Semaphore Primitives and Starvation-free Mutual Exclusion

Computation Structures Group Memo 194
June 1980

Eugene W. Stark

This research was supported by a National Science Foundation Graduate Fellowship.

Semaphore Primitives and Starvation-Free Mutual Exclusion

EUGENE W. STARK

Massachusetts Institute of Technology, Cambridge, Massachusetts

Abstract. Most discussions of semaphore primitives in the literature provide only an informal description of their behavior, rather than a more precise definition. These informal descriptions may be incorrect, incomplete, or subject to misinterpretation. As a result, the literature actually contains several different definitions of the semaphore primitives. The differences are important, since the particular choice of definition can affect whether a solution to the mutual exclusion problem using semaphore primitives allows the possibility of process *starvation*. An attempt is made to alleviate some of the confusion by giving precise definitions of two varieties of semaphore primitives, here called *weak* and *blocked-set* primitives. It is then shown that under certain natural conditions, although it is possible to implement starvation-free mutual exclusion with blocked-set semaphores, it is not possible to do so with weak semaphores. Thus weak semaphores are strictly less "powerful" than blocked-set semaphores.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *mutual exclusion*; *synchronization*

General Terms. Algorithms, Theory

Additional Key Words and Phrases: Semaphores

1. *Introduction*

The *mutual-exclusion problem* is the problem of synchronizing a number of concurrently executing sequential processes to ensure that the execution of certain *critical regions* of the program for one process is not interrupted by the execution of similar regions of the program for another process. Various *synchronization primitives*, or programming language constructs for expressing synchronization, have been proposed for dealing with this problem. Perhaps the most well known of these synchronization primitives are the so-called *semaphore primitives*.

Most discussions of semaphore primitives in the literature provide only an informal description of their behavior. These informal descriptions may be incorrect, incomplete, or subject to misinterpretation. As a result, the literature actually contains several different definitions of the semaphore primitives. The differences are important, since the particular choice of definition can affect whether a solution to the mutual-exclusion problem using semaphore primitives allows the possibility of process *starvation*.

This work was supported in part by a National Science Foundation Graduate Fellowship.

Author's address: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139.

This paper attempts to alleviate some of the confusion by giving precise definitions of two varieties of semaphore primitives, here called *weak* and *blocked-set* primitives. The properties of *symmetry*, *no busy-waiting*, and *no memory* are identified as natural conditions satisfied by "good" solutions to the mutual-exclusion problem. It is then shown that under certain combinations of these conditions, although it is possible to implement starvation-free mutual exclusion with blocked-set semaphores, it is not possible to do so with weak semaphores. Thus weak semaphores are strictly less powerful than blocked-set semaphores.

1.1 VARIOUS INFORMAL DEFINITIONS OF SEMAPHORES. Dijkstra [6] defines a semaphore as a special type of program variable, shared between processes, which may be manipulated only by two special operations, designated **P** and **V**. A semaphore variable may take on only nonnegative integer values. His definition of the effect of the semaphore operations is as follows. A process performing a **P** operation on a semaphore variable s tests the value of s to see if it is greater than zero. If so, then s is decremented, and the process proceeds. The test and resultant decrement are performed in one indivisible step. If the value of s is not greater than zero, the process is said to become *blocked* on the semaphore s and must wait to be signaled by some process executing a **V**(s) operation. A process executing a **V**(s) operation checks to see if there are any processes blocked on s. If there are blocked processes, then one of them is signaled and allowed to proceed. If there are no blocked processes, then s is simply incremented. The **V**(s) operation is assumed to be performed in a single indivisible step.

In Dijkstra's definition of semaphores, processes that are blocked within a **P** operation on a semaphore variable s are distinguished from processes that are about to execute a **P**(s) but have not yet become blocked. This distinction is important in that the execution of a **V**(s) will cause a blocked process to be selected in preference to a process that is not blocked. However, all blocked processes are treated equally as far as being selected is concerned: no effort is made to distinguish processes that have been blocked for a short length of time from those that have been blocked for a longer period. The group of blocked processes at any instant of time may therefore be modeled as a *set*, from which a **V** operation chooses at random a process to be signaled. Let us call semaphores with this type of blocking discipline *blocked-set* semaphores. It is also possible to define *blocked-queue* semaphores, which are like blocked-set semaphores except that the group of blocked processes is maintained as a FIFO *queue*, instead of as a set. Processes becoming blocked are placed at the end of the queue, and processes are selected for signaling from the head of the queue.

As was mentioned above, it appears that blocked-set semaphores are the type that Dijkstra intended to define in [6]. In [7] he indicated the possibility of defining blocked-queue semaphores as well. Blocked-set semaphores also appear to be the type used in [4, 5, 9, 15]. However, a third type of semaphore, much different than either blocked-set or blocked-queue semaphores, is also found in the literature. This is the type of semaphore that may be implemented with indivisible "test-and-set" instructions as follows. A process attempting to perform a **P** operation on a semaphore variable s executes a busy-waiting loop in which the value of s is continually tested. As soon as s is discovered to have a value greater than zero, it is decremented, the decrement and immediately preceding test being performed as one indivisible step. A **V** operation simply increments s in an indivisible step. We will call this type of semaphore a *weak* semaphore. We prove that this choice of terminology is justified, in that weak semaphores are significantly "weaker" than blocked-set or blocked-queue semaphores when their starvation properties are considered.

```
semaphore s initially 1,
loop: ⟨noncritical region⟩
      P(s);
      ⟨critical region⟩
      V(s),
      goto loop;
```

FIG. 1. Solution to the mutual-exclusion problem.

Definitions of weak semaphores equivalent to the definition above may be found in [13, 19, 20]. However, it is here that confusion over the definitions of the semaphore operations becomes evident. The definitions given by Presser [19] are at best incomplete. Under the most straightforward interpretation, though, they appear to define weak semaphores. Habermann [10] criticizes Presser for this, and for presenting a definition of weak semaphores without indicating the difference between this version and the blocked-set semaphores defined by Dijkstra.

Dijkstra [6] also distinguishes between *binary* and *general* semaphores. In the discussion above we informally defined weak, blocked-set, and blocked-queue *general* semaphores. *Binary* semaphores are similar to general semaphores, except that the binary semaphore variable may take on only the values zero and one. The effect of a binary **P** operation is identical to that of a general **P** operation. However, to ensure that the value of the variable s never exceeds one, a binary V(s) operation will simply set s to one, rather than incrementing s as is done in a general V(s) operation. Note that if the value of a binary semaphore variable s is one, which implies that there are no processes blocked on s, then execution of a V(s) operation has no effect.

The types of semaphore primitives found in the literature are by no means limited to those described above. For example, [3, p. 68] defines a kind of weak semaphore whose behavior is significantly different from the other definitions discussed above. As a final indication of the confusion in the literature on semaphores, it is interesting to note that all the sources above claim that their definitions define "Dijkstra's semaphore primitives."

1.2 STARVATION PROPERTIES OF THE VARIOUS DEFINITIONS. The weak, blocked-set, and blocked-queue semaphore primitives defined above have different starvation properties. To see why this might be true, let us see what happens when each definition is used in a simple attempt to solve the mutual-exclusion problem. Consider a number of processes, each executing the program shown in Figure 1. Each process continually alternates between its *critical region* and its *noncritical region*. In order to ensure that mutual exclusion of critical regions among all the processes is obtained, the critical region is bracketed by a P(s)–V(s) pair. Since the value of the semaphore variable s is initially one, and a process desiring to enter the critical region must first perform a P(s) operation, whenever some process is in its critical region, the value of s is zero. Hence other processes attempting to perform **P** operations and enter their own critical regions must wait. Mutual exclusion is therefore obtained regardless of whether weak, blocked-set, or blocked-queue semaphores are used.

Suppose that the semaphore operations are of the weak variety, and consider the execution of two processes, process 1 and process 2. Suppose that process 1 finds the value of s to be one and proceeds into its critical region. Since the value of s is now zero, process 2 is unable to complete its P(s) operation and therefore waits within the **P** operation for the value of s to become positive. Now suppose that process 1 completes execution in its critical region and performs the V(s) operation, setting s to one. Since we have assumed the semaphore operations to be weak, process 2 does not complete its P(s) operation immediately but must retest the semaphore variable

s. It is possible, if process 1 executes quickly enough, for it to loop around and perform another P(s) operation, resetting s to zero, *before process* 2 *could get around to noticing that s ever had the value one.* This scenario may continue indefinitely, with the result that process 2 "starves" forever within its P(s) operation. Note that this argument relies on the fact that in determining the behavior of a system of concurrent processes, we may make no assumptions about the relative speeds of the processes (except for the finite delay property to be discussed below) and must consider all possible orders of executions of steps of the processes as legitimate.

Now, suppose instead that the semaphore operations are defined to be blocked-set operations. The scenario described in the preceding paragraph is no longer possible, since the execution of a V operation by process 1 immediately causes process 2 to complete its P(s) operation. Since s is never set to one, it is not possible for process 1 to complete another P(s) before process 2 finishes its critical region and performs a V(s). However, although starvation is no longer possible with two processes, with three or more processes it again becomes possible for a process to wait forever within the P(s) while other processes successfully complete infinitely many P(s) operations. The reason for this is that the blocked-set V operation selects the blocked process to signal at random and, in particular, gives no preference to a process that may have been blocked for a long time. The situation may be remedied if blocked-queue semaphores are used.

1.3 RELATIVE POWER OF THE TYPES OF SEMAPHORES. The simple scenario just presented indicates that although weak, blocked-set, and blocked-queue semaphores are all able to implement mutual exclusion of critical regions, the three types of semaphores are evidently not equivalent if the possibility of starvation is taken into consideration. We are interested in obtaining more detailed information of this type concerning the relative power of the different kinds of semaphore primitives. We obtain this information by posing and answering questions of the form: Under certain natural constraints, is it possible to implement starvation-free mutual exclusion with a given kind of semaphore? It turns out to be trivially possible to implement starvation-free mutual exclusion with either blocked-queue binary or blocked-queue general semaphores, under any of the constraints we impose. We therefore concentrate our efforts on determining the differences in power between weak binary, weak general, blocked-set binary, and blocked-set general semaphores.

