JOURNAL OF COMPUTER AND SYSTEM SCIENCES 27, 350–377 (1983)

A Technique for Decomposing Algorithms Which Use a Single Shared Variable*

NANCY A. LYNCH†

Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

AND

MICHAEL J. FISCHER

Computer Science Department, University of Washington, Seattle, Washington 98195

Received December 7, 1981

A general theorem is proved which shows how a system of contending asynchronous processes with a special auxiliary supervisor process can be simulated by a system of contending processes without such a supervisor, with only a small increase in the shared space needed for communication. Two applications are presented, synchronization algorithms with different fairness properties requiring $N + c$ and $\lceil N/2 \rceil + c$ (c a constant) shared values to synchronize N processes, respectively.

I. INTRODUCTION

There are many algorithms in the literature for ensuring that the execution of systems of asynchronous processes exhibits various types of synchronization. In a typical formulation, asynchronous processes have "critical regions" of their code—portions of code whose execution is to be restricted so that certain patterns of simultaneous access to critical regions by different processes do not occur. Synchronization protocols are executed by the processes prior to entry to their critical regions, in order to prevent forbidden access patterns from occurring.

The simplest such restriction is that of Dijkstra's mutual exclusion problem [1], which specifies that no pair of processes should have simultaneous access to their critical regions; algorithms satisfying this restriction are useful for arbitrating requests for exclusive access to a single shared resource. Dijkstra's restriction can be generalized in a straightforward way, to specify that no more than $l \geqslant 1$ processes

* This research was supported in part by the National Science Foundation under Grants MCS77-02474, MCS77-15628, MCS78-01698, MCS80-03337, U.S. Army Research Office Contract DAAG29-79-C-0155, and Office of Naval Research Grants N00014-79-C-0873 and N00014-80-C-0221.

† Present address: Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. 02139.

0022-0000/83 $3.00

should have simultaneous access to their critical regions. Algorithms satisfying this more general "l-exclusion" restriction are useful for arbitrating requests for exclusive access to one of l identical copies of a shared resource.

The papers in [2–5] contain clever synchronization algorithms for mutual exclusion and l-exclusion, in an environment where a single shared variable serves as the only means of communication among the processes. The algorithms of these papers are designed so that only those processes which happen to be involved in conflicting demands for critical region access are available for participation in the synchronization protocols—there are no permanent, "dedicated" processes available to help achieve the synchronization. The algorithms of those papers are designed to minimize a certain "space" measure—the number of distinct values needed for the single shared variable. References [2–4] contain small-space algorithms for mutual exclusion with different fairness properties, while [5] contains corresponding small-space algorithms for l-exclusion.

Unfortunately, so many programming tricks seem to be required to achieve the very small space bounds that several of these algorithms (such as Algorithm C of [5] and the algorithm of [3]) are very difficult to understand from their programs. There are many different activities being carried out at once—processes perform many different functions at different times during their protocols, and sometimes share responsibility for performing certain functions. This responsibility is sometimes passed from one process to another using the communication variable. The communication variable is used for many different purposes, and it must be ensured that those different uses do not block each other indefinitely or become confused. The resulting programs are very intricately intertwined sequences of parallel actions, difficult to understand in their entirety, but also apparently difficult to decompose into meaningful subprograms. It seems that considerable benefit would be provided by isolating capabilities generally useful for presenting such algorithms, and then proving general theorems to show how those capabilities can be achieved using the given model. This approach has two advantages: the resulting decomposition should be easier to understand than the original algorithms, and also, portions of the decomposition should be useful for presenting several different algorithms.

In this paper, we isolate one general capability that seems very useful for presenting algorithms such as those in [2–5]. A significant reduction in the length and amount of complication in some of those algorithms results from assuming the existence of a dedicated "supervisor process," always available to aid in the synchronization of the other processes. If such a supervisor is available, a simplifying programming strategy is to push as much of the computation and decision-making as possible into the local computation of the supervisor, since properties of such local computation are well understood. Ideally, the non-supervisor processes would then have simple programs, their jobs reduced to carrying out communication with the supervisor to inform it of their critical region requirements and receive instructions for proceeding. The communication variable would be used only for the minimal communication necessary for this information exchange, and not for helping with computation that could be carried out locally.

In [2–4], the processes do, in effect, "simulate a supervisor" at various points in their code. However, the presentations in those papers tie the supervisor simulation responsibility to certain particular points in the process code, and do not attempt a clean separation between the basic algorithm and the supervisor function.

Of course, the basic model assumed by [2–5] assumes that there are no dedicated supervisor processes, so in order to use such a capability, we must show how to simulate a supervisor process using only the allowed participating processes. The main theorem of this paper is a general small-space supervisor simulation theorem. We then give two applications of this simulation theorem, based on the FIFO mutual exclusion algorithm of [3] and the lockout-free, mutual exclusion Algorithm C of [4]. Not only can versions of those two mutual exclusion algorithms be presented more simply using this decomposition strategy, but as a bonus, this strategy makes these two algorithms immediately extendable to small-space l-exclusion algorithms. We present the generalizations to l-exclusion. The two theorems provide space upper bounds for l-exclusion algorithms satisfying the two different fairness properties, bounds which are considerably better than those claimed in Theorem 4.1 of [5]. Peterson [8] has independently obtained versions of the two l-exclusion algorithms of this paper. Only his version of the "executive" algorithm appears in detail in [8]; the ideas of the FIFO algorithms are sketched. The bounds claimed in [8] are sharper than ours, ($N + l + 6$ and $\lfloor N/2 \rfloor + l + 8$, respectively), since a major effort of that paper is devoted to optimizing the constants. No decomposition of our type is used, however.

We note that [5] also presents some of its results using a general supervisor simulation theorem. However, the simulation of that paper is different from the present one. That simulation is designed to build in immunity to a certain type of process failure, a consideration which is not treated by the present simulation. Consequently, that simulation is not as space-efficient as the present one.

The algorithms presented in this paper are not trivial to understand, even with the given decomposition. There are still several types of communication going on using the same small-shared variable, and care must be taken to ensure that they do not interfere with each other. However, it seems that now the main strategy of each algorithm is reasonably easy to explain and that the main correctness arguments to be made involve the non-interference among the different communications being carried out in parallel.

There is a cost incurred by the decomposition, of an additive constant number of values. It seems possible to save a few (around 10) values by very careful optimization, involving both levels of decomposition. In the interest of simplicity, we have accepted this extra cost.

The remaining sections are organized as follows: Section II contains definitions and notation for processes, systems, and simulation, and for the correctness and fairness properties we wish to achieve. Theorem 1 is also stated, showing that our definition of simulation preserves all of our properties. Section III contains Theorem 2, the general simulation theorem, the restrictions required on the original supervisor system in order that it be simulable, a description of a high-level language

for presenting our algorithms, the program for the simulation and arguments for its correctness. Section IV contains the two applications: an $N + l + 15$-valued FIFO l-exclusion algorithm and an $\lceil N/2 \rceil + l + 18$-valued l-exclusion algorithm avoiding lockout, each for N processes. Each algorithm is presented by first providing a version with a supervisor and then appealing to Theorems 1 and 2. In each case, detailed code is provided, the algorithm is explained informally and arguments are given for the non-interference of communications.

II. Definitions and Notation

Processes and Systems

The definitions in this section are special cases of those in [6], describing an environment in which a finite number of deterministic processes access a single shared communication variable. Access is by a test-and-set operation which reads the value of the variable and changes it in a single indivisible step.

A *variable* x has an associated finite set of *values*, values(x), which the variable can assume. A *variable action* for x is a triple (u, x, v) with $u, v \in \text{values}(x)$; it represents the action of changing the value of x from u to v. act(x) is the set of all variable actions for x.

A *process* p has an associated finite set of *states*, states(p), which it can assume. start(p) is a distinguished *starting state*. A *process action* for p is a triple (s, p, t) with $s, t \in \text{states}(p)$; it represents p going from state s to state t. act(p) is the set of all process actions for p. If P is a set of processes, then $\text{act}(P) = \bigcup_{p \in P} \text{act}(p)$. variable($p$) is the single variable which p is permitted to access.

Every process action occurs in conjunction with a variable action; the pair forms a complete *execution step*. That is, if P is a set of processes and x a variable, we let $\text{steps}(P, x) = \text{act}(P) \times \text{act}(x)$ be the set of execution steps. Oksteps(p) is a subset of steps(p, variable(p)) describing the permissible steps of p. Oksteps is subject to the condition: for any $s \in \text{states}(p)$, $u \in \text{values}(\text{variable}(p))$, there exist exactly one t, v with $((s, p, t), (u, \text{variable}(p), v)) \in \text{oksteps}(p)$. Thus, processes are deterministic. We also use the alternative functional notation $\delta_p(s, u) = (t, v)$ to express the fact that $((s, p, t), (u, \text{variable}(p), v)) \in \text{oksteps}(p)$. If P is a set of processes, then $\text{oksteps}(P) = \bigcup_{p \in P} \text{oksteps}(p)$.

A *system of processes* S has three components: proc(S), a finite set of processes, var(S), a variable which is variable(p) for all $p \in \text{proc}(S)$, and $\text{init}(S) \in \text{values}(\text{var}(S))$, an initial value for var(S).

