INFORMATION AND COMPUTATION 107, 171–184 (1993)

Bounds on Shared Memory for Mutual Exclusion*

JAMES E. BURNS

Georgia Institute of Technology, Atlanta, Georgia 30332

AND

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

The shared memory requirements of Dijkstra's mutual exclusion problem are examined. It is shown that n binary shared variables are necessary and sufficient to solve the problem of mutual exclusion with guaranteed global progress for n processes using only atomic reads and writes of shared variables for communication.

1. INTRODUCTION

The first solution to the mutual exclusion problem was given by Dijkstra (1965). The original definition of the problem requires that n processes be synchronized so that no two processes are simultaneously executing portions of their code that are called "critical sections." Processes execute asynchronously; that is, they execute at independent, finite, nonzero rates, possibly varying over time. A process may halt if it is not executing its critical section or the part of its code that is devoted to synchronization. To prohibit trivial solutions, the system of processes is required to make global progress; that is, if any process is attempting to reach its critical section, then eventually some process must succeed. In Dijkstra's paper, and subsequent papers by Knuth (1966), de Bruijn (1967), and Eisenberg and McGuire (1972), communication between processes was required to be through shared variables. The only actions allowed on shared variables were reads and writes, which were assumed to be indivisible, and, usually,

* A preliminary version of this paper appeared in the 18th Annual Allerton Conference on Communication, Control, and Computing, October 1980. The research was supported in part by Army Research Office Contract DAAG29-79-C-0155, NSF Grants MCS77-15628, MCS78-01689, CCR-8611442, and CCR-8915206, DARPA Grants N00014-83-K-0125 and N00014-89-J-1988, and ONR Grant N00014-85-K-0168.

0890-5401/93 $5.00

any number of readers and writers were allowed to access any shared variable.

Subsequent work has examined the mutual exclusion problem using more powerful primitive actions for accessing shared variables. One of the primary concerns in the papers on mutual exclusion by Cremers and Hibbard (1979), Peterson (1979), Burns (1981b), Burns *et al.* (1982), and others has been the number of shared states required to solve certain variations of the mutual exclusion problem. (The number of shared states is the number of distinct values taken on jointly by the set of shared variables.) In this paper we examine the number of shared states required for Dijkstra's original problem, which assumes the communication mechanism is atomic reads and writes of shared variables.

The mutual exclusion problem has also been explored using communication primitives weaker than atomic reads and writes. Early work in the area by Lamport (1974) and Peterson (1983) explored mutual exclusion under the assumption that a variable can be shared only between the writer and a single reader. Recently, Lamport (1986c) has extended this work to include a much weaker assumption: that only single bit variables can be shared between a reader and a writer. The theoretical basis for this investigation can be found in Lamport (1986a, 1986b).

It is interesting to note that although the algorithms in several papers (Dijkstra, 1965; Knuth, 1966; de Bruijn, 1967; Eisenberg and McGuire, 1972) solve slightly different problems, they all use exactly the same number of shared states, $n \cdot 3^n$ (one n-valued variable and n three-valued variables). (Dijkstra (1965) uses n pairs of binary variables, but each pair takes on only three possible values.) We show that n binary variables (and hence 2^n shared states) are necessary and sufficient to solve the problem posed and solved by Dijkstra: mutual exclusion with global progress (often referred to as deadlock-free mutual exclusion in the literature). The papers by Knuth, de Bruijn, and Eisenberg and McGuire are concerned with starvation-free (also called lockout-free) mutual exclusion, which requires that every process that tries to enter its critical section eventually succeeds. Our lower bound applies to this variation of the problem, but our upper bound algorithm does not.

2. Read/Write Systems

In order to define precisely what we mean by asynchronous systems communicating by shared variables, we introduce a formal model suitable to our purpose. For brevity, we use the following notation throughout: $[n] = \{0, 1, ..., n-1\}$, for any positive integer n.