The flavor of this investigation is similar to that of [2], where solutions to the mutual-exclusion problem are studied for a system of processes that synchronize not with semaphore operations but with a generalized "test-and-set" operation on a single shared variable. In that study, bounds are obtained on the number of distinct values that this variable must be able to record, if solutions are to exist to the mutual-exclusion problem and to the starvation-free mutual-exclusion problem. The most important similarity between our study and that of [2] is that in both, results are stated and proved asserting the existence or nonexistence of solutions to the mutual-exclusion problem satisfying various properties. In both studies, existence results are proved by displaying a solution to the mutual-exclusion problem. Results asserting the nonexistence of solutions are proved indirectly by assuming the existence of a solution satisfying the stated properties and then inferring the existence of a computation that contradicts one or more assumptions.

In other related investigations [8, 14] weak semaphore operations are viewed as conceptual abstractions/reductions of blocked-set and blocked-queue operations. Given any program, its abstraction may be formed by "deleting the scheduler state"; that is, the queuing information maintained inside of blocked-set and blocked-queue semaphores is ignored. In [8] Doeppner shows that a program is deadlock free if and only if its abstraction is deadlock free. Kwong [14] identifies two types of starvation and shows essentially that abstraction increases the possibility of starvation. Kwong also observes that starvation is not preserved by Lipton's type of reduction [16].

1.4 OUTLINE OF PAPER. The remainder of the paper is organized as follows. If we are to prove results concerning the existence or nonexistence of solutions to the starvation-free mutual-exclusion problem, then we must precisely define the class of candidate solutions. These definitions are the subject of Section 2. This section culminates in the definition of a *mutual-exclusion system*, which models a number of concurrently executing sequential processes competing for access to critical regions. A *solution to the mutual-exclusion problem* is defined as a mutual-exclusion system with certain desirable properties.

In Sections 3–5 the basic definitions are used to prove results that indicate the differences in power between the various types of semaphore primitives. To highlight these differences, and to eliminate from consideration solutions to the mutual-exclusion problem that do not use semaphores at all, the properties of *no busy-waiting*, *symmetry*, and *no memory* are defined, and we restrict our attention to mutual-exclusion systems satisfying one or more of these properties. We will see that there are situations in which it is possible to solve the starvation-free mutual-exclusion problem with blocked-set semaphores but not with weak semaphores. We will also see that weak general semaphores are slightly more powerful than weak binary semaphores in a certain sense.

Finally, Section 6 summarizes the results proved and offers a suggestion for future investigation.

2. *Basic Definitions*

In this section we define the term *solution to the mutual-exclusion problem*. This definition consists of several stages. We first define a simple programming language $\mathscr{L}$, which contains an assignment statement, control statements for alternation and iteration, and **P** and **V** semaphore statements. The language $\mathscr{L}$ is what we will use to express solutions to the mutual-exclusion problem. The exact details of this language are not critical; the only important requirement is that a single assignment statement not both depend on and affect the same global variable. Intuitively, this requirement captures the idea that a shared memory location may not be both read and updated in a single indivisible step.

To be able to reason about programs in $\mathscr{L}$, we require a formal semantics for this language. For this reason, we define a model of concurrent computation called *parallel programs*. The parallel-program model is a state-transition model similar to that introduced in [11]. The general parallel-program model does not have sufficient structure for our purposes, and we therefore define a restricted subset of parallel programs, called *systems of processes*. A system of processes models a collection of concurrently executing sequential processes. The language $\mathscr{L}$ will be given a formal semantics by showing how a collection of N statements of $\mathscr{L}$ defines a system of N processes.

Finally, we define *mutual-exclusion systems* to be systems of processes that model a number of processes competing for access to critical regions. A *solution to the mutual-exclusion problem* will be a mutual-exclusion system that has certain desirable properties, including the *mutual-exclusion property* and *freedom from deadlock*. A solution to the *starvation-free* mutual-exclusion problem will be a solution to the mutual exclusion problem that is in addition *starvation free*.

For the remainder of this paper let G, L, and S be fixed finite sets of *global, local,* and *semaphore* variables, respectively. Let $\mathscr{V}$ be a domain of values for these variables which includes at least the set of natural numbers $\mathbf{N}$ and the set of all finite subsets of natural numbers.

2.1 A SIMPLE PROGRAMMING LANGUAGE. The language $\mathscr{L}$ is defined inductively as follows:

(1) **skip** is a statement in $\mathscr{L}$.
(2) If $s \in S$, then the **P** *operations* $\mathbf{P}_{\mathrm{wb}}(s)$, $\mathbf{P}_{\mathrm{wg}}(s)$, $\mathbf{P}_{\mathrm{bsb}}(s)$, and $\mathbf{P}_{\mathrm{bsg}}(s)$ and the **V** *operations* $\mathbf{V}_{\mathrm{wb}}(s)$, $\mathbf{V}_{\mathrm{wg}}(s)$, $\mathbf{V}_{\mathrm{bsb}}(s)$, and $\mathbf{V}_{\mathrm{bsg}}(s)$ are statements. The statements $\mathbf{P}_x(s)$ and $\mathbf{V}_x(s)$ are called *weak binary, weak general, blocked-set binary,* or *blocked-set general* semaphore operations, when x is wb, wg, bsb, or bsg, respectively.
(3) If $u_1, u_2, \ldots, u_m$, and $v_1, v_2, \ldots, v_n$ are elements of $G \cup L$, such that $i \neq j$ implies $u_i \neq u_j$ and $v_i \neq v_j$, and such that $\{u_1, u_2, \ldots, u_m\} \cap \{v_1, v_2, \ldots, v_n\}$ contains no elements of G, then the *assignment statement*

$$(u_1, u_2, \ldots, u_m) := F(v_1, v_2, \ldots, v_n)$$

is a statement, where $F(v_1, v_2, \ldots, v_n)$ is an expression over the variables $v_1, v_2, \ldots, v_n$ which defines a function from $\mathscr{V}^n$ to $\mathscr{V}^m$.
(4) If $v_1, v_2, \ldots, v_n$ are distinct elements of $G \cup L$ and $S_1, S_2 \in \mathscr{L}$, then the *conditional statement*

$$\textbf{if } B(v_1, v_2, \ldots, v_n) \textbf{ then } S_1 \textbf{ else } S_2$$

and the *loop statement*

$$\textbf{while } B(v_1, v_2, \ldots, v_n) \textbf{ do } S_1$$

are statements, where $B(v_1, v_2, \ldots, v_n)$ is an expression over $v_1, v_2, \ldots, v_n$ which defines an n-placed relation over $\mathscr{V}$.
(5) If $S_1, S_2, \ldots, S_m$ are statements, then the *compound statement*

$$\textbf{begin } S_1; S_2; \ldots; S_m \textbf{ end}$$

is a statement.
(6) The only statements in $\mathscr{L}$ are those that can be formed by a finite number of applications of (1)–(5).

If $E(v_1, v_2, \ldots, v_n)$ is an expression over the variables $v_1, v_2, \ldots, v_n$, then we will use the notation $E[x_1, x_2, \ldots, x_n]$ to denote the result obtained by evaluating $E(v_1, v_2, \ldots v_n)$ after substituting the values $x_1, x_2, \ldots, x_n$ for the variables $v_1, v_2, \ldots, v_n$.

2.2 PARALLEL PROGRAMS. A *parallel program* Γ is a four-tuple $\langle V, Q, T, q^{\mathrm{I}}\rangle$, where

(1) V is a finite set of *variables.*
(2) $Q = C \times D$ is the set of *states* for Γ, where C, the set of *control states,* is finite, and D, the set of *data states,* is the set of all functions from V to $\mathscr{V}$. If $q \in Q$, then we write $cnt(q)$ for the control state component of q and $dat(q)$ for the data state component.
(3) T is a finite set of *transitions.* Each $t \in T$ is a pair $\langle ena(t), nxt(t)\rangle$, where $ena(t)$ is a predicate on Q called the *enabling predicate* for t and $nxt(t)$ is a partial function from Q to Q called the *action function* for t. The function $nxt(t)$ is defined for a state q iff $(ena(t))(q)$ is true.
(4) The state q^{I} is a distinguished element of Q called the *initial state.*

If $(ena(t))(q)$ is true, then we say that t is *enabled* in state q. If $q \in Q$ and $v \in V$, then we abbreviate $(dat(q))(v)$ as $q(v)$. To avoid superfluous parentheses, we also abbreviate $(nxt(t))(q)$ as $nxt(q, t)$. If $\alpha = t_0t_1 \cdots$ is a finite or infinite sequence of transitions and $\sigma = q_0q_1 \cdots$ is a finite or infinite sequence of states, then we define $\alpha(i) = t_i$ and $\sigma(i) = q_i$. It is also convenient to define $\alpha(i, j) = t_i \cdots t_{j-1}$ if $i < j$, and $\sigma(i, j) = q_i \cdots q_j$ if $i \leq j$. If $j \leq i$, then $\alpha(i, j) = \Lambda$ (the null sequence), and if $j < i$, then $\sigma(i, j) = \Lambda$.

A finite or infinite sequence of transitions α is an *execution sequence from* q_0 *for* Γ, *with corresponding state sequence* σ, if $|\sigma| = |\alpha| + 1$, $\sigma(0) = q_0$, and for each i such that $0 \leq i < |\alpha|$, transition $\alpha(i)$ is enabled in state $\sigma(i)$ and $\sigma(i + 1) = nxt(\sigma(i), \alpha(i))$. If $\sigma(0) = q^I$, then α is an *initial* execution sequence for Γ. If $|\alpha| = n$, then we define $nxt(\sigma(0), \alpha) = \sigma(n)$, and we say that $\sigma(n)$ is *reachable from* $\sigma(0)$. A *reachable* state is simply a state that is reachable from q^I.

Keller [11] shows how parallel programs can be used to model various kinds of concurrent computation. The notion of control state is convenient for modeling the instruction counters of a number of concurrently executing sequential processes, as we shall now see.