Let N denote the set of natural numbers, including 0. If A is any set, $A^*(A^\omega)$ denotes the set of finite (infinite) sequences of A-elements. A^{count} denotes $A^* \cup A^\omega$. Length: $A^{\text{count}} \to N \cup \{\infty\}$ denotes the number of elements in a given sequence. Let P be a set of processes, x a variable. $\mathscr{E}(P, x) = (\text{steps}(P, x))^{\text{count}}$ is the domain used to describe executions.

It is convenient in this paper to use one device not included in the model of [6]:

We sometimes consider several distinct processes operating on a common local state. This seems quite natural for some algorithms, where a single process performs more than one logical function. A more understandable code can result from separating those functions into distinct processes. In the following definitions, the set Q represents such a set of processes operating on a common state.

Let $e \in \mathscr{E}(P, x)$, $Q \subseteq P$ with $\text{start}(p) = \text{start}(p')$ for all $p, p' \in Q$. Define the latest-value function as follows:

If $\text{length}(e) = 0$, then $\text{latest}(Q, e) = \text{start}(p)$ for any $p \in Q$.

If $1 \leqslant \text{length}(e) < \infty$, and $e = e'((s, p, t), (u, x, v))$, then

$$\begin{aligned} \text{latest}(Q, e) &= t && \text{if } p \in Q, \\ &= \text{latest}(Q, e'), && \text{otherwise.} \end{aligned}$$

If $\text{length}(e) = \infty$, then $\text{latest}(Q, e) = \text{latest}(Q, e')$ provided e can be decomposed as $e'e''$ and no $p \in Q$ appears in e'', and is otherwise undefined.

If $Q = \{p\}$, a singleton (the usual case), we write $\text{latest}(p, e)$ instead of $\text{latest}(\{p\}, e)$.

Cooperative and Hierarchical Systems

A *worker* is a process p for which $\text{states}(p)$ is partitioned into subsets $R(p)$, $T(p)$, $C(p)$, and $E(p)$, (called the *remainder*, *trying*, *critical*, and *exit* regions of p, respectively), so that $\text{start}(p) \in R(p)$ and so that the following are true for any $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$:

(a) $s \in R(p)$ implies $t \in T(p) \cup C(p)$,

(b) $s \in T(p)$ implies $t \in T(p) \cup C(p)$,

(c) $s \in C(p)$ implies $t \in E(p) \cup R(p)$,

(d) $s \in E(p)$ implies $t \in E(p) \cup R(p)$.

That is, a process in its remainder region (resp. critical region), if it takes a step, will either go directly to its critical region (resp. remainder region) or will enter its trying region (resp. exit region). Once in the trying region (resp. exit region), a process will remain in that region until it progresses to its critical region (resp. remainder region). We are not counting local steps taken by the processes while in their remainder or critical regions, but are only considering steps involving synchronization.

A *cooperative system* is a system S of processes whose processes are all workers. A *hierarchical system* S is a system of processes in which all processes but two are workers, comprising $\text{workers}(S)$; the remaining two processes, called $\text{manager}(S)$ and $\text{clerk}(S)$, have the property that $\text{states}(\text{manager}(S)) = \text{states}(\text{clerk}(S))$ and $\text{start}(\text{manager}(S)) = \text{start}(\text{clerk}(S))$. Let $MC(S) = \{\text{manager}(S), \text{clerk}(S)\}$.

A cooperative system is the type of system assumed to be available in [2–5]. A hierarchical system is an augmentation of a cooperative system which allows two additional processes. These processes will be allowed to share a common local state.

The manager and clerk together comprise the "supervisor process" discussed in the

Introduction. For the algorithms we describe, the supervisor performs two distinct logical functions, and it is convenient to separate those two functions into those of two distinct processes, called a manager and a clerk.

Let S be a cooperative system. An *execution sequence* for S is a sequence $e \in (\text{oksteps}(\text{proc}(S)))^{\text{count}} \subseteq \mathscr{E}(\text{proc}(S), \text{var}(S))$ for which the following conditions hold (let $e = (e_i)_{i=1}^{\text{length}(e)}$, where $e_i = ((s_i, p_i, t_i), (u_i, \text{var}(S), v_i))$):

(a) If $p \in \text{proc}(S)$ and there is some i with $p_i = p$, then for the smallest such i it is the case that $s_i = \text{start}(p_i)$. If $i < j$, $p_i = p_j$, and there is no k, $i < k < j$ with $p_k = p_i$, then $t_i = s_j$.

(b) If $\text{length}(e) > 0$, then $u_1 = \text{init}(S)$. Also, $v_i = u_{i+1}$ for $1 \leqslant i < \text{length}(e)$.

Thus, the states of processes and values of the variable are consistent from step to step.

Let S be a hierarchical system. An *execution sequence* for S is a sequence $e \in (\text{oksteps}(\text{proc}(S)))^{\text{count}} \subseteq \mathscr{E}(\text{proc}(S), \text{var}(S))$ for which the following conditions hold (notation is as above):

(a) If $p \in \text{workers}(S)$ and there is some i with $p_i = p$, then for the smallest such e_i it is the case that $s_i = \text{start}(p_i)$. If $i < j$, $p_i = p_j$ and there is no k, $i < k < j$ with $p_k = p_i$, then $t_i = s_j$.

(b) Same as (b) in the preceding definition.

(c) If there is some i with $p_i \in MC(S)$, then for the smallest such i it is the case that $s_i = \text{start}(\text{manager}(S))$ $(= \text{start}(\text{clerk}(S)))$. If $i < j$, p_i, $p_j \in MC(S)$, and there is no k, $i < k < j$ with $p_k \in MC(S)$, then $t_i = s_j$.

Thus, states of worker processes and values of the variable are consistent from step to step. Also, manager(S) and clerk(S) "share a state": Values of the states of the pair of processes are consistent from step to step.

An execution sequence e for a cooperative system s is *admissible* provided $\text{latest}(p, e) \in R(p) \cup C(p)$ for every p for which $\text{latest}(p, e)$ is defined. An execution sequence e for a hierarchical system S is *worker-admissible* provided $\text{latest}(p, e) \in R(p) \cup C(p)$ for every $p \in \text{workers}(s)$ for which $\text{latest}(p, e)$ is defined. An execution sequence e for a hierarchical system S is *admissible* provided it is worker-admissible, and provided if $p \in MC(S)$ and if there are only finitely many i with $p_i = p$, then e is finite.

Thus, admissibility requires workers to continue taking steps while they are in their protocols (but not necessarily while they are in their own code.) They are also not required to leave their critical regions. (This is a weakening of the requirements imposed in [2–4].) The manager and clerk are both required to continue taking steps as long as any workers take steps. If all workers halt, then the manager and clerk are permitted to halt also.

Let $\bar{S}$ be a cooperative system, S a hierarchical system, and assume i: $\text{proc}(\bar{S}) \to \text{workers}(S)$ is an isomorphism. Then we say that $\bar{S}$ *simulates* S provided for every admissible execution sequence $\bar{e}$ of $\bar{S}$, there is an admissible execution sequence e of

S such that e exhibits the same set of region changes by the same (up to isomorphism i) worker processes in the same order, as $\bar{e}$.

Properties of Interest for Cooperative and Hierarchical Systems

(C1) *l-Exclusion*, $l \geqslant 1$.

A cooperative (resp. hierarchical) system S *violates l-exclusion* provided there exist a finite execution sequence e of S and distinct $p_1, \ldots, p_{l+1} \in \text{proc}(S)$ (resp. workers(S)) such that $\text{latest}(p, e) \in C(p)$ for all p_i, $i \in [l+1]$. S *satisfies l-exclusion* provided it does not violate l-exclusion.

(C2) *No l-deadlock*, $l \geqslant 1$.

A cooperative (resp. hierarchical) system S *exhibits l-deadlock* provided there is an admissible execution sequence e of S such that:

(a) all region changes eventually stop in e, and

(b) either (b1) or (b2) holds:

(b1) $\text{latest}(p, e) \in E(p)$ for some $p \in \text{proc}(S)$ (resp. workers(S));

(b2) $\text{latest}(p, e) \in T(p)$ for some $p \in \text{proc}(S)$ (resp. workers(S)) and at most $l-1$ distinct $p \in \text{proc}(S)$ (resp. workers(S)) have $\text{latest}(p, e) \in C(p)$.

S *satisfies* "*no l-deadlock*" provided it does not exhibit l-deadlock.

Thus, the system should continue to make progress as long as either some process is in its exit region, or some process is in its trying region with sufficient available space in the critical region. The system is permitted to stop making progress with processes still in their trying regions, in the case that the critical region remains filled. This formulation is stronger than that in [2, 4] and is similar to that in [5].

(C3) *No infinite bypass.*

A cooperative (resp. hierarchical) system S *exhibits infinite bypass* provided there exist an admissible execution sequence e and $p \in \text{proc}(S)$ (resp. workers(S)) such that

(a) $\text{latest}(p, e) \in T(p) \cup E(p)$,and

(b) infinitely many region changes occur in e.

S *satisfies* "*no infinite bypass*" provided it does not exhibit infinite bypass.

In the literature (including [2, 4]) a property called "no lockout" is usually formalized instead of (C3). "No lockout" is generally expressed in terms of each process making eventual progress. This requirement really includes two conditions: a condition which states that the system as a whole continues to make progress, and a condition which states that no process is indefinitely discriminated against in favor of other processes. Here, these two conditions are treated separately, as (C2) and (C3).