A *system* is a 4-tuple, $S=(P, V, q_0, \phi)$, where P is a finite set of $n>0$ processes, V is a finite set of $m>0$ variables, q_0 is the initial configuration of S, and ϕ is the transition function of S. We always let $P=\{p_0, p_1, ..., p_{n-1}\}$. The possibly infinite set X_i of states of p_i is partitioned into mutually disjoint sets R_i, T_i, C_i, and E_i, which are called the *remainder region*, *trying region*, *critical region*, and *exit region*, respectively. The set of values of the jth variable of V is V_j. A *configuration* of S is an $(n+m)$-tuple, $q=(x_0, x_1, ..., x_{n-1}, v_0, v_1, ..., v_{m-1})$, where $x_i \in X_i$ for each $i \in [n]$, and $v_j \in V_j$ for each $j \in [m]$; we use X_i, V_j, and V (ambiguously) as projection operators defined by $X_i(q)=x_i$, $V_j(q)=v_j$, and $V(q)=(v_0, ..., v_{m-1})$.

Let Q be the set of all configurations of S. Then ϕ is a total function, $\phi: [n] \times Q \to Q$, satisfying the following conditions. Let i be an integer in $[n]$ and q be a configuration of S. If $q'=\phi(i, q)$ then we write $q \xrightarrow{i} q'$ and require that

- $X_j(q')=X_j(q)$ for all $j \in [n]-\{i\}$.
- If $X_i(q) \in R_i \cup T_i$, then $X_i(q') \in T_i \cup C_i$.
- If $X_i(q) \in C_i \cup E_i$, then $X_i(q') \in E_i \cup R_i$.

The first condition implies that processes communicate only by the shared variables, since no process can directly affect the state of another. The latter two conditions enforce a cyclic order on the regions (remainder, trying, critical, exit) but allow direct transitions from remainder to critical and critical to remainder. We suppress any internal structure in the remainder and critical regions in order to focus on the synchronization protocols. If we allowed cycling in, say, the remainder region, a process could be designed to wait for fortuitous circumstances to try for the critical region while not being counted as competing, contrary to intuition. Effectively, "internal" steps while in the remainder or critical region are ignored, but "important" steps, those when a process begins to try for the critical region or leaves it, are explicitly accounted for.

A system $S=(P, V, q_0, \phi)$ is a *read/write system* if for every $i \in [n]$, X_i can be partitioned into $2m$ disjoint sets $Read_i^k$ and $Write_i^k$ for $k \in [m]$ such that the following conditions hold for every configuration q of S. If $X_i(q) \in Read_i^k$ then we say P_i is *about to read* the kth variable at q and required that

- $V(\phi(i, q))=V(q)$ and
- for every configuration q' of S with $X_i(q')=X_i(q)$, if $V_k(q')=V_k(q)$ then $X_i(\phi(i, q'))=X_i(\phi(i, q))$.

We say that the transition $q \to \phi(i, q)$ is a *read* of the kth variable by p_i. The definition implies that a read does not change any shared variable and

that the new state of the reading process depends only on the value of the variable read. This corresponds to the normal interpretation of a process reading a variable.

If $X_i(q) \in \text{Write}_i^k$ then we say P_i is *about to write* the kth variable at q and require that

- $V_j(\phi(i, q)) = V_j(q)$ for all $j \in [m] - \{k\}$ and
- for every configuration q' of S with $X_i(q') = X_i(q)$, $V_k(\phi(i, q')) = V_k(\phi(i, q))$ and $X_i(\phi(i, q')) = X_i(\phi(i, q))$.

We say that the transition $q \to \phi(i, q)$ is a *write* of the kth variable by process i. Thus, in a write only a single variable is changed and the new value depends only on the state of the writing process. This implies that a write overwrites the old value by a new value. Again, our formal definition of a write corresponds naturally to the normal interpretation.

A *history* of S is a finite or infinite sequence of process indices (elements of $[n]$). An occurrence of index i in a history is referred to as a *step* of process i in the history. If q_1 is a configuration of S and $h = i_1 i_2 \cdots$ is a finite or infinite history of S, then $q_1 q_2 \cdots$ is the *computation* from q_1 by h, where $q_{j+1} = \phi(i_j, q_j)$ for $j = 1, 2, \ldots$. If h is finite, then *result*(q, h) is the final configuration in the computation from q by h. We say that configuration q' is *reachable from configuration* q if there exists a finite history h such that $q' = result(q, h)$. A configuration is simply *reachable* if it is reachable from q_0. A history, h, of S is *fair* from configuration q if for every finite prefix, h_1, of h and for every $i \in [n]$, $X_i(\text{result}(q, h_1)) \notin R_i$ implies that i occurs in the suffix of h following h_1. A process *halts* in a history if and only if it appears only a finite number of times. Thus, halting relates to a history rather than to the state of a process. If history h is fair from q, any process that halts in h (does not take an infinite number of steps) is in its remainder region after its last step in the computation from q by h. In other words, once a process begins trying for the critical region, we can depend on it to continue to interact with the other processes until it has reached its critical region (if ever) and returned to its remainder region.