2.3 SYSTEMS OF PROCESSES. A *system of N processes* is a parallel program $\Gamma = \langle V, Q, T, q^I \rangle$ with the following properties:

(1) If C is the set of control states for Γ, then $C = \prod_{i=1}^{N} C_i$. The set C_i is called the set of *control states for process i*. We write $cnt_i(q)$ for the projection of q into C_i.
(2) $T = \bigcup_{i=1}^{N} T_i$, where the T_i are disjoint sets of *transitions for process i*.
(3) For all $q, q' \in Q$, if $cnt_i(q) = cnt_i(q')$ and $dat(q) = dat(q')$, then for all $t \in T_i$,
    (a) $(ena(t))(q)$ is true iff $(ena(t))(q')$ is true;
    (b) $cnt_i(nxt(q, t)) = cnt_i(nxt(q', t))$ and $dat(nxt(q, t)) = dat(nxt(q', t))$;
    (c) $cnt_j(nxt(q, t)) = cnt_j(q)$ for all $j \neq i$ with $1 \leq j \leq N$.

Thus in a system of N processes the control state models the N instruction counters. The program being executed by process i is modeled by the set of transitions T_i. Condition (3) above stipulates that a transition for process i must neither depend on nor affect the instruction counter of some other process $j \neq i$.

We will denote a transition for process i in a system of processes by an expression of the form

$$\textbf{when } c_1 \textbf{ and } B(v_1, v_2, \ldots, v_n) \textbf{ do } (u_1, u_2, \ldots, u_m) := F(v_1, v_2, \ldots, v_n) \textbf{ then } c_2,$$

where $c_1, c_2 \in C_i$. Such an expression defines a transition t as follows:

(1) $(ena(t))(q)$ is true iff $cnt_i(q) = c_1$ and $B[q(v_1), q(v_2), \ldots, q(v_n)]$ is true.
(2) If $(ena(t))(q)$ is true, then the state $q' = nxt(q, t)$ is related to q by
    (a) $cnt_i(q') = c_2$;
    (b) $cnt_j(q') = cnt_j(q)$ for all $j \neq i$ with $1 \leq j \leq N$;
    (c) $q'(v) = q(v)$ for all $v \in V - \{u_1, u_2, \ldots, u_m\}$;
    (d) $q'(u_j) = F_j[q(v_1), \ldots, q(v_n)]$ for all j with $1 \leq j \leq m$, where F_j denotes the projection of F on the jth coordinate.

We say that a transition *depends* on a variable if that variable occurs among the v_i in its defining expression, and that a transition *affects* a variable if that variable occurs among the u_i in its defining expression. In addition, we will often abbreviate the defining expressions for transitions by omitting the **and** clause if B is identically true and the **do** clause if $m = 0$.

Let Γ be a system of N processes, and let α be an execution sequence for Γ, with corresponding state sequence σ. Then α has the *finite-delay property* unless it is infinite and there exists i with $1 \leq i \leq N$, and $k \geq 0$, such that

(1) for each $j \geq k$, some transition in T_i is enabled in state $\sigma(j)$;
(2) for no $j \geq k$ is $\alpha(j) \in T_i$.

If α has the finite-delay property, then we also say that α is *valid*. The finite-delay property characterizes those execution sequences in which no process runs infinitely more slowly than another. Another way of stating this is that valid execution sequences satisfy the requirement of "fair scheduling" of processes. The finite-delay property is mentioned in [11, 17] and used in definition of an "admissible schedule" in [2]. It appears to be necessary for any discussion of starvation-free synchronization.

2.4 SEMANTICS OF $\mathscr{L}$. In this section we give a formal semantics to the language $\mathscr{L}$ by showing how each statement A of $\mathscr{L}$ defines a set $\mathscr{C}_i(A)$ of control states and a set $\mathscr{T}_i(A)$ of transitions for process i in a system of processes. Before we can define these sets of states and transitions, however, we must determine an appropriate set of variables V for this system of processes. We cannot simply let $V = G \cup L \cup S$, since we have yet to capture the idea that each process should have its own copy of the local variables in L. Also, we will need extra variables to represent the sets of blocked processes associated with blocked-set semaphore variables. Therefore, let $L_i = L \times \{i\}$, and let $L^* = \bigcup_{i=1}^{N} L_i$. Let B_S be the set which contains, for each $s \in S$, a new *scheduler variable* b_s. All scheduler variables are *set* variables; that is, they will always have finite sets of natural numbers as their values. Let $V = G \cup L^* \cup S \cup B_S$.

For each statement $A \in \mathscr{L}$ define the set $\mathscr{C}_i(A)$ of control states of A for process i as follows. If A is not a blocked-set **P** operation, then $\mathscr{C}_i(A)$ is defined recursively by $\mathscr{C}_i(A) = \{before_i(A), after_i(A)\} \cup SUBST$, where $SUBST$ is the union of all $\mathscr{C}_i(A')$ with A' a proper substatement of A. If A is a blocked-set **P** operation, then $\mathscr{C}_i(A) = \{before_i(A), after_i(A), waiting_i(A)\}$. As their names suggest, the control states $before_i(A)$ and $after_i(A)$ represent the points in the code for process i just before and just after the statement A, respectively. Similarly, $waiting_i(A)$ identifies the control point within a blocked-set **P** operation where process i resides while it is blocked.

We can now define the set $\mathscr{T}_i(A)$ for each $A \in \mathscr{L}$ by induction. In the following, as in the remainder of the paper, we will abbreviate $\langle v, i\rangle \in L_i$ by v^i. In addition, it will be notationally convenient to define $v^i = v$ for $v \in V - L^*$. In cases (2b) and (2d) below, replace $INCR$ by $s := 1$ for a binary **V** operation and by $s := s + 1$ for a general **V** operation.

(1) If $A =$ **skip**, then

$$\mathscr{T}_i(A) = \{\textbf{when } before_i(A) \textbf{ then } after_i(A)\}.$$

(2a) If $A = \mathbf{P}_{\mathrm{wb}}(s)$ or $A = \mathbf{P}_{\mathrm{wg}}(s)$, then

$$\mathscr{T}_i(A) = \{\textbf{when } before_i(A) \textbf{ and } s > 0 \textbf{ do } s := s - 1 \textbf{ then } after_i(A)\}.$$

(2b) If $A = \mathbf{V}_{\mathrm{wb}}(s)$ or $A = \mathbf{V}_{\mathrm{wg}}(s)$, then

$$\mathscr{T}_i(A) = \{\textbf{when } before_i(A) \textbf{ do } INCR \textbf{ then } after_i(A)\}.$$

(2c) If $A = \mathbf{P}_{\mathrm{bsb}}(s)$ or $A = \mathbf{P}_{\mathrm{bsg}}(s)$, then

$$\begin{aligned}\mathscr{T}_i(A) = \{&\textbf{when } before_i(A) \textbf{ and } s > 0 \textbf{ do } s := s - 1 \textbf{ then } after_i(A),\\ &\textbf{when } before_i(A) \textbf{ and } s = 0 \textbf{ do } b_s := b_s \cup \{i\} \textbf{ then } waiting_i(A),\\ &\textbf{when } waiting_i(A) \textbf{ and } i \notin b_s \textbf{ then } after_i(A)\}.\end{aligned}$$

(2d) If $A = \mathbf{V}_{\mathrm{bsb}}(s)$ or $A = \mathbf{V}_{\mathrm{bsg}}(s)$, then

$$\mathcal{T}_i(A) = \bigcup_{j=1}^{N} \{\textbf{when } before_i(A) \textbf{ and } j \in b_s \textbf{ do } b_s := b_s - \{j\} \textbf{ then } after_i(A)\} \cup \{\textbf{when } before_i(A) \textbf{ and } b_s = \varnothing \textbf{ do } INCR \textbf{ then } after_i(A)\}.$$

(3) If $A = (u_1, u_2, \ldots, u_m) := F(v_1, v_2, \ldots, v_n)$, then

$$\mathcal{T}_i(A) = \{\textbf{when } before_i(A) \textbf{ do } (u_1^i, u_2^i, \ldots, u_m^i) := F(v_1^i, v_2^i, \ldots, v_n^i) \textbf{ then } after_i(A)\}.$$

(4a) If $A = \textbf{if } B(v_1, v_2, \ldots, v_n) \textbf{ then } S_1 \textbf{ else } S_2$, then

$$\begin{aligned}\mathcal{T}_i(A) = \{&\textbf{when } before_i(A) \textbf{ and } B(v_1^i, v_2^i, \ldots, v_n^i) \textbf{ then } before_i(S_1),\\ &\textbf{when } before_i(A) \textbf{ and } \neg B(v_1^i, v_2^i, \ldots, v_n^i) \textbf{ then } before_i(S_2),\\ &\textbf{when } after_i(S_1) \textbf{ then } after_i(A),\\ &\textbf{when } after_i(S_2) \textbf{ then } after_i(A)\} \cup \mathcal{T}_i(S_1) \cup \mathcal{T}_i(S_2).\end{aligned}$$

(4b) If $A = \textbf{while } B(v_1, v_2, \ldots, v_n) \textbf{ do } S_1$, then

$$\begin{aligned}\mathcal{T}_i(A) = \{&\textbf{when } before_i(A) \textbf{ and } B(v_1^i, v_2^i, \ldots, v_n^i) \textbf{ then } before_i(S_1),\\ &\textbf{when } before_i(A) \textbf{ and } \neg B(v_1^i, v_2^i, \ldots, v_n^i) \textbf{ then } after_i(A),\\ &\textbf{when } after_i(S_1) \textbf{ then } before_i(A)\} \cup \mathcal{T}_i(S_1).\end{aligned}$$

(5) If $A = \textbf{begin } S_1; S_2; \ldots; S_m \textbf{ end}$, then

$$\begin{aligned}\mathcal{T}_i(A) = \{&\textbf{when } before_i(A) \textbf{ then } before_i(S_1),\\ &\textbf{when } after_i(S_1) \textbf{ then } before_i(S_2), \ldots,\\ &\textbf{when } after_i(S_{m-1}) \textbf{ then } before_i(S_m),\\ &\textbf{when } after_i(S_m) \textbf{ then } after_i(A)\}\\ &\cup \mathcal{T}_i(S_1) \cup \mathcal{T}_i(S_2) \cup \cdots \cup \mathcal{T}_i(S_m).\end{aligned}$$

Note how case (2) above makes precise the informal definitions of the various types of semaphores presented earlier. Weak semaphore operations consist of a single transition. A process cannot execute a weak $\mathbf{P}(s)$ operation until a state is reached in which the semaphore variable s has a positive value. When this occurs, s is decremented and the process proceeds. A weak general $\mathbf{V}(s)$ causes s to be incremented, and a weak binary $\mathbf{V}(s)$ operation simply causes s to be set to one.