(C4) FIFO.

A cooperative (resp. hierarchical) system S *violates* FIFO *order* provided there exist $p, q \in \text{proc}(S)$ (resp. workers(S)) and finite execution sequence $e = e'e''$ of S such that (a) or (b) holds:

(a) Both (a1) and (a2) hold:

(a1) $\text{latest}(p, e') \in T(p)$ and p does not change regions in e'',

(a2) $\text{latest}(q, e') \in R(q)$ and $\text{latest}(q, e) \in C(q)$.

(b) Both (b1) and (b2) hold:

(b1) $\text{latest}(p, e') \in E(p)$ and p does not change regions in e'',

(b2) $\text{latest}(q, e') \in C(q)$ and $\text{latest}(q, e) \in R(p)$.

S *satisfies* FIFO provided it does not violate FIFO order.

Thus, FIFO order is preserved through both the trying region and the exit region.

Preservation of Properties by Simulation

THEOREM 1. *Let $\bar{S}$ be a cooperative system, S a hierarchical system, and assume $\bar{S}$ simulates S. Then if S satisfies any of properties* (C1)–(C4), *it follows that $\bar{S}$ satisfies the corresponding property (for the same value of l).*

Proof. All properties deal only with order of region changes, so the result is immediate from the definitions. ∎

III. The Simulation Theorem

General Strategy

In this section, we present the main simulation theorem. We wish to start with as general a hierarchical system as possible and simulate it using a cooperative system. Each process of the cooperative system simulates one worker process of the original hierarchical system. In addition, one process of the cooperative system at a time has the responsibility of simulating the manager and clerk. The first process to enter its trying or exit region first assumes the responsibility of simulating both the manager and clerk. It continues the simulation of the manager and clerk as long as it remains in its protocol. At the point when it is about to leave the protocol and go to its critical or remainder region, it passes the responsibility of simulating the manager and clerk to another process in its trying or exit region, by a communication protocol. That process behaves similarly. If at any time, a process simulating the manager and clerk is about to leave its trying or exit region but there is no remaining process to assume the responsibility, the leaving process simply puts the necessary manager–clerk state information in the variable and goes to its critical or remainder region. That state information is then available so that any new process that enters can resume the simulation of the manager and clerk.

Unlike the simulations implicit in the algorithms of [2–4], the present simulation does not require processes in their critical regions to participate in manager–clerk simulation. This is because our formulation permits processes to halt in their critical regions.

Restrictions on Hierarchical Systems

Certain restrictions are necessary for this simulation to be carried out.

DEFINITION. A hierarchical system S is called *r-regular* (r an integer) provided it satisfies the following six properties:

(1) If $p \in \text{workers}(S)$, $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$ and $t \in C(p)$ (resp. $R(p)$), then both (1a) and (1b) hold.

(1a) $s \in T(p)$ (resp. $E(p)$) and $u = v$, and

(1b) if $w \in \text{values}(x)$, then $((s, p, t), (w, x, w)) \in \text{oksteps}(p)$.

That is, the last operation of each worker's protocol is a "NO-OP". This requirement is necessary because a process about to enter its critical or remainder region can delay a long time while relinquishing its responsibility to simulate the manager and clerk, perhaps allowing other processes to change regions during the delay. It is not permissible for this delay to introduce orders of region changes not possible in the simulated hierarchical system. To prevent the introduction of new behavior, it is sufficient that the original system also have the possibility of a corresponding delay occurring with the same intervening region changes. Formally, this is made possible by requiring an extra, dummy step to occur. Since this step can occur at any time in the original hierarchical system, the new delay does not introduce any new orders of region changes.

Let $\text{prelim}(p)$ denote $\{s \in \text{states}(p)$: there exists some $((s, p, t), (u, x, v)) \in \text{oksteps}(p)$ with $t \in C(p) \cup R(p)\}$. Thus, $\text{prelim}(p)$ denotes states in $T(p) \cup E(p)$ immediately before a "NO-OP" is executed. An arbitrary subset of $\text{states}(\text{manager}(S)) = \text{states}(\text{clerk}(S))$ is called $\text{safe}(S)$. These states must satisfy several properties.

(2) Let e be any execution sequence of S which is worker-admissible, and in which both $\text{clerk}(S)$ and $\text{manager}(S)$ appear infinitely many times. Then for infinitely many distinct prefixes e' of e it is the case that $\text{latest}(MC(S), e') \in \text{safe}(S)$.

Thus, safe states can be made to occur by running the manager and clerk while the workers continue their normal operation.

(3) Let $e = e'e''$ be any finite execution sequence of S, and suppose $\text{latest}(MC(S), e') \in \text{safe}(S)$ and $\text{manager}(S)$ does not appear in e''. Then $\text{latest}(MC(S), e) \in \text{safe}(S)$.

That is, only the manager's own steps can cause its state to change from safe to unsafe. Thus, a safe state, once achieved, can be made to persist by stopping the manager.

(4) Let $e = e'e''$ be any infinite execution sequence of S with latest$(MC(S), e') \in$ safe(S). Assume e is worker-admissible, that manager(S) does not appear in e'' but clerk(S) appears infinitely many times in e''. Then after some finite initial subsequence of e, all region changes stop and var(S) always has the value 0 (i.e., all variable actions past that point are of the form $(0, \text{var}(S), 0)$).

Thus, stopping the manager at a safe state and allowing the clerk and workers to continue to run will eventually result in all activity ceasing and the shared variable becoming cleared for communication.

(5) If $((s, p, t), (u, x, v))$ and $((s', p, t'), (u, x, v')) \in$ oksteps(clerk(S)), then $v = v'$.

That is, the clerk always has the same effect on the variable, regardless of its own current state. This property makes it possible to simulate the effect of the clerk on the variable without knowing the clerk's state.

(6) $|\{s \in \text{states}(\text{manager}(S))$: $s \in \text{safe}(S)$, there is a finite execution sequence e of S with latest$(MC(S), e) = s$ and for no $p \in$ workers(S) is it the case that latest$(p, e) \in T(p) \cup E(p)\}| = r$.

That is, there are only r different safe manager states which could exist at times when there is no process available for assuming responsibility for the simulation. Let this set of safe states be denoted by free(S). If S is r-regular, then strengthened versions of properties (2), (4), and (6) can be proved easily:

DEFINITION. An execution sequence e of an r-regular hierarchical system S is *semi-worker-admissible* provided latest$(p, e) \in R(p) \cup C(p) \cup$ prelim(p) for every $p \in$ workers(S) for which latest(p, e) is defined.

That is, semi-admissibility requires workers to continue taking steps while they are in their protocols, *unless they are in preliminary states.* Since a preliminary state can be transformed by a "NO-OP" into a critical or remainder state, the rest of the system cannot distinguish between a process being in a preliminary state and in its critical or remainder region. Thus, an r-regular system S satisfies the following three properties:

(2′) Let e be any execution sequence of S which is semi-worker-admissible, and in which both clerk(S) and manager(S) appear infinitely many times. Then for infinitely many distinct prefixes e' of e it is the case that latest$(MC(S), e') \in$ safe(S).

(4′) Let $e = e'e''$ be any infinite execution sequence of S with latest$(MC(S), e') \in$ safe(S). Assume e is semi-worker-admissible, that manager(S) does not appear in e'' but clerk(S) appears infinitely many times in e''. Then after some finite initial subsequence of e, all region changes stop and var(S) always has the value 0 (i.e., all variable actions past that point are of the form $(0, \text{var}(S), 0)$).

(6′) $|\{s \in \text{states}(\text{manager}(S))$: $s \in \text{safe}(S)$, there is a finite execution sequence e

of S with latest$(MC(S), e) = s$ and for no $p \in$ workers(S) is it the case that latest$(p, e) \in (T(p) \cup E(p)) - \text{prelim}(p)\}| = r$.

A High-Level Language for Describing Processes

Algorithms will be described in an Algol-like language similar to the ones used in [3, 5, 7] but designed to make the translation into the basic model transparent. Added to the usual sequential programming constructs are two synchronization statements, *lock* and *unlock*. In addition, the construct "waitfor C" is used as an abbreviation for "while not C do [unlock; lock]."

Lock and unlock statements always occur in pairs, an "unlock" followed immediately (syntactically) by a "lock." *Location counter values* correspond to the points in the code immediately preceding each lock statement. States of the process p defined by a program corresponding to a particular location counter value together with values for all the program's local variables, with one such combination designated as the start state. Transitions are defined as follows: If the program is started with its location counter and local variable values described by state s, and u as the value of the shared variable x and if the program is then run according to usual sequential programming rules, it might or might not reach an unlock statement. If it does, if t is the state describing the resulting location counter and local variable values, and if v is the new value of the shared variable, then let $\delta_p(s, u) = (t, v)$. If it does not, then let $\delta_p(s, u) = (s, u)$. (In general, of course, this decision is not effective, but still gives a well-defined answer. In actual execution, the values leading to the second alternative should never occur.)

Thus, all computation, including local variable changes and control steps, is done while the shared variable is locked. The variations on the language used in [2, 5, 7] allow local computation to be performed while the variable is unlocked. In this paper, where we allow more than one process to access the same local state, we wish to leave no room for ambiguous translation.