3. Mutual Exclusion with Global Progress

A configuration q of a system $S = (P, V, q_0, \phi)$ *violates mutual exclusion* if there are distinct values, i, $j \in [n]$ such that $X_i(q) \in C_i$ and $X_j(q) \in C_j$ (i.e., two processes are simultaneously in their critical regions). System S *satisfies mutual exclusion* if no reachable configuration q of S violates mutual exclusion.

Process i is said to *change regions* from q by h if there exist finite prefixes h_1 and h_2 of h such that $X_i(result(q, h_1))$ is in a different region than

$X_i(result(q, h_2))$. A system S has the *global progress* property if for every reachable configuration, q, and every nonnull history, h, that is fair from q, some process changes regions from q by h. Note that a system with global progress can allow a process to starve (never get to the critical region even though trying forever). The upper bound result below allows starvation and thus does not solve starvation-free mutual exclusion.

4. The Upper Bound

The algorithm in Fig. 1 is in a Pascal-like notation. (A similar algorithm was discovered independently by Lamport (1986c). We include the algorithm for completeness and because Lamport's algorithm has some additional complications in order to achieve an additional property, self-stabilization.) Note that, for brevity, three predicates are expressed using the "for all" notation. It is understood that these would be translated into separate atomic read operations for each shared variable accessed. Correctness does not depend on the order in which these variables are read.

The algorithm begins execution with all shared variables set to *down* and with process i executing the program in Fig. 1 for each $i \in [n]$.

THEOREM 4.1. *For every integer $n > 0$ there exists a read/write-system of n processes and n binary shared variables that solves the problem of mutual exclusion with global progress.*

Proof. For any integer $n > 0$, it is clear (using the natural and obvious translation from the algorithm to the formal model) that the algorithm in

```
type flagtype = (down, up);
shared var Flag : array [0..n − 1] of flagtype initially [down, ..., down];

program Process_i;
  local var j : 0..n − 1;
begin
  while true do begin
    remainder;                                              (* Remainder region *)
    repeat                                                  (* Trying region Entry *)
      Flag[i] ← down;
      repeat until ∀j ∈ {0, ..., i − 1} Flag[j] = down;     (* Trying region Subentry *)
      Flag[i] ← up;
    until ∀j ∈ {0, ..., i − 1} Flag[j] = down;
    repeat until ∀j ∈ {i + 1, ..., n − 1} Flag[j] = down;   (* Trying region Gateway *)
    critical;                                               (* Critical region *)
    Flag[i] ← down;                                         (* Exit region *)
  end.
```

FIG. 1. The Mutual Exclusion Algorithm.

Fig. 1 defines a read/write system with n processes and n binary shared varaibles. We must show that the system has the global progress property and satisfies mutual exclusion.

Suppose the system lacks the global progress property. Then we can reach a configuration, q, at which at least one process is not in its remainder region and a history h, fair from q, such that no process changes region from q by h. Since the only looping in each process's program occurs in the trying region, we observe that for each $i \in [n]$, either $X_i(q) \in R_i$ and i does not occur in h, or $X_i(q) \in T_i$ and i occurs infinitely often in h. We call the processes that are not in their remainder regions at q "active."

For each $i \in [n]$, define the following subsets of T_i:

- $Entry_i$ = the sets of states of p_i corresponding to the (entire) first repeat loop.
- $Subentry_i$ = the set of states of p_i corresponding to the inner repeat loop of $Entry_i$.
- $Gateway_i$ = the sets of states of p_i corresponding to the the second (unnested) repeat loop.

We note that if p_i reaches $Gateway_i$, then it remains there for the rest of the computation and $Flag[i]$ is continuously equal to *up*. Let $k = \min\{i \in [n] : p_i \text{ is active at } q\}$. Since p_k eventually detects that all $Flag[i] = down$ for $i \in [k-1]$, p_k reaches $Gateway_k$ after a finite prefix of h. After an appropriate extension of this prefix, every active p_i will either be in $Gateway_i$ or will begin cycling forever in $Subentry_i$ with $Flag[i] = down$, since all processes that do not reach $Gateway_i$ will detect $Flag[k] = up$. Let q' be the configuration reached after this extended prefix of h, and let $l = \max\{i \in [n] : X_i(q') \in Gateway_i\}$ (which is defined since the Gateway is not empty). Now p_l will find $Flag[i] = down$ for all $i \in \{l+1, \ldots, n-1\}$, so p_l will change regions from q' by the remaining suffix of h, contradicting our supposition. Therefore, global progress is guaranteed.