The operation of blocked-set semaphore operations is somewhat more complicated. Associated with each semaphore variable is a *scheduler variable* b_s. In any given state the value of b_s represents the set of processes blocked on the semaphore variable s in that state. A process executing a blocked-set $\mathbf{P}(s)$ operation first checks to see if s has a positive value. If so, then s is decremented and the process proceeds. If the value of s is found to be zero, then the process enters its index in the set b_s of blocked processes and then waits for some process executing a $\mathbf{V}(s)$ operation to signal it by removing its index from b_s. A process executing a blocked-set $\mathbf{V}(s)$ operation either determines that b_s is empty, in which case s is incremented (set to one, in the case of a blocked-set binary $\mathbf{V}(s)$), or makes a nondeterministic selection of one of the indices in b_s which it removes, thus signaling the corresponding waiting process. This nondeterministic selection is modeled by the set of transitions $\bigcup_{j=1}^{N} \{\textbf{when } before_i(A)$ $\textbf{and } j \in b_s \textbf{ do } b_s := b_s - \{j\} \textbf{ then } after_i(A)\}$.

We will say that a transition t is *part of a semaphore operation* if $t \in \mathcal{T}_i(A)$, where A is a $\mathbf{P}$ or $\mathbf{V}$ operation. Note that since $\mathcal{T}_i(A)$ is a singleton set if A is a weak $\mathbf{P}$ or $\mathbf{V}$ operation, we may, without ambiguity, say that t *is* a $\mathbf{P}$ or $\mathbf{V}$ operation if t is the element of such a singleton set.

2.5 MUTUAL-EXCLUSION SYSTEMS. We wish to model a system in which each process continually alternates execution between a *critical region* and a *noncritical* *region.* A process leaving its noncritical region and attempting to enter its critical region must first execute the synchronization protocol in the *trying region.* Similarly, a process leaving the critical region and returning to the noncritical region must execute the synchronization protocol in the *leaving region.* The fact that a process may remain in the noncritical region forever is modeled by allowing a process in the noncritical region to become *halted* rather than executing the trying region protocol.

More precisely, suppose $N \geq 1$, and for each i with $1 \leq i \leq N$ let Tr_i and Lv_i be given statements of $\mathscr{S}$ called the *trying region* and *leaving region*, respectively, for process i. Define

$$\begin{aligned} C_i &= \mathscr{C}_i(\mathrm{Tr}_i) \cup \mathscr{C}_i(\mathrm{Lv}_i) \cup \{\mathit{critical}_i, \mathit{noncritical}_i, \mathit{halted}_i\}, \\ T_i &= \mathscr{T}_i(\mathrm{Tr}_i) \cup \mathscr{T}_i(\mathrm{Lv}_i) \cup \{\textbf{when } \mathit{noncritical}_i \textbf{ then } \mathit{before}_i(\mathrm{Tr}_i), \\ &\qquad \textbf{when } \mathit{noncritical}_i \textbf{ then } \mathit{halted}_i, \\ &\qquad \textbf{when } \mathit{after}_i(\mathrm{Tr}_i) \textbf{ then } \mathit{critical}_i, \\ &\qquad \textbf{when } \mathit{critical}_i \textbf{ then } \mathit{before}_i(\mathrm{Lv}_i), \\ &\qquad \textbf{when } \mathit{after}_i(\mathrm{Lv}_i) \textbf{ then } \mathit{noncritical}_i\}. \end{aligned}$$

Let $C = \prod_{i=1}^{N} C_i$ and $T = \bigcup_{i=1}^{N} T_i$, let D be the set of all functions from V to $\mathscr{V}$, and let $Q = C \times D$, and suppose $q^{\mathrm{I}} \in Q$ is given such that $\mathit{cnt}_i(q^{\mathrm{I}}) = \mathit{noncritical}_i$, for $1 \leq i \leq N$, and $q^{\mathrm{I}}(b_s) = \varnothing$ for all scheduler variables b_s. Then the parallel program $\Gamma = \langle V, Q, T, q^{\mathrm{I}} \rangle$ is a *mutual-exclusion system of N processes.*

Suppose $q \in Q$. Then we say that process i is *in the noncritical region* (respectively, *in the critical region, halted*) in state q if $\mathit{cnt}_i(q) = \mathit{noncritical}_i$ (respectively, $\mathit{critical}_i$, halted_i). Process i is *in the trying region* (respectively, *in the leaving region*) in state q if $\mathit{cnt}_i(q) \in \mathscr{C}_i(\mathrm{Tr}_i)$ (respectively, $\mathscr{C}_i(\mathrm{Lv}_i)$). The transition **when** $\mathit{noncritical}_i$ **then** $\mathit{before}_i(\mathrm{Tr}_i)$ (respectively, **when** $\mathit{critical}_i$ **then** $\mathit{before}_i(\mathrm{Lv}_i)$, **when** $\mathit{noncritical}_i$ **then** halted_i) is called the *trying transition* (respectively, *leaving transition, halting transition*) for process i.

A *mutual-exclusion system with weak binary semaphores* is a mutual-exclusion system such that the only semaphore operations in the trying and leaving regions are weak binary semaphore operations. Mutual-exclusion systems with weak general semaphores, blocked-set binary semaphores, and blocked-set general semaphores are defined similarly.

2.6 Solutions to the Mutual-Exclusion Problem.

A mutual-exclusion system has the *mutual-exclusion property* if there is no reachable state q such that more than one process is in the critical region in state q.

Process i in a mutual-exclusion system is *deadlocked* in a state q if process i is in the trying or leaving region in state q and there is no finite execution sequence α from q consisting only of transitions for processes in the trying region, critical region, or leaving region, such that process i is not in the trying or leaving region in state $\mathit{nxt}(q, \alpha)$. A mutual-exclusion system is *deadlock free* if no process is deadlocked in any reachable state.

A mutual-exclusion system is *free from indefinite postponement* if there is no valid infinite initial execution sequence, with corresponding state sequence σ, such that for some k and all $j \geq k$ no process is in the critical region in state $\sigma(j)$. The requirement of freedom from indefinite postponement implies Dijkstra's requirement on a solution to the mutual-exclusion problem that, "If two processes are about to enter their critical regions, it must be impossible to devise for them such finite speeds, that the decision which one of the two is the first to enter its critical region is postponed to eternity."

A mutual-exclusion system is *starvation free* if there is no valid infinite initial execution sequence, with corresponding state sequence σ, having the following properties:

(1) There is a process i such that for all $k \geq 0$, there exist $m \geq k$ and $n \geq k$ with process i in the critical region in state $\sigma(m)$ and not in the critical region in state $\sigma(n)$.
(2) There is a process i' and a $k' \geq 0$, such that for all $j \geq k'$, process i' is in the trying or leaving region in state $\sigma(j)$.

A *solution to the mutual-exclusion problem for N processes* is a mutual-exclusion system of N processes that has the mutual-exclusion property, is deadlock free, and is free from indefinite postponement. A *solution to the starvation-free mutual-exclusion problem* is a solution to the mutual-exclusion problem that is starvation free.

3. *Busy-Waiting and Symmetry*

With the basic definitions out of the way we may now proceed with our investigation. The first question it is reasonable to ask is whether there are any solutions at all to the starvation-free mutual-exclusion problem. We assume implicitly that $N \geq 2$, since it is trivial to solve the problem for $N = 1$ process. We immediately see that the answer to this question is "yes," as is illustrated by any of a number of solutions to be found in the literature, for example, Knuth's solution in [12]. Moreover, Knuth's solution illustrates the fact that it is not even necessary to use semaphores to solve the problem. Thus we will not be able to compare the power of semaphore primitives on the basis of their ability to implement starvation-free mutual exclusion unless we impose restrictions sufficient to eliminate semaphore-free solutions from consideration.

Knuth's solution has two properties that might be viewed as undesirable if found in a solution using semaphores. We will term these properties *busy-waiting* and *asymmetry*. Busy-waiting means that processor time is used for waiting, rather than for more useful computation. One reason semaphores were introduced was to allow the mutual-exclusion problem to be solved without busy-waiting. Knuth's solution is asymmetric in the sense that processes do not all execute identical program text; rather, the program to be executed by process i depends explicitly on the process number i. In contrast, few solutions in the literature that use semaphores are asymmetric. Thus it seems reasonable to restrict our attention solely to symmetric mutual-exclusion systems, and to mutual-exclusion systems with no busy-waiting, in an attempt to eliminate semaphore-free solutions. In this section we precisely define these notions and show that indeed there are no semaphore-free symmetric solutions to the mutual-exclusion problem and no semaphore-free solutions without busy-waiting.

We first prove that any solution to the mutual-exclusion problem has a particular execution sequence in which all processes other than processes 1 and 2 become halted, and process 1 executes infinitely many critical and noncritical regions while process 2 remains in the noncritical region. This execution sequence, though not valid, will be useful for constructing various execution sequences that are valid.

LEMMA 3.1. *Let Γ be a solution to the mutual-exclusion problem. If q_0 is a reachable state such that process 1 is in the noncritical (respectively, critical) region in state q_0 and all other processes are either halted or in the noncritical region in state q_0,* *then there is a finite execution sequence α for process 1 from q_0 such that process 1 is in the critical (respectively, noncritical) region in state $nxt(q, \alpha)$.*

PROOF. An execution sequence for process 1 is an execution sequence consisting only of transitions in T_1. Let t_0 be the trying (respectively, leaving) transition for process 1. Then process 1 is in the trying (respectively, leaving) region in state $q_1 = nxt(q_0, t_0)$, and all other processes are either halted or in the noncritical region in state q_1. Since Γ is deadlock free, there is an execution sequence $t_1 \cdots t_{n-1}$ for process 1 such that process 1 is in the critical (respectively, noncritical) region in state $q_n = nxt(q_1, t_1 \cdots t_{n-1})$. Let $\alpha = t_0 t_1 \cdots t_{n-1}$. □

LEMMA 3.2. *Let Γ be a solution to the mutual-exclusion problem for $N \geq 2$ processes. Then there is an infinite initial execution sequence α for Γ with corresponding state sequence σ, along with indices ncr_i and cr_i for each $i \geq 1$, such that*

(1) *Processes 1 and 2 are in the noncritical region in state $\sigma(ncr_1)$.*
(2) *For all variables v, $(\sigma(ncr_1))(v) = q^I(v)$.*
(3) *For all $j \geq ncr_1$ and all k with $3 \leq k \leq N$, process k is halted in state $\sigma(j)$.*
(4) *For all $j \geq ncr_1$, $\alpha(j) \in T_1$.*
(5) *For all $i \geq 1$, cr_i is the least $j > ncr_i$ such that process 1 is in the critical region in state $\sigma(j)$, and ncr_{i+1} is the least $j > cr_i$ such that process 1 is in the noncritical region in state $\sigma(j)$.*

PROOF. We inductively construct an increasing sequence $\alpha_1, \alpha_2, \ldots$ of finite initial execution sequences, along with the indices ncr_i and cr_i. The sequence α will be the unique infinite initial execution sequence with all of the α_i as prefixes. It will follow from the construction that α has properties (1)–(5).

