (1966), p. 321.

[La] Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem." CACM 17, 8 (1974), p. 453.

[Li] Lipton, R. "Limitations of Synchronization Primitives with Conditional Branching and Global Variables." Proc. Sixth Annual Symposium on Theory of Computing, (1974).

[LZS] Lipton, R., Y. Zalcstein and L. Snyder. "A Comparative Study of Models of Parallel Computation." Proc. 15th Annual Symposium on Switching and Automata Theory, (1974).

[MY] Miller, R. and C. Yap. "Formal Specification and Analysis of Loosely Connected Processes." IBM Research Report RC 6716, 9/77.

[PF] Peterson, G. and M. Fischer. "Economical Solutions for the Critical Section Problem in a Distributed System." Proc. Ninth ACM Symposium on Theory of Computing, (1977), p. 91-97.

[RP] Rivest, R. and V. Pratt. "The Mutual Exclusion Problem for Unreliable Processes: Preliminary Report." Proc. 17th Annual Symposium on Foundation of Computer Science, (1976), p. 1-8.