Suppose that mutual exclusion can be violated. Then there are values i, $j \in [n]$ such that $i \neq j$ and a finite history h such that $q = result(q_0, h)$, $X_i(q) \in C_i$ and $X_j(q) \in C_j$. Although p_i might cycle in $Entry_i$ several times before reaching its critical region at q, there must be a configuration at which p_i sets $Flag[i] = up$ for the last time before going critical. Let q_a be this configuration for p_i, and q_b be a similar configuration for p_j in the computation $q_0 q_1 \cdots q_k$ from q_0 by h ($q = q_k$). We may assume without loss of generality that $a < b$. But then for every c, $a \leqslant c \leqslant k$, $Flag[i] = up$ at q_c. (See Fig. 2.) Since p_j must test $Flag[i]$ after q_b, say at q_d, it either repeats another cycle in $Entry_j$, or loops in $Gateway_j$ during $q_b q_{b+1} \cdots q_k$,

$$q_0 \to \cdots \to \underbrace{q_a \to \cdots \to q_b \to \cdots \to q_d \to \cdots \to q_k}_{Flag[i] = up}$$

FIG. 2. The execution showing why mutual exclusion cannot be violated.

contradicting our supposition. Therefore, the algorithm also satisfies mutual exclusion and the theorem is proved. ∎

5. THE LOWER BOUND

The main result of this section is the following theorem, which shows that *any* mutual exclusion algorithm using atomic reads and writes for communication must use at least one variable per process, regardless of how large the variables are. Note that our shared variables are available for reading and writing by all processes. If variables were restricted to allow only one specific writer for each variable the lower bound would be trivial, since it is easy to see that every process must write something before entering the critical region.

THEOREM 5.1. *If S is a read/write system with at least two processes and S solves mutual exclusion with global progress, then S must have at least as many variables as processes.*

Since this result applies to all possible algorithms, it really says something about limitations on communication in read/write systems. Our preliminary definitions and lemmas give us the rigorous tools we need for the main result. Throughout, we let $S = (P, V, q_0, \phi)$ be a read/write system. The "looks like" equivalence relation defined below formally relates configuration's that are indistinguishable by a given process.

DEFINITION 5.1. Let q and q' be configurations of S, and p_i be a process of S. If $V(q) = V(q')$ and $X_i(q) = X_i(q')$ then q *looks like* q' to p_i (or q and q' *look alike* to p_i).

The following lemma merely states that the behavior of a set of processes is identical beginning from either of two configurations that look alike to them.

LEMMA 5.2. *Let q and q' be configurations of S, P' be a subset of P, and h be a finite history of S consisting only of steps of processes in P'. If q looks like q' to every process in P', then $V(result(q, h)) = V(result(q', h))$ and for each $p_j \in P'$, $X_j(result(q, h)) = X_j(result(q', h))$.*

Proof. From the definitions by induction on the length of h. ∎

The next definitions have to do with the communication of information between processes and how such communication can be blocked.

DEFINITION 5.2. Let q_1 be a configuration of S, h be a history of S, $i \in [n]$, $v \in V$, and $q_1 q_2 \cdots$ be the computation from q_1 by h. Let $j > 0$ be an integer such that $q_j \to q_{j+1}$ is a write of v by p_i. If there exists a positive integer $k > j$ such that $q_k \to q_{k+1}$ is a write of v, and if for all l, $j < l < k$, $q_l \to q_{l+1}$ is not a read of v by any process other than p_i, then we say that the write of v by p_i at q_j is *obliterated* from q_1 by h.

An obliterated write communicates no information to the other processes in a particular execution because after obliteration no trace of the write is left. A more severe constraint on communication occurs if *all* the writes of a process are obliterated. This is captured by the following definition.

DEFINITION 5.3. Let q be a configuration of S, h be a history of S, and $i \in [n]$. We say p_i is *hidden from* q by h if there exist histories h_1 and h_2, h_1 finite, such that $h = h_1 h_2$, $X_i(result(q, h_1)) \in R_i$, and every write by p_i in the computation from $result(q, h_1)$ by h_2 is obliterated from $result(q, h_1)$ by h_2. When q is understood from context, we simply say "p_i is hidden by h."