Basis. Define $\alpha_1 = t_0 t_1 \cdots t_{N-3}$, where t_i is the halting transition for process $i + 3$ for each i with $0 \leq i \leq N - 3$. Clearly α_1 is an initial execution sequence for Γ, and if we define $\mathrm{ncr}_1 = N - 2$ and $q_{\mathrm{ncr}_1} = nxt(q^I, \alpha_1)$, then processes 1 and 2 are in the noncritical region in state q_{ncr_1}, process k is halted in state q_{ncr_1} for all k with $3 \leq k \leq N$, and $q_{\mathrm{ncr}_1}(v) = q^I(v)$ for all variables v.

Induction step. Suppose, for some $i \geq 1$, that α_i is an initial execution sequence for Γ such that if $q_{\mathrm{ncr}_i} = nxt(q_1, \alpha_i)$, then processes 1 and 2 are in the noncritical region in state q_{ncr_i} and process k is halted in state q_{ncr_i} for all k with $3 \leq k \leq N$. By Lemma 3.1 there is an execution sequence β_i of length m_i, consisting only of transitions for process 1, such that if $\mathrm{cr}_i = \mathrm{ncr}_i + m_i$ and $q_{\mathrm{cr}_i} = nxt(q_{\mathrm{ncr}_i}, \beta_i)$, then process 1 is in the critical region in state q_{cr_i}. Similarly, there is an execution sequence γ_i of length n_i, consisting only of transitions for process 1, such that if $\mathrm{ncr}_{i+1} = \mathrm{cr}_i + n_i$ and $q_{\mathrm{ncr}_{i+1}} = nxt(q_{\mathrm{cr}_i}, \gamma_i)$, then process 1 is in the noncritical region in state $q_{\mathrm{ncr}_{i+1}}$. We may assume, without loss of generality, that β_i and γ_i are the shortest sequences with this property. Define $\alpha_{i+1} = \alpha_i\beta_i\gamma_i$. Then $q_{\mathrm{ncr}_{i+1}} = nxt(q_1, \alpha_{i+1})$, and process 1 is in the noncritical region in state $q_{\mathrm{ncr}_{i+1}}$. Since β_i and γ_i contain only transitions for process 1, it is clear that process 2 is in the noncritical region in state $q_{\mathrm{ncr}_{i+1}}$, and process k is halted in state $q_{\mathrm{ncr}_{i+1}}$ for all k with $3 \leq k \leq N$. □

3.1 BUSY-WAITING. A mutual exclusion system *has busy-waiting* if for every $M \geq 0$ there is a process i, indices m and n, and an initial execution sequence α, with corresponding state sequence σ, such that

(1) For all j with $m \leq j \leq n$, process i is in the trying or leaving region in state $\sigma(j)$.
(2) The number of $\alpha(m), \alpha(m + 1), \ldots, \alpha(n)$ that are transitions for process i is at least M.

This definition captures the notion that the essence of busy-waiting is that no a priori bound may be placed on the number of transitions a process may execute in the trying or leaving region. Let NBW denote the class of all mutual exclusion systems with *no busy-waiting.*

THEOREM 3.1. *Every semaphore-free solution to the mutual-exclusion problem has busy-waiting.*

PROOF. Let Γ be a semaphore-free solution to the mutual-exclusion problem. Let α be the execution sequence for Γ constructed in Lemma 3.2, and let σ be the corresponding state sequence. Clearly the subsequence $\alpha(0, \mathrm{cr}_1)$ of α is an initial execution sequence for Γ such that process 1 is in the critical region in state $\sigma(\mathrm{cr}_1)$. It is easily seen by the definition of a mutual-exclusion system and of the semantic mapping $\mathscr{T}_i$ that since Γ is semaphore free, process 2 has an enabled transition in any reachable state in which it is not halted. In addition, whenever process 2 is in the noncritical region, it has the option of executing the trying transition, rather than the halting transition. Thus, corresponding to each $n \geq 0$ is a finite execution sequence β_n of length n from state $\sigma(\mathrm{ncr}_1)$ which contains only transitions for process 2. By the definition of a mutual exclusion system, and because Γ has the mutual-exclusion property, process 2 must be in the trying region in state $nxt(\sigma(\mathrm{cr}_1), \beta_n(0, j))$ for all j with $0 < j \leq n$. Since n may be chosen arbitrarily large, Γ has busy-waiting. □

3.2 SIMILAR PROCESSES AND SYMMETRY. We say that processes i and j in a mutual exclusion system are *similar* if $\mathrm{Tr}_i = \mathrm{Tr}_j$ and $\mathrm{Lv}_i = \mathrm{Lv}_j$. Note that if i and j are similar processes, then there is a natural bijection φ_{ij} between the sets of control states C_i and C_j (φ_{ij}:*noncritical*$_i \mapsto$ *noncritical*$_j$, φ_{ij}:*before*$_i(A) \mapsto$ *before*$_j(A)$, etc.). In addition, it is easily proved by structural induction on $\mathscr{L}$ that there is a bijection ψ_{ij} between the sets of transitions T_i and T_j, where if $t^i \in T_i$ is not part of a semaphore operation, and

$$t^i = \textbf{when } c_1 \textbf{ and } B(v_1^i, \ldots, v_n^i) \textbf{ do } (u_1^i, \ldots, u_m^i) := F(v_1^i, \ldots, v_n^i) \textbf{ then } c_2,$$

then

$$t^j = \psi_{ij}(t^i) = \textbf{when } \varphi_{ij}(c_1) \textbf{ and } B(v_1^j, \ldots, v_n^j)$$
$$\textbf{do } (u_1^j, \ldots, u_m^j) := F(v_1^j, \ldots, v_n^j) \textbf{ then } \varphi_{ij}(c_2).$$

It will be convenient to use superscripts to indicate correspondence under ψ_{ij}; thus in the sequel t^i and t^j will always denote corresponding transitions in the similar processes i and j.

Let i and j be similar processes in a mutual-exclusion system Γ, and let q and q' be states of Γ. We say that *q looks to process i as q' looks to process j* (in symbols, $q \;{}_i{=}_j\; q'$) if

(1) $\varphi_{ij}(cnt_i(q)) = cnt_j(q')$,
(2) $q(v^i) = q'(v^j)$ for all $v^i \in G \cup L_i$.

If $q \;{}_i{=}_j\; q$, then we say that state *q looks alike to processes i and j.* If $q \;{}_i{=}_i\; q'$, then we say *q looks like q' to process i.* Note that

(1) $q \;{}_i{=}_i\; q$,
(2) $q \;{}_i{=}_j\; q'$ iff $q' \;{}_j{=}_i\; q$,
(3) if $q \;{}_i{=}_j\; q'$ and $q' \;{}_j{=}_k\; q''$, then $q \;{}_i{=}_k\; q''$.

LEMMA 3.3. *Let i and j be similar processes in a mutual-exclusion system Γ. Let t^i be a transition for process i, not part of a semaphore operation, and let q and q' be* *states such that $q \,{}_i{=}_j\, q'$. If t^i is enabled in state q, then t^j is enabled in state q', and $nxt(q, t^i) \,{}_i{=}_j\, nxt(q', t^j)$.*

PROOF. The lemma follows directly from the definition of ψ_{ij}. □

A mutual exclusion system is *symmetric* if for every two processes i and j,

(1) processes i and j are similar, and
(2) $q^I \,{}_i{=}_j\, q^I$.

Let SYM denote the class of symmetric mutual-exclusion systems. We have one important technique for proving results about symmetric mutual-exclusion systems. This technique, embodied in Lemma 3.5 below, states that if a state q_0 looks alike to the similar processes 1 and 2, and process 1 can execute some sequence of transitions α from state q_0, then as long as α contains no transitions that are part of semaphore operations, processes 1 and 2 can execute in "lock-step" from q_0. This means that processes 1 and 2 alternate execution of corresponding transitions, with the result that neither process 1 nor 2 is aware of the other's existence. Later we will generalize this result to apply to sequences α containing semaphore operations.

If processes i and j in a mutual exclusion system are similar, and if $\alpha = t_0^i t_1^i \cdots$ is a sequence of transitions in T_i, then let $alt_{ij}(\alpha, m, n)$ denote the sequence $t_m^i t_m^j t_{m+1}^i t_{m+1}^j \cdots t_{n-1}^i t_{n-1}^j$, if $n > m$, or Λ, if $n \leq m$. We will omit the subscripts and write $alt(\alpha, m, n)$ when i and j are clear from the context.

LEMMA 3.4. *Let processes 1 and 2 be similar processes in a mutual-exclusion system, and let q be a state such that $q \,{}_1{=}_2\, q$; suppose transition $t^1 \in T_1$, not part of a semaphore operation, is enabled in state q, and suppose $q' = nxt(q, t^1)$. If the transition $t^2 \in T_2$ corresponding to t^1 depends on no variables affected by t^1, then t^2 is enabled in state q', and if $q'' = nxt(q', t^2)$, then $q'' \,{}_1{=}_2\, q''$ and $q'' \,{}_1{=}_1\, q'$.*

PROOF. The essence of the proof is that process 1 sees the same values of variables in state q as process 2 sees in state q'. Because of this, even though t^2 may overwrite global variables that were affected by t^1, the values written by t^2 are the same as those written by t^1.