Main Result

THEOREM 2. *Let S be an r-regular hierarchical system with* $|\text{values}(\text{var}(S))| = m$. *Then there is a cooperative system $\bar{S}$ which simulates S such that* $|\text{values}(\text{var}(\bar{S}))| = m + r + 4$.

Proof. The shared variable X of $\bar{S}$ has values$(X) =$ values$(\text{var}(S)) \cup$ {'NEW', 'SEND', 'ACK', 'DONE'} $\cup$ free(S). We assume that the three sets of values in this union are all disjoint. We call the values in values$(\text{var}(S))$ the *ordinary* values, denoted ORD, those in {'NEW'} $\cup$ free(S) the *selection* values, denoted SEL, and those in {'SEND', 'ACK', 'DONE'} the *communication* values, denoted COMM. init$(\bar{S}) =$ start(manager$(S)) \in$ free(S).

We describe a process $\bar{p}$ of $\bar{S}$. $\bar{p}$ has local variables as follows:

P	for the state of simulated process $p \in$ proc(S),
MGR	for the state of the manager (and clerk),

LIST	for recording a list of X-values during a restricted mode of simulation of the clerk,
C	for holding a code being transmitted or received,
FINISH	for indicating the end of receipt of a message,
M	for holding an interrupted communication value.

The initial value of P is start(p), and all other local variables are initialized at 0. The starting location counter value of $\bar{p}$ is at the *last* lock statement of the program.

The components of LIST will be denoted by $L_1, L_2, \ldots, L_{|\text{LIST}|}$, in the order in which they are placed in the list. We write mgr as an abbreviation for manager(S) and clk as an abbreviation for clerk(S). States are transmitted in unary; we assume *code* to be a function which assigns an integer to each manager state, and *decode* the corresponding decoding function. The subscripts 1 and 2 indicate the two components of a (state, variable-value) pair, respectively.

Process $\bar{p}$

```
        while true do
(1)       [while (P ∉ prelim(p) and X ∉ SEL) do
            [if X ∈ COMM then [M ← X; X ← 0];
             (P, X) ← δ_p(P, X);
             if X = 0 then [X ← M; M ← 0];
             unlock; lock];
(2)       if P ∉ prelim(p) then
(2a)        [if X ∈ free(S) then [MGR ← X; X ← 0]
             else /*X = 'NEW'*/
               [M ← 'ACK'; X ← 0;
                while FINISH = 0 do
                  [if X = 'SEND' then [M ← 'ACK'; C ← C + 1];
                   if X = 'DONE' then FINISH ← 1;
                   X ← 0; if P ∉ prelim(p) then (P, X) ← δ_p(P, X);
                   LIST ← LIST, X;
                   X ← (δ_clk(start(clk), X))_2;
                   if X = 0 then [X ← M; M ← 0];
                   unlock; lock]
                MGR ← δ_clk(··· δ_clk(δ_clk(decode(C), L_1), L_2),..., L_|LIST|);
                LIST ← 0; C ← 0; FINISH ← 0]
(2b)        while P ∉ prelim(p) do [(MGR, X) ← δ_clk(δ_mgr(MGR, X)); (P, X) ← δ_p(P, X); unlock; lock];
(2c)        while MGR ∉ safe(S) do [(MGR, X) ← δ_clk(δ_mgr(MGR, X)); unlock; lock];
(2d)        while X ≠ 0 do [(P, X) ← δ_clk(P, X); unlock; lock];
(2e)        if MGR ∈ free(S) then X ← MGR
            else [X ← 'NEW'; C ← code(MGR);
                  while C > 0 do [waitfor X = 'ACK'; X ← 'SEND'; C ← C − 1];
                  waitfor X = 'ACK'; X ← 'DONE']]
(3)       else if M ≠ 0 then [waitfor X = 0; X ← M; M ← 0];
          P ← (δ_p(P, 0))_1;
          unlock; lock]
```

ALGORITHM A

The regions of $\bar{p}$ are defined as follows: Any state s of $\bar{p}$ is in the same region as the state of p stored in the local variable P of state S.

We use the facts in the following two paragraphs to justify the faithfulness of the simulation.

The variable X is used both for simulating the actions of system S (when X has an ordinary value) and for coordination among processes simulating the manager and clerk (when X has a selection or communication value). Selection and communication values are only placed in X to replace the ordinary value of 0. Thus, if at any time a process sees a selection or communication value in X, and replaces it with 0 and simulates steps of system S, it will not cause any incorrect steps of S to be simulated.

Property (5) implies that it is possible for a process to simulate steps of the clerk without knowing the clerk's latest state. The correct effect on the variable X is obtained if the clerk step is simulated starting at an arbitrary clerk state. However, the proper effect on the clerk's state must also be achieved. Provided no other process simulates a manager or clerk step in the meantime, it is possible for a process unaware of the clerk's state to simulate the effect on X of several clerk steps, saving the X-values seen at each access. Then if the process later learns the state of the clerk prior to the enactment of these steps, it can apply δ_{clk} to this state and the sequence of saved X-values to bring the clerk state up-to-date.

We now describe the operation of the algorithm. Process $\bar{p}$ is initially uninvolved in the simulation of the clerk and manager. It executes ① as long as p is not ready to leave its protocol ($P \notin \text{prelim}(p)$) and $\bar{p}$ has not been asked to simulate the manager and clerk ($X \notin \text{SEL}$). While $\bar{p}$ is uninvolved, it will simulate steps of p, unlocking the variable between each step to allow other processes to take steps. While carrying out this simulation, $\bar{p}$ may occasionally see one of the communication values (indicating that responsibility for the manager and clerk is being passed from one process to another). If this occurs, $\bar{p}$ does not simply wait and try again. Instead, $\bar{p}$ interrupts the communication to simulate p's step, using 0 as the presumed value of X and holding on to the communication value. $\bar{p}$ continues at subsequent steps to simulate p's steps, but replaces the held communication value at the first opportunity.

It is quite possible that $\bar{p}$ is never asked to help simulate the manager and clerk. Then $\bar{p}$ might stay in ① forever (if p never reaches a preliminary state). Alternatively, $\bar{p}$ might leave ① when $P \in \text{prelim}(p)$. In this case, $\bar{p}$ simply waits (at ③) until it can replace any communication value it still holds, and can thereafter leave the protocol. Then p's final step is a NO-OP, by property (1), so it is permissible for $\bar{p}$ to assume that the value of X is 0 for the last step of the protocol.

In the more complicated case, $\bar{p}$ exits ① without having p ready to change regions ($P \notin \text{prelim}(p)$), upon being asked to simulate the manager and clerk($X \in \text{SEL}$). In this case, $\bar{p}$ must execute ②. ② consists of a protocol ⑫ to initialize the simulation of the manager and clerk, followed by ⑫ which continues the simulation of both the manager and clerk, along with p. When p is about to change regions ($P \in \text{prelim}(p)$) then $\bar{p}$ enters ⑫–⑫ which transfer the control of the simulation of the manager and clerk elsewhere. We examine ⑫–⑫ in greater detail.

(2b) is a straightforward loop carrying out a simulation of the manager, clerk and p. (2c) is a loop which is executed until the manager enters a safe state. (2d) is a loop which is executed until the value of X reaches 0; $\bar{p}$ attempts to achieve this effect by simulating the clerk but not the manager. (2e) is the actual transfer of manager–clerk state. In case the manager is in the restricted set free(S) of states, the state is simply left in X. Otherwise, an initial message 'NEW' is placed in X, followed by a communication protocol which sends the manager state, coded in unary.

Although the simulation of the manager is to be allowed to be stopped temporarily, we wish to continue the simulation of the clerk even during the transfer of the manager state. Thus, some process must have responsibility for the clerk's simulation during the transfer protocol. We choose to require the new intended recipient of the manager state to begin simulating the clerk immediately upon receipt of a selection value. Since this recipient does not have the clerk's state available at the time the selection value is received, it simulates the clerk in the "temporary mode" discussed above, making the required changes to the variable X but saving the values of X it sees during the temporary mode simulation for later updating of the clerk's state.

We now consider (2a) the protocol for assuming the responsibility for simulating the manager and the clerk. If a state in free(S) is encountered, then $\bar{p}$ is able to begin simulating immediately the manager and clerk. Otherwise, $\bar{p}$ must receive the communicated manager state. The protocol for receiving the state involves $\bar{p}$ simulating a step of p and a step of the clerk at every step of $\bar{p}$. In addition, $\bar{p}$ participates in the communication protocol at every opportunity: when $X = 0$ and there is an acknowledgement to send, $\bar{p}$ sends it. After the state is received, it is updated with the list of saved X-values.

We now sketch an argument for the correctness of the theorem. The number of values of the variable $\bar{X}$ is easy to check. We must show that $\bar{S}$ simulates S. Let $\bar{e}$ be any admissible execution sequence of $\bar{S}$. From $\bar{e}$, define an execution sequence e of S by extracting from $\bar{e}$ all of the S-steps simulated. (Clerk steps simulated in temporary mode are counted as ordinary clerk steps.) The paragraphs following the code are used to justify the fact that e is, in fact, an execution sequence of S. Since regions of $\bar{p}$ are defined to be the same as the regions of the simulated process p, and exactly the p-steps which appear in $\bar{e}$ are included in e, it is clear that the same set of region changes occur, in the same order.