The importance of the remainder region in the above definition is that a process can legitimately be halted in the remainder region by a fair history. Therefore, no other process can wait until a hidden process takes some action, because a hidden process looks just like one halted in the remainder region. Also, note that if p_i is in its remainder region at q, then p_i is hidden from q by *any* history that excludes i.

LEMMA 5.3. *Let S be a read/write system, q be a configuration of S, h be a finite history of S and P' be a subset of processes of P such that each process in P' is hidden from q by h. Let $q' = result(q, h)$. Then there is a reachable configuration q'' of S such that every $p_i \in P'$ is in its remainder region at q'', and q'' looks like q' to each process in $P - P'$.*

Proof. For every $p \in P'$, there is some longest prefix h_p of h such that p is in its remainder region at $result(q, h_p)$. Let k_p be the number of steps of p in the suffix of h after h_p (we refer to these steps as *hidden steps*). We prove the lemma by induction on k, defined by

$$k = \sum_{p \in P'} k_p.$$

The basis with $k = 0$ holds trivially.

Suppose $k > 0$. Let p_i be the process in P' that takes the last hidden step in the computation from q by h. Then h can be written as $h_1 i h_2$, where the i corresponds to the last step of p_i. Let h' be $h_1 h_2$. (See Fig. 3). We first show that the processes of P' are hidden by h'. There are two cases to consider.

If the last step by p_i is a read, then clearly h' still hides all the processes in P' from q, since no write after this read will be affected.

Suppose that the last step by p_i is a write. If some process in P' is not hidden by h', then there is a write w_j by some $p_j \in P'$ to some variable v such w_j is not obliterated in the computation from q by h'. Since w_j is obliterated from q by h, the last write p_i must be to the same variable, v. But since p_i is hidden by h, there must be some write that obliterates p_i's last write before any process other than p_i reads v. But since p_i has no more steps, this same write will obliterate w_j before any other process reads v, contradicting the assumption. Thus, all processes in P' are hidden from q by h'.

By the induction hypothesis, there is a reachable configuration q''' which looks like $q'' = result(q, h')$ to all processes in $P - P'$, and such that every process in P' is in its remainder region at q'''. Since the last step of p_i in h is either a read or an obliterated write, we also have that q' and q'', and hence q' and q''', look alike to all processes in $P - P'$. This establishes the lemma. ∎

If some variable is about to be written, any other process writing that variable is in danger of having its write obliterated. We say that the variable is "covered," as described formally in the next definition.

DEFINITION 5.4. Let S be a read/write system, q be a configuration of S, and $i \in [n]$. If p_i is about to write $v \in V$ at q, then we say v is covered at q by p_i.

Suppose a process only writes variables covered by other processes entering its critical region; then other processes will be unaware that the process is critical in an execution in which all the covered variables are

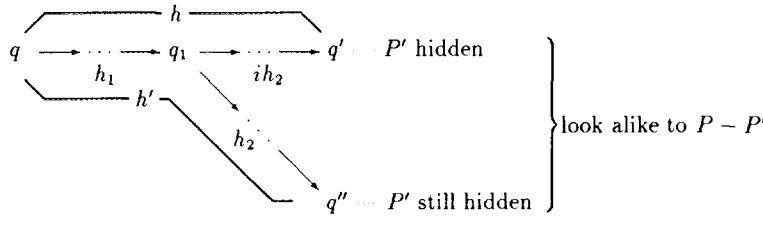

FIG. 3. ID's q and q' look alike to all processes other than p_i.

written one after the other, obliterating all the information that the critical process left behind. Indeed, the following lemma shows that a process must leave behind information in a variable that is not covered before going to its critical region.

LEMMA 5.4. *Let S be a read/write system with at least two processes that solves mutual exclusion with global progress, h be a finite history of S, $q = result(q_0, h)$, and $i \in [n]$. Suppose p_i is hidden from q_0 by h. If p_i goes to its critical region on its own from q by a history $h_1 = ii \cdots i$, then p_i must write some variable in the computation from q by h_1 that is not covered by any other process at q.*

Proof. By contradiction. First note that since p_i is hidden from q_0 by h, there is a point in the computation from q_0 by h at which p_i is in its remainder region and such that all of its following writes have been obliterated in reaching q. Suppose p_i reaches its critical region from q by history h_1 without writing any non-covered variable. Let h_2 be a history consisting of exactly one step of each process in $P - \{p_i\}$ that is about to write at q. Then every write of p_i from q is obliterated from q by $h_1 h_2$, so p_i is hidden from q_0 by hh_1h_2.