More precisely, suppose

$$t^1 = \textbf{when } c_1 \textbf{ and } B(v_1^1, \ldots, v_n^1) \textbf{ do } (u_1^1, \ldots, u_m^1) := F(v_1^1, \ldots, v_n^1) \textbf{ then } c_2.$$

Then

$$t^2 = \textbf{when } \varphi_{12}(c_1) \textbf{ and } B(v_1^2, \ldots, v_n^2) \textbf{ do } (u_1^2, \ldots, u_m^2) := F(v_1^2, \ldots, v_n^2) \textbf{ then } \varphi_{12}(c_2).$$

Since $cnt_1(q) = c_1$, $q \,{}_1{=}_2\, q$, and executing t^1 cannot change the control state of process 2, we have that $cnt_2(q') = \varphi_{12}(c_1)$. Since t^2 depends on no variables affected by t^1, we know that $q'(v_i^2) = q(v_i^2) = q(v_i^1)$ for $1 \leq i \leq n$. Since t^1 is enabled in state q, we have that $B[q(v_1^1), \ldots, q(v_n^1)]$ is true, hence $B[q'(v_1^2), \ldots, q'(v_n^2)]$ is true, and therefore t^2 is enabled in state q'. Let $q'' = nxt(q', t^2)$. Note that $cnt_2(q'') = \varphi_{12}(c_2)$, $cnt_1(q') = c_2$, and hence $cnt_1(q'') = c_2$, since execution of t^2 cannot change the control state for process 1.

To show that $q'' \,{}_1{=}_2\, q''$ and $q'' \,{}_1{=}_1\, q'$, we must show that if $w \in G$, then $q''(w) = q'(w)$, and if $w^1 \in L_1$, then $q''(w^1) = q'(w^1)$ and $q''(w^1) = q''(w^2)$. Suppose $w \in G$. If w is not affected by t^2, then $q''(w) = q'(w)$. If w is affected by t^2, then $w = u_i^2$ for some i. But then $q''(w) = F_i[q'(v_1^2), \ldots, q'(v_n^2)] = F_i[q(v_1^1), \ldots, q(v_n^1)] = q'(w)$.

Finally, suppose $w^1 \in L_1$. Then $q''(w^1) = q'(w^1)$, since t^2 cannot affect variables in L_1. If w^1 is not affected by t^1, then w^2 is not affected by t^2, and hence $q''(w^1) = q'(w^1) = q(w^1) = q(w^2) = q'(w^2) = q''(w^2)$. If w^1 is affected by t^1, then $w^1 = u_i^1$ for some i.

In this case, $q''(w^1) = q'(w^1) = F_i[q(v_1^1), \ldots, q(v_n^1)] = F_i[q'(v_1^2), \ldots, q'(v_n^2)] = q''(w^2)$. □

LEMMA 3.5 (LOCK-STEP LEMMA). *Suppose processes* 1 *and* 2 *are similar processes in a mutual-exclusion system, q_0 is a state such that $q_0 \; {}_1{=}_2 \; q_0$, and α is an execution sequence of length m for process* 1 *from q_0. If α contains no transitions that are part of semaphore operations, then $\beta = alt(\alpha, 0, m)$ is also an execution sequence from q_0. In addition, if $q_m = nxt(q_0, \alpha)$ and $q'_m = nxt(q_0, \beta)$, then $q'_m \; {}_1{=}_2 \; q'_m$, $q'_m \; {}_1{=}_1 \; q_m$, and $q'_m(s) = q_m(s)$ for all semaphore variables s.*

PROOF. The proof is by induction on m.

Basis. If $m = 0$, then $\alpha = \Lambda$, and the lemma holds trivially.

Induction step. Suppose the lemma holds for sequences of length $m \geq 0$. We shall show that the lemma holds for sequences α of length $m + 1$. Suppose $\alpha = t_0^1 t_1^1 \cdots t_m^1$, and let $\sigma = q_0 q_1 \cdots q_{m+1}$ be the corresponding state sequence. Application of the lemma to the subsequence $\alpha(0, m)$ of length m shows that $alt(\alpha, 0, m)$ is an execution sequence from q_0. If $q'_m = nxt(q_0, alt(\alpha, 0, m))$, then $q'_m \; {}_1{=}_2 \; q'_m$, $q'_m \; {}_1{=}_1 \; q_m$, and $q'_m(s) = q_m(s)$ for all semaphore variables s. Since $q'_m \; {}_1{=}_1 \; q_m$ and t_m^1 is enabled in state q_m, t_m^1 is enabled in state q'_m by Lemma 3.3. Let $q''_m = nxt(q'_m, t_m^1)$. Since t_m^1 (and hence t_m^2) is not part of a semaphore operation, t_m^2 depends on no variables affected by t_m^1. Hence Lemma 3.4 implies that t_m^2 is enabled in state q''_m, and if $q'_{m+1} = nxt(q''_m, t_m^2)$, then $q'_{m+1} \; {}_1{=}_2 \; q'_{m+1}$ and $q'_{m+1} \; {}_1{=}_1 \; q_{m+1}$. Since neither t_m^1 nor t_m^2 can affect a semaphore variable, we have that $q'_{m+1}(s) = q_{m+1}(s)$ for all semaphore variables s. □

THEOREM 3.2. *Every solution to the mutual-exclusion problem in SYM uses semaphores.*

PROOF. Suppose Γ is a solution to the mutual-exclusion problem that is in SYM but uses no semaphores. Let α be the execution sequence for Γ constructed in Lemma 3.2 Then $\sigma(\mathrm{ncr}_1) \; {}_1{=}_2 \; \sigma(\mathrm{ncr}_1)$, and $\alpha(\mathrm{ncr}_1, \mathrm{cr}_1)$ is an execution sequence for process 1 from $\sigma(\mathrm{ncr}_1)$ which contains no transitions that are part of semaphore operations. Application of Lemma 3.5 shows that $\beta = alt(\alpha, \mathrm{ncr}_1, \mathrm{cr}_1)$ is also an execution sequence from $\sigma(\mathrm{ncr}_1)$ for Γ, and if $q = nxt(\sigma(\mathrm{ncr}_1), \beta)$, then $q \; {}_2{=}_1 \; q$ and $q \; {}_1{=}_1 \; \sigma(\mathrm{cr}_1)$. But this means that processes 1 and 2 are both in the critical region in state q. Since $q = nxt(q^I, \alpha(0, ncr_1)\beta)$, we have a contradiction of the assumption that Γ has the mutual-exclusion property. □

4. *Weak and Blocked-Set Semaphore Solutions in NBW and SYM*

Now that we have shown that there are no semaphore-free solutions to the mutual-exclusion problem in either NBW or SYM, it is reasonable to ask whether there are any weak or blocked-set semaphore solutions in these classes. The answer to these questions is "yes," as will now be shown. In this paper, existence results are argued by displaying a purported solution to the starvation-free mutual-exclusion problem in the desired class. They are stated as examples rather than theorems, since rigorous proofs of correctness are not supplied.

Example 4.1. There exists a solution to the starvation-free mutual-exclusion problem in NBW $\cap$ SYM with either blocked-set binary or blocked-set general semaphores. □

Such a solution, due to Morris [18], is displayed in Figure 2. This solution is surprisingly difficult to understand, and no attempt to explain its operation will be

```
$G$ = {count1, count2} initially {0, 0}
$L$ = {local} initially {0}
$S$ = {a, b, m} initially {1, 1, 0}

(Trying region $Tr_i$)
begin
    $P_{bsb}(b)$;
    local := count1;
    count1 := local + 1;
    $V_{bsb}(b)$;
    $P_{bsb}(a)$;
    $P_{bsb}(b)$;
    local := count1;
    count1 := local − 1;
    local := count2;
    count2 := local + 1;
    if count1 > 0 then begin
        $V_{bsb}(b)$;
        $V_{bsb}(a)$
    end else begin
        $V_{bsb}(b)$;
        $V_{bsb}(m)$
    end;
    $P_{bsb}(m)$
end

(Leaving region $Lv_i$)
begin
    local := count2;
    count2 := local − 1;
    if count2 > 0 then $V_{bsb}(m)$
    else $V_{bsb}(a)$;
    local := 0
end
```

FIG. 2. Morris' solution to starvation-free mutual exclusion.

made here. The reader who is interested in a detailed discussion of this solution should refer to the explanation and informal correctness argument given in [18] or to the more formal correctness proof of [1], which uses temporal logic techniques. It is important to note that the freedom from starvation of this solution depends crucially on its use of blocked-set semaphores rather than weak semaphores.

Example 4.2. There exists a solution to the starvation-free mutual-exclusion problem in NBW ∩ SYM with either weak binary or weak general semaphores. □

Figure 3 displays a weak semaphore solution in NBW − SYM, and Figure 4 shows how to use this solution to construct a solution in NBW ∩ SYM through the addition of a new semaphore variable. The lines preceded by asterisks in Figure 3 are not in the language $\mathcal{L}$; however it is a straightforward but tedious process to rewrite these lines as statements of $\mathcal{L}$.

The solution of Figure 3 uses the collection of variables $flag_i$, along with the roving pointer *next* to implement a fair, but not FIFO, queuing discipline among the N processes. When process i wishes to execute in its critical region, it first indicates this to the other processes by setting $flag_i$ to one. It then checks the variable *empty* to see if any process is currently executing in its critical region. If not, then process i proceeds into its critical region. If there is another process in its critical region, then process i waits on its private semaphore variable s_i. When a process leaves its critical region, it examines the variables $flag_j$ to select the next process to execute in the critical region. Although weak semaphore operations on the variable *mutex*1 are used to obtain mutually exclusive access to the variable *empty*, no starvation is possible, since the use of *next* ensures that no process may overtake a waiting process more than once.