It remains to show that e is admissible. If not, then either

(a) e is not worker-admissible, or

(b) e is infinite and either

(b1) the clerk only takes finitely many steps or

(b2) the manager only takes finitely many steps.

If $\bar{e}$ is finite, then admissibility of $\bar{e}$ implies that each $\bar{p}$ is in either its critical or remainder region at the end of $\bar{e}$. Then e is also finite, and each $p \in \text{proc}(S)$ is in either its critical or remainder region at the end of e. Thus, in this case, e is admissible. Thus, we assume from now on that $\bar{e}$ is infinite.

Examination of the code shows that every step of each $\bar{p}$ simulates a step of the corresponding p except possibly for those steps at the beginning of which the value of P is in prelim(p). Thus, in e, workers do not halt except in their critical or remainder regions, or in preliminary states, so that e is semi-worker-admissible. Examination of the code also shows that the manager does not halt in e unless it halts in a safe state.

We have ensured that some process is always responsible for simulating the clerk; since $\bar{e}$ is admissible and infinite, it follows that infinitely many clerk steps are simulated in e. Thus, the only ways in which e could fail to be admissible are if some process p stops in e in a preliminary state, or if e is infinite and the manager stops in e, in a safe state.

We argue that the communication of the manager state cannot be interrupted in $\bar{e}$. That is, once a process picks up a 'NEW' message, it is guaranteed eventually to receive the corresponding 'DONE' message. For if not, then there is some fixed communication value V which never gets delivered, in $\bar{e}$, to its intended recipient. Therefore, there must be infinitely many distinct steps of $\bar{e}$ after which V is located in the local variable M of some interrupting process. During this communication, the manager is stopped at a safe state in e. As we have already claimed, the clerk takes infinitely many steps in e. Also, e is semi-worker-admissible. By property (4′), all region changes eventually stop in e and the shared variable retains a value of 0 in e from some point onward. Thus, there is a point in $\bar{e}$ beyond which no further region changes occur and beyond which the *simulated* shared variable never takes on a value other than 0. After this point in $\bar{e}$, any process $\bar{q}$ holding value V in its local variable M will copy V into the shared variable X at the next step of $\bar{q}$, and V will never again be removed from the shared variable to be held in any process' local variable M. This is a contradiction.

We next argue that no process p can halt, in e, in a preliminary state. Assume the contrary: $\bar{p}$ continues to take steps in $\bar{e}$ since $\bar{p}$ remains in its trying or exit region and $\bar{e}$ is admissible. Then $\bar{p}$ must eventually continue looping forever in one of ⑵c–⑵e or ①. We consider cases.

If $\bar{p}$ loops in ⑵c, then the manager and clerk each take infinitely many steps in e. Since e is semi-worker-admissible, property (2′) implies that the manager eventually reaches a safe state. This (together with property (3)) contradicts the assumption that $\bar{p}$ loops in ⑵c.

If $\bar{p}$ loops in ⑵d, property (4′) leads to a similar contradiction.

If $\bar{p}$ loops in ⑵e or ③, then either a 'NEW' message is sent which never reaches its destination, or else the communication of the manager state is interrupted. The latter possibility has already been ruled out. The former possibility cannot occur because property (6′) implies that if a 'NEW' message is put into X, there is some process in its trying or exit region, but not in a preliminary state, available to receive the message.

We have thus argued that no process p halts in a preliminary state in e.

Finally, we argue that the manager cannot stop in e, in a safe state. Assume the contrary, and consider a point in e beyond which the manager takes no steps. Since infinitely many clerk steps occur in e and e is worker-admissible, property (4) implies

that all region changes eventually stop in e and the shared variable retains a value of 0 in e from some point onward. Thereafter, from some point onward, no processes are ever in preliminary states in e. After a corresponding point in $\bar{e}$, the only possibilities are that there is a value in free(S) in the shared variable X, that some process is in loop (2b), that a 'DONE' message has been sent but not received by its intended recipient and that a 'DONE' message has been received but the recipient process has not yet entered loop (2b). (Any other situation involves some process being in a preliminary state in e.)

Assume that a value in free(S) is in X. There must be a process in its trying or exit region but not in a preliminary state (since $\bar{e}$ is infinite and if no processes are in their trying or exit regions then any process which takes a step would change regions). The next time such a process takes a step it will detect the selection value and simulate a manager step, a contradiction.

If some process is in loop (2b), then (since its simulated process never again enters a preliminary state) its next step will simulate a manager step, a contradiction.

We have already argued that the communication of the manager state does not get interrupted. Thus, a 'DONE' message which is sent is eventually received by its intended recipient. Finally, once a 'DONE' message is received, examination of the code shows that the recipient process will enter loop (2b) on the following step. Therefore, the manager does not halt in e, and so e is admissible. ∎

IV. Applications

The remainder of the paper presents two small-shared-variable l-exclusion algorithms for hierarchical systems, and then appeals to Theorems 1 and 2 to obtain such algorithms for cooperative systems. We use many ideas from [2–4], but make a few changes in the interests of generality and simplicity.

For example, we describe the two algorithms as much alike as possible. We make the trying and exit protocols identical, each simply a general protocol in which a process requests a region change. Since we have a manager and clerk available, we concentrate as much of the computation as possible in the manager and clerk, trying to avoid having work which could be handled locally involve the communication. Since none of the ideas involved in the local computation pose any difficulty, we do not describe all the local computation in operational detail, but simply summarize local steps.

An $N + l + c$ FIFO *l-exclusion Algorithm*

THEOREM 3. *Let $l \geqslant 1$, $N \geqslant 2$. There is an $l+1$-regular hierarchical system S with $|\text{workers}(S)| = N$, and $|\text{values}(\text{var}(S))| = N + 10$, satisfying l-exclusion* (C1), *no l-deadlock* (C2), *and* FIFO (C4).

Proof. Figure 1 depicts the (trying or exit) protocol of a worker process. A worker goes to the WAIT subregion (with a possible detour in the HOLD subregion).

From there, it is eventually singled out to go to the $TALK_1$ subregion. There, it communicates its identity, its region (trying or exit) and other necessary information, to the manager. It then goes to $MAIN_1$. The manager collects information from several workers in this way, until it finally decides some particular worker is to be permitted to go to its critical or remainder region. The manager then communicates with each worker from the MAIN subregion in turn, by singling it out to go to the $TALK_2$ subregion and from there, to the $MAIN_2$ subregion until the manager locates the chosen worker. At this point, the manager allows the chosen worker to change regions and then communicates with all the processes in the $MAIN_2$ subregion to send them back to the $MAIN_1$ subregion.

The communication is carried out using a clever device from [3]. Each worker process, upon entering its trying or exit protocol, leaves its process number in X. If several processes enter in succession, each remembers the number of its predecessor as it leaves its own number in X. Periodically, the clerk replaces the process number in X with 0, adding that process number to the end of a "tail list" kept by the manager. The next worker entering its protocol after X has been set to zero does not know the number of its predecessor (but recognizes its ignorance). However, *if the manager and all workers are free to communicate, it should be clear that together they can reconstruct the total arrival order.*

The shared variable X of S has values$(X) = \{0,..., N\} \cup \{$'$TALK_1$', '$TALK_2$',

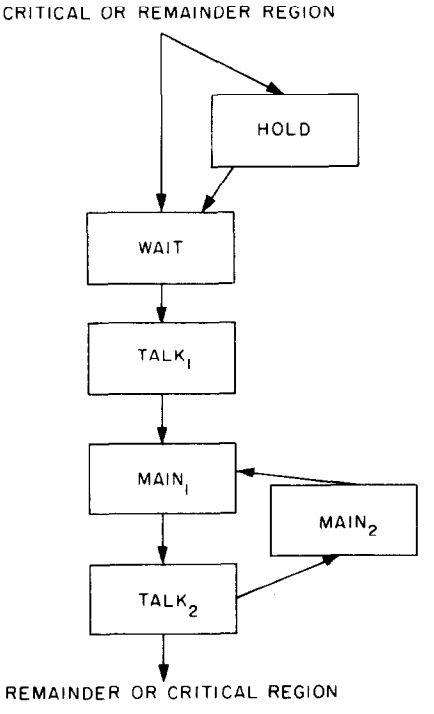

FIG. 1. Trying or exit protocol for the FIFO algorithm.

'SEND', 'ACK', 'DONE', 'YES', 'NO', 'MOVE', 'OK'}; init(S) = 0. Elements of the first set in the union are called *ordinary values* and those of the second set are called *communication values*. Communication values are only placed in X to replace the ordinary value of 0.

We first give the programs for the manager and clerk of S. The combined state of the manager and clerk has local variables

NEXT	for recording the number of a worker chosen to be permitted a requested region change,
CR	for recording the number of workers in their critical regions,
QUEUE	for a queue of *process numbers*, *status indicators* ('TRYING' or 'EXIT'), and tags ('$MAIN_1$' or '$MAIN_2$') of processes in the $MAIN_1$, $MAIN_2$ and $TALK_2$ subregions, kept ordered by their order of entry to their trying or exit protocols,
TAILLIST	for a data structure designed to help form QUEUE. It contains a list of process numbers for tails of sublists eventually to be appended to QUEUE.
WORKING	for another data structure to help form QUEUE. It contains an unordered list of *process numbers*, *status indicators* ('TRYING' or 'EXIT') and *predecessor numbers* (for predecessors in the order of entry to trying or exit regions).
C	for holding a code being received,
FINISH	for indicating the end of receipt of a message,
J	for a received process number,
STATUS	for a received status indicator, and
PRED	for a received predecessor number.