Now we use Lemma 5.3: there is a reachable configuration q'' that looks like $q' = result(q, h_1h_2)$ to all the other processes, but has p_i in its remainder region. Let h_3 be a non-null fair history from q'' that excludes steps of p_i. (History h_3 exists because there are at least two processes and p_i is in its remainder region at q''.) Since the system has the global progress property, some other process $p_j \neq p_i$ reaches its critical region in the computation from q'' by some finite prefix h_4 of h_3. But then h_4 applied to q' also makes p_j critical (by Lemma 5.2 since q' and q'' look alike to $P - \{p_i\}$). But then *result* (q', h_4) has both p_i and p_j critical, a violation of mutual exclusion. ∎

Combining the definitions of "covered" and "hidden" will give us our goal. We call a variable "nullified" by a process if the process does not "communicate" with the other processes and covers the variable.

DEFINITION 5.5. Let S be a read/write system, q be a configuration of S, h be a finite history of S, and v be a variable in V. We say that v is *nullified* from q by h if there is a process that is hidden from q by h and that covers v at $result(q, h)$.

For any reachable configuration q of S let $ret(q)$ be the history defined as follows: give each process that is not in its remainder region at q one step in turn, then repeat in round-robin fashion until one goes to the remainder region—this must happen by the global progress property. Repeat this procedure from the resulting configuration until all processes

are in their remainder regions. A configuration with all processes in their remainder regions is *quiescent*.

It should be clear that, for any reachable q, $ret(q)$ exists and that $result(q, ret(q))$ is quiescent. In particular, $result(q_0, ret(q_0))$ is quiescent (and might very well be q_0). Thus, Theorem 5.1 follows immediately from the following lemma, which implies there is a finite history from q_0 yielding n distinct nullified variables.

LEMMA 5.5. *Let S be a read/write-system with $n \geq 2$ processes that solves mutual exclusion with global progress. For every k, $1 \leq k \leq n$, and every reachable, quiescent configuration q_1 of S, there is a finite history h of S using only processes $p_0, p_1, \ldots, p_{k-1}$ such that k distinct variables are nullified from q_1 by h.*

Proof. Let q_1 be any reachable, quiescent configuration of S. We proceed by induction on k.

For the basis, let $k = 1$. By the global progress property, there must be a finite history h' consisting only of 0's such that p_0 goes critical at $result(q_1, h')$. Since q_1 is quiescent, p_0 is trivially hidden by the empty history from q_1, Lemma 5.4 applies and p_0 must write some variable in the computation from q_1 by h'. Let h be the shortest prefix of h' in which p_i does not write a variable, but is about to write some variable, namely w. Since it writes nothing, p_0 is hidden from q_1 by h and covers w at $result(q_1, h)$. Thus, $\{w\}$ is nullified from q_1 by h, and the lemma holds for $k = 1$.

We now show the inductive step ($k \Rightarrow k+1$). Assume the lemma holds for k, $0 < k < n$. Then there is a finite history h_1 from q_1 using only processes $p_0, p_1, \ldots, p_{k-1}$ such that a set of k variables is nullified from q_1 by h_0. Let W_2 be the set of these variables and $q_2 = result(q_1, h_1)$.

Note that $ret(q_2)$ begins with one step by each of $p_0, p_1, \ldots, p_{k-1}$ and then puts these processes back into their remainder regions, reaching a quiescent configuration. Applying the induction hypothesis again from $result(q_2, ret(q_2))$, we can get another history h_2' such that k distinct variables are nullified from $result(q_2, ret(q_2))$ by h_2'. Let W_3 be this set of variables, $h_2 = ret(q_2) \cdot h_2'$, and $q_3 = result(q_2, h_2)$.

We can repeat this construction *ad infinitum*, finding successive histories $h_3, h_4, \ldots$ and sets $W_4, W_5, \ldots$. (See Fig. 4.) Note that by the way each h_i is constructed, each of the processes in $p_0, p_1, \ldots, p_{k-1}$ is hidden from q_i by h_i for $i > 0$.