In Figure 4, the trying regions Tr_i and Lv_i of Figure 3 are used in the construction of new symmetric trying regions Tr_i' and Lv_i'. The first time through the trying

```
$G$ = {$next, empty, flag_1, flag_2, . . , flag_N$} initially {1, 1, 0, 0, . . . , 0}
$L$ = {$first$} initially {0}
$S$ = {$mutex1, s_1, s_2, . . . , s_N$} initially {1, 0, 0, . . . , 0}

(Trying region $Tr_i$)                  (Leaving region $Lv_i$)
begin                                   begin
    $flag_i$ := 1;                              $flag_i$ := 0,
    $first$ := 0;                               $P_{wb}(mutex1)$;
    $P_{wb}(mutex1)$;                   *       for $j$ := $next$ + 1 to $N$ step 1, 1 to $next$ step 1 do
    if $empty \neq 0$ then begin        *           if $flag_j$ = 1 then begin
        $empty$ := 0;                   *               $next$ := $j$;
        $first$ := 1                    *               $V_{wb}(s_j)$;
    end else skip,                      *               goto $out$
    $V_{wb}(mutex1)$;                   *           end else skip;
    if $first$ = 0 then $P_{wb}(s_i)$ else skip   *   $empty$ := 1;
end                                     *$out$:     $V_{wb}(mutex1)$
                                        end
```

FIG. 3. Weak semaphore solution in NBW − SYM.

```
$G' = G \cup$ {$procno$} initially {1}
$L' = L \cup$ {$ident$} initially {0}
$S' = S \cup$ {$mutex2$} initially {1}

(Trying region $Tr'_i$)                 (Leaving region $Lv'_i$)
begin                                   if $ident$ = 1 then $Lv_1$
    if $ident$ = 0 then begin           else if $ident$ = 2 then $Lv_2$
        $P_{wb}(mutex2)$,                   .
        $ident$ := $procno$;                .
        $procno$ := $ident$ + 1;        else if $ident$ = $N$ then $Lv_N$
        $V_{wb}(mutex2)$                else skip
    end else skip,
    if $ident$ = 1 then $Tr_1$
    else if $ident$ = 2 then $Tr_2$
        .
        .
        .
    else if $ident$ = $N$ then $Tr_N$
    else skip
end
```

FIG. 4. Construction of solution in NBW ∩ SYM.

region, each process "picks a number" and saves it in the local variable *ident*. This value is subsequently used to select among the N trying- and leaving-region protocols.

5. "*No Memory*" *and the Power of Weak Semaphores*

The solution presented in Figure 4 has the somewhat pathological property that it is only superficially symmetric but in essence is an asymmetric solution. This superficial symmetry is obtained by having each process "pick a number" the first time through the trying region. This number is retained throughout the lifetime of the process and is used to select among the N different synchronization protocols. Thus processes have "memory" in the sense that they retain and use information about previous synchronization history to modify future synchronization protocols. It is noted in [2] that few solutions to mutual exclusion in the literature have this memory property.

We may formalize the notion of memory by saying that a mutual-exclusion system has *no memory* if for each process i, variable $v^i \in L_i$, and reachable state q such that process i is in the noncritical region in state q, $q(v^i) = q^I(v^i)$. Let NM denote the set of all mutual-exclusion systems with no memory. Note that the solution of Figure 2 has the no-memory property, and thus there is a blocked-set semaphore solution to the starvation-free mutual-exclusion problem in NBW ∩ SYM ∩ NM. In contrast, the results of this section will show that there is no such solution using weak semaphores.

LEMMA 5.1 (WEAK BINARY LOCK-STEP). *Lemma 3.5 holds for mutual-exclusion systems with weak binary semaphores even if α is allowed to contain* V *operations.*

PROOF. The details of the proof by induction on the length of α are omitted. □

LEMMA 5.2. *Let Γ be a mutual-exclusion system with weak binary semaphores. Suppose α is an execution sequence for process 1 from a state q_0. If q_0' is a state such that $q_0' \; {}_1{=}_1 \; q_0$, and if $q_0'(s) \geq q_0(s)$ for all semaphore variables s, then α is an execution sequence from q_0' as well.*

PROOF. Note that it suffices to prove the lemma for finite sequences α, since if α is infinite, then application of the finite case shows that any finite prefix of α is an execution sequence from q_0', and hence α itself is an execution sequence from q_0'. It is convenient to prove a somewhat stronger result, namely, in addition to proving that α is an execution sequence from q_0', prove that if $q_m = nxt(q_0, \alpha)$ and $q_m' = nxt(q_0', \alpha)$, then $q_m \; {}_1{=}_1 \; q_m'$ and $q_m'(s) \geq q_m(s)$ for all semaphore variables s. The details introduce nothing new and are omitted. □

THEOREM 5.1. *There is no solution to the starvation-free mutual-exclusion problem in SYM ∩ NM that has weak binary semaphores.*

PROOF. Suppose Γ is a solution to the starvation-free mutual-exclusion problem that is in SYM ∩ NM. Suppose further that Γ has weak binary semaphores. We will construct an execution sequence for Γ that starves process 2. Since this contradicts the assumption that Γ is starvation free, we conclude that Γ cannot exist.

Let α be the execution sequence for process 1 constructed in Lemma 3.2, and let σ be the corresponding state sequence. We will first show that for each $i > 0$, $\alpha(\mathrm{ncr}_i, \mathrm{cr}_i)$ contains at least one **P** operation. To see this, suppose there were no **P** operations in $\alpha(\mathrm{ncr}_i, \mathrm{cr}_i)$ for some i. Because Γ is in SYM ∩ NM, we know that processes 1 and 2 are similar and that $\sigma(\mathrm{ncr}_i) \; {}_1{=}_2 \; \sigma(\mathrm{ncr}_i)$. Application of Lemma 5.1 shows that $alt(\alpha, \mathrm{ncr}_i, \mathrm{cr}_i)$ is also an execution sequence from $\sigma(\mathrm{ncr}_i)$, and that if $q = nxt(\sigma(\mathrm{ncr}_i), alt(\alpha, \mathrm{ncr}_i, \mathrm{cr}_i))$, then $q \; {}_1{=}_2 \; q$ and $q \; {}_1{=}_1 \; \sigma(\mathrm{cr}_i)$. Hence processes 1 and 2 are both in the critical region in state q, a contradiction of the assumption that Γ has the mutual-exclusion property.

Thus each $\alpha(\mathrm{ncr}_i, \mathrm{cr}_i)$ must contain at least one **P** operation. Let the indices $p_1, p_2, \ldots$ be defined so that $\alpha(p_i)$ is the first **P** operation in $\alpha(\mathrm{ncr}_i, \mathrm{cr}_i)$, and let s_i be the corresponding semaphore variable. Since S is finite but there are infinitely many s_i, there must be one semaphore variable $\bar{s}$ such that $s_i = \bar{s}$ for infinitely many i. Let k be the least i for which $s_i = \bar{s}$. We will now show that $\beta = \alpha(0, \mathrm{ncr}_k)alt(\alpha, \mathrm{ncr}_k, p_k)\alpha(p_k, \infty)$ is an initial execution sequence for Γ that starves process 2.

Obviously $\alpha(0, \mathrm{ncr}_k)$ is an initial execution sequence for Γ. Now $\alpha(\mathrm{ncr}_k, p_k)$ contains no **P** operations, and $\sigma(\mathrm{ncr}_k) \; {}_1{=}_2 \; \sigma(\mathrm{ncr}_k)$. Application of Lemma 5.1 shows that $alt(\alpha, \mathrm{ncr}_k, p_k)$ is an execution sequence from state $\sigma(\mathrm{ncr}_k)$. In addition, if $q = nxt(\sigma(\mathrm{ncr}_k), alt(\alpha, \mathrm{ncr}_k, p_k))$, then $q \; {}_1{=}_2 \; q$ and $q \; {}_1{=}_1 \; \sigma(p_k)$. Thus in state q both processes 1 and 2 are about to execute $\mathbf{P}(\bar{s})$ operations. In addition, all semaphore variables have the same values in state q as they do in state $\sigma(p_k)$. Lemma 5.2

therefore shows that $\alpha(p_k, \infty)$ is an execution sequence from q for Γ. Thus β is an infinite initial execution sequence for Γ.

To show that β starves process 2, note that process 1 enters and exits its critical region infinitely often in β, as a consequence of the construction of α in Lemma 3.2. Also, process 2 remains forever in the trying region in β, about to execute a $\mathbf{P}(\bar{s})$ operation. Since $\alpha(p_k, \infty)$ contains infinitely many $\mathbf{P}(\bar{s})$ operations and each $\mathbf{P}(\bar{s})$ sets $\bar{s}$ to zero, process 2 is disabled infinitely often in β. The sequence β is therefore a valid execution sequence in which process 2 starves. This contradicts the assumption that Γ is starvation free, and we conclude that Γ cannot exist. □

If α is a finite sequence of transitions for a mutual-exclusion system with weak general semaphores, and if s is a semaphore variable, then define the *index of* α *with respect to* s, denoted $ind(s, \alpha)$, to be the number of $\mathbf{P}(s)$ transitions in α minus the number of $\mathbf{V}(s)$ transitions in α. The notion of the index of a sequence of transitions is useful for the following reason: If q is a state and α is an execution sequence from q, then $(nxt(q, \alpha))(s) = q(s) - ind(s, \alpha)$.

LEMMA 5.3 (WEAK GENERAL LOCK-STEP). *Suppose processes* 1 *and* 2 *are similar processes in a mutual-exclusion system with weak general semaphores,* q_0 *is a state such that* $q_0 \;{}_1{=}_2\; q_0$, *and* α *is an execution sequence of length* m *for process* 1 *from* q_0. *If* $q_0(s) - 2\cdot ind(s, \alpha(0, j)) \geq 0$ *for all* $s \in S$ *and* $0 < j \leq m$, *then the sequence* $\beta = alt(\alpha, 0, m)$ *is also an execution sequence from* q_0. *In addition, if* $q'_m = nxt(q_0, \beta)$, *then* $q'_m \;{}_1{=}_2\; q'_m$, $q'_m \;{}_1{=}_1\; q_m$, *and for all* $s \in S$, $q'_m(s) = q_0(s) - 2\cdot ind(s, \alpha)$.

PROOF. The proof is by induction on the length of α as before. The additional hypothesis is used to show that the various $\mathbf{P}$ operations are enabled. □

LEMMA 5.4. *Let* Γ *be a mutual-exclusion system with weak general semaphores. Suppose* α *is an execution sequence for process* 1 *from a state* q_0. *If* $q'_0 \;{}_1{=}_1\; q_0$ *and* $q'_0(s) - ind(s, \alpha(0, j)) \geq 0$ *for all* $s \in S$ *and* $0 < j \leq |\alpha|$, *then* α *is an execution sequence from* q'_0 *as well.*

PROOF. The details of the proof by induction on the length of α are omitted. □

THEOREM 5.2. *Any solution to the starvation-free mutual-exclusion problem that has weak general semaphores and is in* $SYM \cap NM$ *also has busy-waiting.*

The construction in the proof of this theorem is somewhat more involved than those that have appeared so far, so it will be convenient to separate out some reasonably independent parts of the proof as Lemmas 5.5 and 5.6 and Corollary 5.1.