All variables are initialized at 0. The starting location counter value of clerk(S) is the single lock statement of its program (implicit in the waitfor construct). The starting location counter value of manager(S) is the last lock statement of its program.

Code and *decode* are similar to the corresponding functions in Algorithm A. The pair (TAILLIST, WORKING) is said to be *complete* provided for every process number p in TAILLIST, it is the case that WORKING contains a chain starting with p and ending with 0 (i.e., triples of the form (p_i, a_i, p_{i+1}) appear in WORKING, for $0 \leqslant i \leqslant j$, where $p_0 = p$ and $p_j = 0$). The statement "append (TAILLIST, WORKING) to QUEUE," for a complete pair (TAILLIST, WORKING), means to combine the information represented in TAILLIST AND WORKING to form a total arrival order for the represented worker processes, and to append this order to the end of the QUEUE. The actual information added to the QUEUE, for each process, is just its process number together with its status ('TRYING' or 'EXIT', as recorded in WORKING) and the tag '$MAIN_1$'.

The statement "NEXT $\leftarrow$ chosen process, if any," means to decide based on the current QUEUE and CR count whether some process is to be permitted to change regions; if so then NEXT is set to the process number of the next such process, while if not, then NEXT is set to 0. The proper execution of this statement by the manager

is the key to all of the required properties (C1), (C2), and (C4). That is, the manager must ensure that no more than l workers are ever allowed simultaneously into their critical regions, that FIFO order is not violated in either the trying or exit regions, that workers continue to move from their trying regions to their critical regions if there is room, and that workers continue to leave their exit regions in any case. To be definite, we might cause the manager to alternate between phases when it allows one worker in its exit region to go to its remainder region (if possible) and phases when it allows one worker in its trying region to go to its critical region (if possible). In the first phase, the manager lets the first worker in QUEUE with status 'EXIT' (if any) go to its remainder region. In the second phase, it only lets a worker go to its critical region if there is room ($CR < l$) and in this case it lest the first worker in QUEUE with status 'TRYING' (if any) go to its critical region.

Other local action statements should be self-explanatory.

```
    Clerk(S)
    while true do [waitfor X ∈ {1,..., N}; TAILLIST ← TAILLIST, X; X ← 0]
    Manager(S)
    while true do
①     [while (TAILLIST, WORKING) is incomplete do
        [waitfor X = 0; X ← 'TALK1'; waitfor X = 'OK'; X ← 'ACK';
         while FINISH = 0 do
           [waitfor X ∈ {'SEND', 'DONE'};
            if X = 'SEND' then [X ← 'ACK'; C ← C + 1] else [FINISH ← 1; X ← 0]]
         (J, STATUS, PRED) ← decode(C); C ← 0; FINISH ← 0;
         add (J, STATUS, PRED) to WORKING;
         if STATUS = 'EXIT' then CR ← CR − 1;
         J ← 0; STATUS ← 0; PRED ← 0];
②     append (TAILLIST, WORKING) to QUEUE; TAILLIST ← 0; WORKING ← 0;
③     NEXT ← chosen process, if any;
      if NEXT ≠ 0 then
        [while J ≠ NEXT do
           [waitfor X = 0; X ← 'TALK2'; waitfor X = 'OK'; X ← 'ACK';
            while FINISH = 0 do
              [waitfor X ∈ {'SEND', 'DONE'};
               if X = 'SEND' then [X ← 'ACK'; C ← C + 1] else FINISH ← 1]
            J ← decode(C); C ← 0; FINISH ← 0;
            if J ≠ NEXT then
              [change J's tag to 'MAIN2' in QUEUE;
               X ← 'NO'; waitfor X = 'OK'; X ← 0]]
         if NEXT's status in QUEUE = 'TRYING' then CR ← CR + 1;
         remove NEXT from QUEUE; X ← 'YES'; waitfor X = 'OK'; X ← 0;
         NEXT ← 0; J ← 0]
④     while there is a process in QUEUE with tag 'MAIN2' do
        [waitfor X = 0; X ← 'MOVE'; waitfor X = 'OK'; X ← 0;
         change some tag in QUEUE from 'MAIN2' to 'MAIN1']
      unlock; lock]
```

ALGORITHM B(i)

The clerk simply executes a loop which zeros the shared variable whenever it sees a process number, adding the number to the local TAILLIST. The manager executes loop ①, which assembles all the newly available process entry order information. That is, it repeatedly puts the value 'TALK$_1$' into X, to initiate communication with an arbitrary worker process in the WAIT subregion, then executes a protocol to receive the worker's process number, status and predecessor (in unary), decrementing the CR-count if the worker informs the manager that it is in the exit region. The manager continues executing ① until it has complete entry order information. (While the manager is accumulating this information, the clerk can continue to add process numbers to TAILLIST. However, since there are only finitely many workers, eventually a time must be reached when (TAILLIST, WORKING) is complete.)

The manager then executes ②, which adds the newly assembled information to the end of the QUEUE. It next executes ③; it decides whether a process is to be selected, and if so, attempts to locate it. The manager repeatedly puts the value 'TALK$_2$' into X to initiate communication with an arbitrary worker process in the MAIN$_1$ subregion, then executes a protocol to receive the worker's process number. As long as the process is not the correct one, the manager sends a response of 'NO' to it. When the correct process is encountered, the manager sends a response of 'YES'. All of the incorrect worker processes involved in locating the correct process will end this phase of execution in the MAIN$_2$ subregion, so the manager executes ④, a loop which tells these processes, one by one, to return to the MAIN$_1$ subregion.

Next, we give the program for process $p \in$ workers(S). Process p has local variables as follows:

I	which stores a process number identifying p, in the range $1 \leqslant I \leqslant N$,
STATUS	which holds the value 'TRYING' or 'EXIT', identifying the region of the protocol,
PRED	for recording the predecessor,
C	for holding a code being sent, and
M	for holding an interrupted communication value.

The initial value of I is an identifier for p, the initial value of STATUS is 'EXIT', and all other variables are initialized at 0. The starting location counter value of p is the last lock statement of the program.

```
        Process p
        while true do
①         [if X ∈ {0,..., N} then [PRED ← X; X ← I]
                             else [M ← X; X ← I; waitfor X = 0; X ← M; M ← 0];
②         waitfor X = 'TALK1'; X ← 'OK'; C ← code(I, STATUS, PRED);
          while C > 0 do [waitfor X = 'ACK'; X ← 'SEND'; C ← C − 1];
          waitfor X = 'ACK'; X ← 'DONE';
③ MAIN1:  waitfor X = 'TALK2'; X ← 'OK'; C ← code(I);
          while C > 0 do [waitfor X = 'ACK'; X ← 'SEND'; C ← C − 1];
```

```
waitfor X = 'ACK'; X ← 'DONE';
waitfor X ∈ {'YES', 'NO'};
if X = 'NO' then [X ← 'OK'; waitfor X = 'MOVE'; X ← 'OK'; goto MAIN_1 |];
X ← 'OK';
unlock; lock;
unlock; lock; STATUS ← if STATUS = 'TRYING' then 'EXIT' else 'TRYING' |
```

ALGORITHM B(ii)

The regions of p are defined as follows: Any state s of p for which the location counter is at the final lock statement of the program is in either the critical or remainder region; if the value of local variable STATUS in s is 'EXIT' then $s \in R(p)$, and if the value is 'TRYING' then $s \in C(p)$. In all other cases, $s \in T(p) \cup E(p)$; if the value of STATUS in s is 'TRYING' then $s \in T(p)$, and if it is 'EXIT' then $s \in E(p)$.

Process p, upon entry to its protocol, executes ①, which leaves p's identifier in X; if a communication value is seen in X, p holds the communication value, waiting for the first opportunity to replace it in X. Next p executes ②. Process p waits (in the WAIT subregion) until it sees a 'TALK$_1$' message from the manager. This is p's signal to communicate its process number, status and predecessor process number to the manager. After this communication protocol is completed, p moves to the MAIN$_1$ subregion. Process p then executes ③. Then p waits in the MAIN$_1$ subregion until it sees a 'TALK$_2$' message from the manager. This is p's signal to communicate its process number to the manager. Process p waits for the manager's decision, 'YES' or 'NO', on whether p can change regions. If 'NO' is seen, p waits in subregion MAIN$_2$ for a 'MOVE' message, before returning to MAIN$_1$. If 'YES' is seen, p executes a NO-OP and changes region.

In the following paragraphs, we give further arguments for the claimed properties of Algorithm B. We rely to a certain extent on the reader's understanding of the protocols to convince him of their correctness properties; we provide more detailed arguments to show that the various communications do not indefinitely block each other.

We now outline why Algorithm B satisfies (C1), (C2), and (C4) and is $l+1$-regular. Properties (C1) and (C4) follow directly from the manager's correct choice of a selection policy. Property (C2) also depends on a correct selection policy, but in addition requires us to argue that all communication values sent eventually reach their intended recipients (in spite of interruptions).