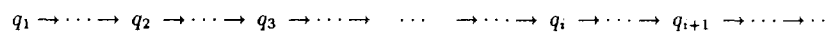

FIG. 4. The unbounded nullification sequence.

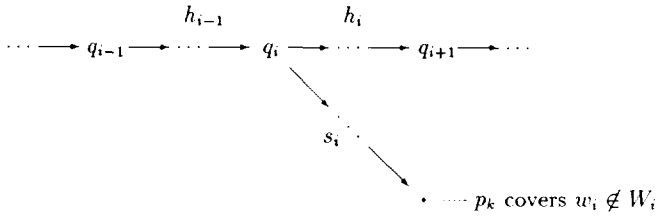

FIG. 5. The construction of side branches. Note that s_i only involves process p_k.

In order to extend the induction to $k+1$ processes, we need to bring p_k into the computation. To this end, we construct "side branches," histories s_i for each $i>1$ such that s_i proceeds from q_i and involves only p_k. By Lemma 5.3, for each $i>1$ there is some state q_i' reachable from q_{i-1} such that

- $p_0, ..., p_{k-1}$ are in their remainder regions, and
- q_i' looks like q_i to p_k.

From q_i', p_k must be able to reach its critical region on its own by some finite history $s_i' = kk \cdots k$, and hence also by s_i' from q_i. Note that p_k is trivially hidden from q_i by the empty history, since it is in its remainder region at q_i. By Lemma 5.4, p_k must write some variable w_i during the computation from q_i by s_i', where w_i is not covered by any process in $p_0, ..., p_{k-1}$. Thus, $w_i \notin W_i$. Let s_i be the shortest prefix of s_i' where (from q_i) p_k covers some variable $w_i \notin W_i$. (See Fig. 5.)

We are almost done, since at $result(q_i, s_i)$ each of $p_0, ..., p_k$ covers a distinct variable. We still need to find a computation in which the covering processes are hidden. We use a combinatorial trick. Choose i and j, $i<j$, such that $w_i = w_j$; we know that these i and j exist by a pigeonhole argument, since the chain of q_i's we constructed is infinite but there are only a finite number of variables. Create a history (see Fig. 6)

$$h = h_1 h_2 \cdots h_{i-1} s_i h_i h_{i+1} \cdots h_{j-1}.$$

Note that any writes by p_k in s_i are to the variables in W_i. Since h_i begins with a series of writes to each of the variables in W_i, all writes by p_k

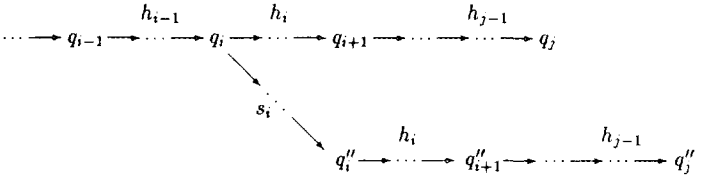

FIG. 6. The construction of the history h.

are obliterated from q_i by h_i, and p_k takes no steps thereafter. Now p_k is hidden by h, but we must also check that $p_0, ..., p_{k-1}$ are hidden.

Let $q''_{i+1} = result(q_1, h_1 h_2 \cdots h_{i-1} s_i h_i)$. Since any writes by p_k by p_k are obliterated, q''_{i+1} looks like q_{i+1} to $p_0, ..., p_{k-1}$, so these processes behave the same in the original computation reaching q_j and in the computation from q_1 by h. Therefore, $p_0, ..., p_{k-1}$ are hidden from q_1 by h and cover W_j at $q''_j = result(q_1, h)$.

Since $p_0, p_1, ..., p_k$ are hidden from q_1 by h, the variables they cover at q''_j are nullified. Now $w_i \notin W_j$, since $w_i = w_j \notin W_j$. Since in the computation from q_1 by h, $p_0, p_1, ..., p_{k-1}$ nullify k distinct variables in W_j and p_k nullifies $w_i \notin W_j$, the $k+1$ processes nullify $k+1$ different variables. ∎

6. Further Remarks

It is possible that the techniques used in the proof of Lemma 5.5 could be used to prove similar theorems for other exclusion problems. In order to apply the method, the following characteristics appear to be needed:

- A basis for the induction. It must be possible to show that some number of variables can be covered by a set of hidden processes.
- For the inductive step, a lemma similar to Lemma 5.4 must be proved. This requires that a subset of $n-1$ processes of a system of n processes forms a system of the type required.