For showing $l+1$-regularity, we define the safe manager states to be exactly those in which the location counter is at the final lock of the manager's program. With this definition, it is easy to check properties (1), (3)–(5) of the definition of $l+1$-regularity. To observe (6), note that the states of free(S) are exactly those in which the manager's program counter is at the final lock, and in which NEXT, QUEUE, TAILLIST, WORKING, C, FINISH, J, STATUS, and PRED all have the value 0. Thus, there is exactly one free state for each possible value of CR, $0 \leqslant \mathrm{CR} \leqslant l$. Verification of property (2) requires us to argue that all communication values sent

eventually reach their intended recipients; if this is so, then the manager will not remain indefinitely in any of parts ①, ②, ③, or ④ of its code.

Assume some communication value V does not reach its intended recipient, in some admissible execution sequence e of S. Then there must be infinitely many distinct steps of e after which V is located in the local variable M of some interrupting worker process. Eventually in e, no further region changes occur (since if communication is interrupted, the manager will stop allowing processes to change regions). Thereafter, from some point onward, the only values for X are 0 and V (since the clerk continues to operate, replacing process numbers in X with 0). After this point, any process p holding V in its local variable M will copy V into X at p's next step, and V will never again be removed from X to be hold in any worker's local variable M (since no further processes change region). This is a contradiction. ∎

COROLLARY 3.1. *Let* $l \geqslant 1$, $N \geqslant 2$. *There exists a cooperative system* S *with* $|\text{proc}(S)| = N$ and $|\text{values}(\text{var}(S))| = N + l + 15$, *satisfying l-exclusion* (C1), *no l-deadlock* (C2) *and* FIFO (C4).

Proof. The proof is by Theorems 1–3. (Note that some values of the shared variable need to be renamed in order to preserve disjointness of ordinary and communication values in the simulation construction.) ∎

THEOREM 4. *Let* $l \geqslant 1$, $N \geqslant 2$. *There is an* $l+1$*-regular hierarchical system* S *with* $|\text{workers}(S)| = N$ *and* $|\text{values}(\text{var}(S))| = \lceil N/2 \rceil + 13$, *satisfying l-exclusion* (C1), *no l-deadlock* (C2), *and no infinite bypass* (C3).

Proof. This proof differs from the previous proof in the protocol up to the TALK_1 subregion, but keeps the same strategy for passing through and for managing the MAIN_1, TALK_2, and MAIN_2 subregions. The protocol used up to the TALK_1 subregion is based on the surprisingly space-efficient "executive protocol" of Algorithm C of [4]. Figure 2 depicts the (trying or exit) protocol of a worker process. Two additional subregions, EXEC and IDLE, which did not appear in the protocol for Algorithm B, are included in the present protocol.

Each worker process, upon entering its trying or exit protocol, attempts to increment a counter kept in X by 1 in order to communicate its presence to the manager. (There are insufficiently many permissible values of X for each process to be able to leave a unique identifier in X, so the entry strategy of Algorithm B cannot be used.) If this increment succeeds, and if the worker subsequently encounters a 'TALK_1' message in X, the worker communicates its process number and status to the manager, and then proceeds to the MAIN_1 subregion, continuing from that point as in the previous algorithm. However, a difficulty arises if a worker attempting to increment the value in X sees that the maximum possible count $\lceil N/2 \rceil$ is already there. In this case, the entering worker "becomes an executive," resetting X to 0 and thereby temporarily hiding the presence of himself and $\lceil N/2 \rceil$ other workers from the manager and clerk. The executive, in the EXEC subregion, sends out special STOP messages to cause $\lceil N/2 \rceil$ workers in WAIT (plus perhaps some additional workers

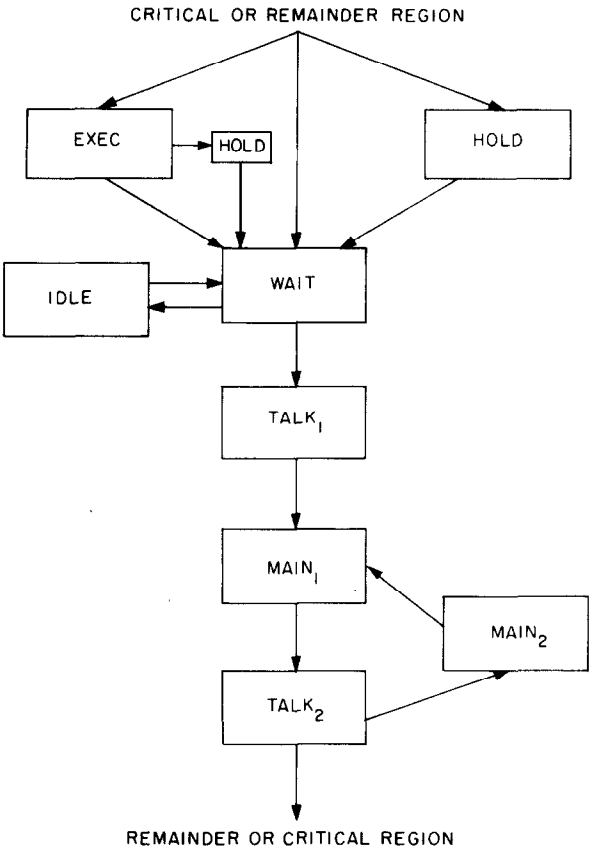

FIG. 2. Trying or exit protocol for the algorithm which avoids infinite bypass.

who enter their protocols in the meantime) to move to a separate IDLE subregion. The executive then proceeds to attempt the increment a second time. This time, having ensured that sufficiently many workers are "out of the way" (in the IDLE subregion), the executive encounters no difficulty and accomplishes the increment.

Periodically, the clerk replaces the count in X with 0, adding the count found in X to a local variable COUNT of the manager. Thus, the manager always has a count of some of the workers in the WAIT and HOLD subregions, but is possibly missing some of those which are destined to receive STOP messages and go to the IDLE subregion, and some whose increments have not yet been replaced in X with 0, by the clerk. However, once the executive completes the protocol which sends all of the necessary workers to the IDLE subregion, and then completes its own increment of X, the manager's count of the number of workers in the WAIT and HOLD subregions (including the executive) is missing only those whose increments are not yet recorded by the clerk. Then the manager is able to ensure that it converses with

each worker from the WAIT and HOLD subregions, one at a time, by sending 'TALK$_1$' messages as in the previous algorithm.

When non-executive workers in the WAIT subregion encounter a 'TALK$_1$' message, they respond with their process number and move to the MAIN$_1$ subregion, continuing as in the previous algorithm. When an executive in the WAIT subregion encounters a 'TALK$_1$' message, it must do more, however. For each worker in the IDLE subregion, the executive sends a special 'GO' message to the worker (which will then move into the WAIT subregion) and increments X so that the clerk can record the presence of a new worker in the WAIT subregion. (Since these workers are being introduced by the executive one at a time, there is no danger of too many increments occurring at once and a worker attempting to increment a value of $X = \lfloor N/2 \rfloor$.) Only when all of the workers from IDLE have been integrated into the system in this way does the executive complete its executive function and respond to the manager with its process number. From that point onward, the executive behaves exactly like the other workers. The manager can thus assemble a QUEUE corresponding to the one in Algorithm B; however, the QUEUE in Algorithm C represents order of receipt of 'TALK$_1$' messages rather than protocol arrival order.

It must be noted that two executives cannot exist at once. From the time one worker becomes an executive until it responds to the manager with its process number, there are always at least $\lceil N/2 \rceil$ processes other than the executive in the WAIT, HOLD, and IDLE subregions. Thus, there are insufficiently many workers to cause another increment past $\lceil N/2 \rceil$ to occur.

The shared variable X of S has values$(X) = \{0, 1, ..., \lceil N/2 \rceil\} \cup \{$'TALK$_1$', 'TALK$_2$', 'SEND', 'ACK', 'DONE', 'YES', 'NO', 'MOVE', 'OK', 'STOP', 'GO', 'GONE'$\}$; init$(S) = 0$. Elements of the first set in the union are called *ordinary values*, and those of the second set except for 'STOP' are called *communication values*. Communication values are only placed in X to replace the ordinary value of 0.

We first give the programs for the manager and clerk of S. The combined state has local variables

NEXT, CR, C, FINISH, J, and STATUS as in Algorithm B(i),

QUEUE for a queue of *process numbers*, *status indicators* ('TRYING' or 'EXIT') and *tags* ('MAIN$_1$' or 'MAIN$_2$') of processes in the MAIN$_1$, MAIN$_2$, and TALK$_2$ subregions, kept ordered by their order of receipt of 'TALK$_1$' messages, and

COUNT to hold a number of workers known to be in WAIT and HOLD.

All variables are initialized at 0. The starting location counter value of clerk(S) is its single lock statement and that of manager(S) is the last lock statment of its program.

Code and *decode* are as before. The statement "add T to QUEUE" adds triple T to the end of the QUEUE.