Note that the second condition does not hold for the dining philosophers problem, for example.

Although the algorithm presented in this paper does use fewer shared states than previous solutions, it does not meet one requirement that was imposed by Dijkstra (1965). Dijkstra states that "The solution must be symmetrical between the n computers, as a result we are not allowed to introduce static priority." The algorithm here does not meet this condition because p_0 apparently has the highest priority (it will never be locked out). The issue of symmetry is explored by Burns (1981a) and Johnson and Schneider (1985). In Burns (1981a), one definition of symmetry proposed is so strong that mutual exclusion with global progress is not possible at all. Under a second, weaker definition, a solution exists, but it requires more space than the algorithm given here: one variable that can take on n values is required in addition to n binary shared variables. This is near-optimal since it is also shown that *some* variable must take on at least n values, and the lower bound of this paper implies that at least n shared variables are required in all. A new solution by Styer and Peterson (1989) shows that it is possible to solve the problem under the weaker definition of symmetry using n shared variables of size $O(n)$.

ACKNOWLEDGMENT

We thank Mike Fischer and the referees for helpful comment and criticism.

RECEIVED June 12, 1989; FINAL MANUSCRIPT RECEIVED August 6, 1991

REFERENCES

BURNS, J. E. (1978), Mutual exclusion with linear waiting using binary shared variables, *SIGACT NEWS* **10**, 42–47.

BURNS, J. E. (1981a), Symmetry in systems of asynchronous processes, *in* "Proceedings, 22nd IEEE Symposium on Foundations of Computer Science," pp. 169–174.

BURNS, J. E. (1981b), "Complexity of Communication among Asynchronous Parallel Processes," Ph.D. thesis, Georgia Institute of Technology.

BURNS, J. E., LYNCH, N. A., JACKSON, P., FISCHER, M. J., AND PETERSON, G. L. (1982), Data requirements for implementation of N-process mutual exclusion using a single shared variable, *J. Assoc. Comput. Mach.* **29**, 183–205.

DE BRUIJN, N. G. (1967), Additional comments on a problem in concurrent programming control, *Comm. ACM* **10**, 137.

CREMERS, A., AND HIBBARD, T. N. (1979), Mutual exclusion of N processors using an $O(N)$-valued message variable (extended abstract), *in* "Lecture Notes in Computer Science," Vol. 62, pp. 165–176, Springer-Verlag, Berlin/New York.

DIJKSTRA, E. W. (1965), Solution of a problem in concurrent programming control. *Comm. ACM* **8**, 569.

EISENBERG, M. A., AND R. MCGUIRE, M. (1972), Further comments on Dijkstra's concurrent programming control problem, *Comm. ACM* **15**, 137.

JOHNSON, R. E., AND SCHNEIDER, F. B. (1985), Symmetry and similarity in distributed systems, *in* "Proceedings, Fourth Annual ACM Symposium on Principles of Distributed Computing," pp. 13–22.

KNUTH, D. E. (1966), Additional comments on a problem in concurrent control, *Comm. ACM* **9**, 321–322.

LAMPORT, L. (1974), A new solution of Dijkstra's concurrent programming problem, *Comm. ACM* **17**, 453–455.

LAMPORT, L. (1986a), On interprocess communication. I. Basic formalism, *Distrib. Comput.* **1**, 77–85.

LAMPORT, L. (1986b), The mutual exclusion problem. I. A theory of interprocess communication, *J. Assoc. Comput. Mach.* **33**, 313–326.

LAMPORT, L. (1986c), The mutual exclusion problem. II. Statement and solutions. *J. Assoc. Comput. Mach.* **33**, 327–348.

PETERSON, G. L. (1979), "The Complexity of Parallel Algorithms," Ph.D. Thesis, Univ. of Washington.

PETERSON, G. L. (1983), A new solution to Lamport's concurrent programming problem using small shared variables, *ACM Trans. Programming Languages Systems* **5**, 56–65.

STYER, E. F., AND PETERSON, G. L. (1989), "Tight Bounds for Shared Memory Symmetric Mutual Exclusion Problems. Technical Report GIT-ICS-89/09, School of Information and Computer Science, Georgia Institute of Technology. (To appear *in* "Proceedings, Eight Annual ACM Symposium on Principles of Distributed Computing, August, 1989.")