The manager code is *identical* to that of Algorithm B(i) from section ② onward; moreover, ① only differs from ① of B(i) in its decoding function, its disposition of the received triple and its handling of the variable COUNT.

```
    Clerk(S)
    while true do [waitfor X ∈ {1,..., ⌈N/2⌉}; COUNT ← COUNT + X; X ← 0]
    Manager(S)
①   while true do
      [while COUNT > 0 do
        [waitfor X = 0; X ← 'TALK1'; waitfor X = 'OK'; X ← 'ACK';
         while FINISH = 0 do
          [waitfor X ∈ {'SEND', 'DONE'};
           if X = 'SEND' then [X ← 'ACK'; C ← C + 1] else [FINISH ← 1; X ← 0]]
         (J, STATUS) ← decode(C); C ← 0; FINISH ← 0;
         add (J, STATUS, 'MAIN1') to QUEUE;
         if STATUS = 'EXIT' then CR ← CR − 1;
         COUNT ← COUNT − 1;
         J ← 0; STATUS ← 0];
②     append (TAILLIST, WORKING) to QUEUE; TAILLIST ← 0; WORKING ← 0;
③     NEXT ← chosen process, if any;
      if NEXT ≠ 0 then
        [while J ≠ NEXT do
          [waitfor X = 0; X ← 'TALK2'; waitfor X = 'OK'; X ← 'ACK';
           while FINISH = 0 do
            [waitfor X ∈ {'SEND', 'DONE'};
             if X = 'SEND' then [X ← 'ACK'; C ← C + 1] else FINISH ← 1]
           J ← decode(C); C ← 0; FINISH ← 0;
           if J ≠ NEXT then
            [change J's tag to 'MAIN2' in QUEUE;
             X ← 'NO'; waitfor X = 'OK'; X ← 0]]
         if NEXT's status in QUEUE = 'TRYING' then CR ← CR + 1;
         remove NEXT from QUEUE; X ← 'YES'; waitfor X = 'OK'; X ← 0;
         NEXT ← 0; J ← 0]
④     while there is a process in QUEUE with tag 'MAIN2' do
        [waitfor X = 0; X ← 'MOVE'; waitfor X = 'OK'; X ← 0;
         change some tag in QUEUE from 'MAIN2' to 'MAIN1']
      unlock; lock]
```

ALGORITHM C(i)

Next, we give the program for process $p \in \text{workers}(S)$, p has local variables I, STATUS, C, and M as before,

COUNT	used if p becomes an executive, to hold a count of workers to be sent to the IDLE subregion,
IDLERS	to hold a count of workers which p has caused to enter the IDLE subregion.

The initial value of I is an identifier for p, in the range $1 \leqslant I \leqslant N$, the initial value of STATUS is 'EXIT', and all other variables are initialized at 0. The starting location counter value of p is the last lock statement of the program. The worker code is identical to that of Algorithm B(ii) from the middle of Section ③ (of Algorithm C(i)) onward.

```
                Process p
①               while true do
                  [if X = |N/2| then
                    [X ← 0; COUNT ← |N/2|;
                     while COUNT > 0 do
                       [if X ∈ {0,..., |N/2| − 1} then COUNT ← COUNT + X else M ← X;
                        X ← 'STOP';    IDLERS ← IDLERS + 1;    COUNT ← COUNT − 1;    waitfor
                        X ≠ 'STOP']
                     if M ≠ 0 then [waitfor X = 0; X ← M; M ← 0]];
②                 if X = 'STOP' then [X ← 1; waitfor X = 'GO'; X ← 'GONE']
                  else if X ∈ {0,..., |N/2| − 1} then X ← X + 1
                       else [M ← X; X ← 1; waitfor X = 0; X ← M; M ← 0];
③                 waitfor X = 'TALK1';
                  while IDLERS > 0 do
                    [X ← 'GO'; waitfor X = 'GONE'; X ← 1; IDLERS ← IDLERS − 1; waitfor X = 0];
                  X ← 'OK'; C ← code(I, STATUS);
                  while C > 0 do [waitfor X = 'ACK'; X ← 'SEND'; C ← C − 1];
                  waitfor X = 'ACK'; X ← 'DONE';
④ MAIN1:          waitfor X = 'TALK2'; X ← 'OK'; C ← code(I);
                  while C > 0 do [waitfor X = 'ACK'; X ← 'SEND'; C ← C − 1];
                  waitfor X = 'ACK'; X ← 'DONE';
                  waitfor X ∈ {'YES'; 'NO'};
                  if X = 'NO' then [X ← 'OK'; waitfor X = 'MOVE'; X ← 'OK'; goto MAIN1];
                  X ← 'OK';
                  unlock; lock;
                  unlock; lock; STATUS ← if STATUS = 'TRYING' then 'EXIT' else 'TRYING']
```

ALGORITHM C(i))

The regions of p are defined exactly as for Algorithm B(ii).

Process p, upon entry to its protocol, first checks to see if it is to become an executive. If so, then the body of the main conditional in ① is executed, which sends 'STOP' messages to as many workers as there are increments removed by p from X. Along the way, if additional increments are seen by p, those are also included in the number of workers to be idled. While executing the stopping protocol, p might see a communication value; if so, p holds the value until the end of the stopping protocol, and then replaces the value in X at the first opportunity. After completing the stopping protocol and replacing any communication values it might have picked up, p goes on to ②. (If p is not to become an executive, p goes immediately to ② upon entry to its protocol.)

In ②, process p succeeds in incrementing the count in the variable X. In addition, if p sees a 'STOP' message, p goes to the IDLE subregion, from there waiting to be permitted to go to the WAIT subregion. (In this case, p is accepting a 'STOP' message intended originally for another process, but the interchange will not affect any of the desired properties of the algorithm.) If p sees a communication value, p holds the value in order to accomplish the increment, and later replaces that value in X at the first opportunity.

In ③, p waits for a 'TALK$_1$' message. If p is not an executive (IDLERS = 0) then P is ready to transmit its process number and status. If p is an executive, then p first executes a protocol to send all workers in the IDLE subregion to the WAIT subregion, informing the manager of the result by setting X to 1 for each transfer of one process. After this protocol, p is ready to transmit its process number and status. From this point, the code is identical to that in Algorithm B(ii).

As before, we rely on the reader's understanding of the protocols to convince him of their correctness properties; we argue further why the various communications do not indefinitely block each other.

We require Algorithm C to satisfy (C1)–(C2), (C3) and to be $l+1$-regular. Properties (C1) and (C3) depend on the manager's correct choice of selection policy, while (C2) depends on both having a correct selection policy and the fact that all communication values and 'STOP' messages sent, eventually reach their intended recipients (in spite of interruptions).

For showing $l+1$-regularity, we define the safe manager states as in the previous algorithm. Properties (1), (3)–(5) are again straightforward, and (6) follows as for Algorithm B. Property (2) again requires us to argue that all communication values and 'STOP' messages sent, eventually reach their intended recipients.

But 'STOP' messages are never interrupted, and so must reach their intended recipients. Also, if some communication value X does not reach its intended recipient, in some admissible execution sequence e of S, then the same contradiction is reached as for Theorem 3. ∎

COROLLARY 4.1. *Let* $l \geqslant 1$, $N \geqslant 2$. *There exists a cooperative system* S *with* $|\text{proc}(S)| = N$ *and* $|\text{values}(\text{var}(S))| = \lfloor N/2 \rfloor + l + 18$, *satisfying l-exclusion* (C1), *no l-deadlock* (C2), *and no infinite bypass* (C3).

Proof. The proof is by Theorems 1, 2, and 4. ∎

ACKNOWLEDGMENTS

The authors thank Nancy Griffeth and Jim Burns for helpful suggestions and considerable listening time.

REFERENCES

1. E. DIJKSTRA, Solution of a problem in concurrent programming control, *Comm. ACM* **9** (1965), p. 569.
2. A. CREMERS AND T. HIBBARD, Mutual Exclusion of N processors using an $O(N)$-valued message variable, *in* "Proceedings, 5th ICALP Udine, Italy," Springer Lecture Notes in Computer Science, No. 62, pp. 165–176, July 1978.
3. A. CREMERS AND T. HIBBARD, "Arbitration and Queueing Under Limited Shared Storage Requirements," Forschungsbericht, No. 83, University of Dortmund, 1979.
4. J. E. BURNS, M. J. FISCHER, P. JACKSON, N. A. LYNCH, AND G. L. PETERSON, Data requirements for

implementation of N-Process mutual exclusion using a single shared variable, *J. Assoc. Comput. Mach.* **29** (2) (1982) 183–205.

5. M. J. FISCHER, N. A. LYNCH, J. E. BURNS, AND A. BORODIN, Resource allocation with immunity to limited process failure, *in* "Proceedings, 20th Annual Symposium on Foundations of Computer Science," Puerto Rico, 1979; GIT-ICS-79/10.
6. N. A. LYNCH AND M. J. FISCHER, "On Describing the Behavior and Implementation of Distributed Systems," GIT-ICS-79/03; Semantics of Concurrent Computation Proceedings, pp. 147–171, Evian, France, preliminary version in Lecture Notes in Computer Science, in *Theoret. Comput. Sci.*, in press.
7. N. A. LYNCH, Upper bounds for static resource allocation in a distributed system, *J. Comput. System Sci.* **23**, (2) (1981), 254–278.
8. G. L. PETERSON, "New Bounds on Mutual Exclusion Problems," TR-68, The University of Rochester, 1980.