In the sequel, let Γ be a solution to the starvation-free mutual-exclusion problem that has weak general semaphores and is in SYM $\cap$ NM. Let α be the infinite initial execution sequence for Γ constructed in Lemma 3.2, and let σ be the corresponding state sequence. If $s \in S$, then let $\#_{P(s)}(i, j)$ denote the number of $\mathbf{P}(s)$ operations in the subsequence $\alpha(i, j)$ of α. Let $ind(s, i, j)$ abbreviate $ind(s, \alpha(i, j))$. Define $val(s, i, j) = (\sigma(i))(s) - 2\cdot ind(s, i, j)$. Intuitively $val(s, i, j)$ represents the value the semaphore s would have if processes 1 and 2 were to execute the lock-step execution sequence $alt(\alpha, i, j)$ from state $\sigma(i)$.

The intuition behind the following lemma is that if processes 1 and 2 begin executing in lock-step from state $\sigma(\mathrm{ncr}_i)$ for some $i \geq 1$, then there must be some $\mathbf{P}$ operation before both processes reach the critical region, where the processes are forced to "split up."

LEMMA 5.5. *For each* $i \geq 1$ *there exists an index* $split_i$, *which is the least* j *with* $ncr_i \leq j < cr_i$, *such that* $\alpha(j)$ *is a* $\mathbf{P}$ *operation on some semaphore* s *with* $val(s, ncr_i, j) < 2$.

Moreover, if s_i^* denotes the semaphore variable on which the transition $\alpha(split_i)$ operates, then $val(s_i^*, ncr_i, split_i) = 1$.

Proof. If $val(s, \mathrm{ncr}_i, j) \geq 2$ for each j with $\mathrm{ncr}_i \leq j < \mathrm{cr}_i$ and $\alpha(j)$ a **P** operation on some semaphore variable s, then we could apply Lemma 5.3 to show that $alt(\alpha, \mathrm{ncr}_i, \mathrm{cr}_i)$ is an execution sequence from $\sigma(\mathrm{ncr}_i)$. Since this means that processes 1 and 2 would both be in the critical region in state $nxt(\sigma(\mathrm{ncr}_i), alt(\alpha, \mathrm{ncr}_i, \mathrm{cr}_i))$, it must be the case that there is at least one j with $\mathrm{ncr}_i \leq j < \mathrm{cr}_i$ and $\alpha(j)$ a **P** operation on some semaphore s, such that $val(s, \mathrm{ncr}_i, j) < 2$. Let $split_i$ be the least such j, and let s_i^* be the corresponding semaphore variable.

It remains to be shown that $val(s_i^*, \mathrm{ncr}_i, split_i) \neq 0$. By the construction of $split_i$ in the preceding paragraph, we know that $val(s, \mathrm{ncr}_i, j) \geq 0$ for all semaphore variables s and all j with $\mathrm{ncr}_i \leq j \leq split_i$. We may therefore apply Lemma 5.3 to show that $alt(\alpha, \mathrm{ncr}_i, split_i)$ is an execution sequence from $\sigma(\mathrm{ncr}_i)$. If $q = nxt(\sigma(\mathrm{ncr}_i), alt(\alpha, \mathrm{ncr}_i, split_i))$, then $q(s_i^*) = val(s_i^*, \mathrm{ncr}_i, split_i)$. But processes 1 and 2 are both about to execute a $\mathbf{P}(s_i^*)$ operation in state q and hence would be deadlocked if $q(s_i^*) = 0$. This would be a contradiction, and we conclude that $val(s_i^*, \mathrm{ncr}_i, split_i) = 1$, as asserted. □

Processes 1 and 2 can execute the sequence $alt(\alpha, \mathrm{ncr}_i, split_i)$ from $\sigma(\mathrm{ncr}_i)$ before being forced to split up, but there is no guarantee that process 1 will be able to continue alone unhindered. We would like an answer to the question of how far it is safe for processes 1 and 2 to execute in lock-step from $\sigma(\mathrm{ncr}_i)$, if process 1 is then to continue alone unhindered. The answer to this question depends upon how great the values of the semaphore variables are in state $\sigma(\mathrm{ncr}_i)$ and is given by Corollary 5.1 below.

Lemma 5.6. *Let $i > 0$, and let $c: S \rightarrow \mathbf{N}$, with the property that $(\sigma(j))(s) \geq c(s)$ for all $j \geq ncr_i$ and all $s \in S$. Then there exists an index $safe_i$, which is the least j with $ncr_i \leq j \leq split_i$, such that the transition $\alpha(j)$ is a **P** operation on some semaphore variable s, and $ind(s, ncr_i, j) = c(s)$.*

Proof. It suffices to show the existence of one such j, with $s = s_i^*$. Let $I(j)$ abbreviate $ind(s_i^*, \mathrm{ncr}_i, j)$. Thus $I(\mathrm{ncr}_i) = 0 \leq c(s_i^*)$. Now $val(s_i^*, \mathrm{ncr}_i, split_i) = 1$ by the definition of $split_i$. But $val(s_i^*, \mathrm{ncr}_i, split_i) = (\sigma(\mathrm{ncr}_i))(s_i^*) - 2 \cdot I(split_i)$. Hence

$$\begin{aligned} I(split_i) &= (\sigma(\mathrm{ncr}_i))(s_i^*) - I(split_i) - 1 \\ &= (\sigma(split_i))(s_i^*) - 1 \\ &= (\sigma(split_i + 1))(s_i^*) \\ &\geq c(s_i^*). \end{aligned}$$

Thus $I(\mathrm{ncr}_i) = 0$ and $I(split_i) \geq c(s_i^*)$. But $|I(j) - I(j+1)| \leq 1$ for $\mathrm{ncr}_i \leq j < split_i$. Hence $I(j) = c(s_i^*)$ for some j with $\mathrm{ncr}_i \leq j \leq split_i$. □

Corollary 5.1. *The sequence $\alpha(safe_i, \infty)$ is an execution sequence from state $nxt(\sigma(ncr_i), alt(\alpha, ncr_i, safe_i))$.*

Proof. Let $q = nxt(\sigma(\mathrm{ncr}_i), \alpha(\mathrm{ncr}_i, safe_i))$, and let

$$q' = nxt(\sigma(\mathrm{ncr}_i), alt(\alpha, \mathrm{ncr}_i, safe_i)).$$

By the definition of $safe_i$, $q'(s) \geq q(s) - c(s)$ for all $s \in S$. Since $(\sigma(j))(s) \geq c(s)$ for all $j \geq \mathrm{ncr}_i$ and all $s \in S$, it follows that $q(s) - ind(s, safe_i, j) \geq c(s)$ for all $j \geq safe_i$. Thus $q'(s) - ind(s, safe_i, j) \geq 0$ for all $j \geq safe_i$. Since $q \;{}_1{=}_2\; q'$, Lemma 5.4 shows that $\alpha(safe_i, \infty)$ is an execution sequence from q'. □

PROOF OF THEOREM 5.2. We will show that Γ has busy-waiting by showing the existence of a semaphore variable $\bar{s}$ such that for any $M \geq 0$ there is an $m > 0$ with $\#_{P(\bar{s})}(\mathrm{ncr}_m, \mathrm{cr}_m) > M$. This will be accomplished as follows. We will show the existence of a sequence of functions $c_0, c_1, \ldots$ mapping S to $\mathbb{N}$ such that

(1) For all $i \geq 0$, $c_i < c_{i+1}$, where $c_i < c_{i+1}$ is defined to mean that for all semaphore variables s, $c_i(s) \leq c_{i+1}(s)$, and strict inequality holds for at least one s.

Since there are only a finite number of semaphore variables, property (1) implies that there must be a semaphore variable $\bar{s}$ such that $c_i(\bar{s})$ increases in an unbounded fashion with increasing i. We will show in addition the existence of a sequence of natural numbers $k_0, k_1, \ldots$ with the properties:

(2) for all $i \geq 0$, all $j \geq \mathrm{ncr}_{k_i}$, and all semaphore variables s, $(\sigma(j))(s) \geq c_i(s)$;
(3) for any n there is an $m \geq n$ such that $\#_{P(\bar{s})}(\mathrm{ncr}_{k_m}, \mathrm{cr}_{k_m}) \geq c_m(\bar{s})$.

Properties (1) and (3) show that Γ has busy-waiting, since given $M \geq 0$ we may choose n such that $c_n(\bar{s}) \geq M$, then select $m \geq n$ so that $\#_{P(\bar{s})}(\mathrm{ncr}_{k_m}, \mathrm{cr}_{k_m}) \geq c_m(\bar{s})$.

We now turn to the inductive construction of the functions c_i and numbers k_i.

Basis. Define $k_0 = 1$, and let $c_0(s) = 0$ for all $s \in S$. Obviously, for all $j \geq \mathrm{ncr}_{k_0}$ and all $s \in S$, $(\sigma(j))(s) \geq c_0(s)$.

Induction step. Suppose for some $i \geq 0$ we have a function c_i and number k_i such that for all $j \geq \mathrm{ncr}_{k_i}$ and all $s \in S$, $(\sigma(j))(s) \geq c_i(s)$.

Lemma 5.6 and Corollary 5.1 show the existence of safe_{k_i}, such that $\beta = \mathit{alt}(\alpha, \mathrm{ncr}_{k_i}, \mathit{safe}_{k_i})\alpha(\mathit{safe}_{k_i}, \infty)$ is an execution sequence from $\sigma(\mathrm{ncr}_{k_i})$ for Γ. In β, process 1 executes infinitely many critical regions while process 2 remains in the trying region, about to execute a P operation on some semaphore, call it $\tilde{s}_i$. If β were valid, then we would have a contradiction of the assumption that Γ is starvation free. Hence β must not be valid. The only way for this to occur is if there exists $k_{i+1} > k_i$ such that $q_j(\tilde{s}_i) > 0$ for all $j \geq \mathrm{ncr}_{k_{i+1}}$, where $q_j = \mathit{nxt}(\sigma(\mathrm{ncr}_{k_i}), \mathit{alt}(\alpha, \mathrm{ncr}_{k_i}, \mathit{safe}_{k_i})\alpha(\mathit{safe}_{k_i}, j))$. But since $q_j(\tilde{s}_i) > 0$ and $q_j(\tilde{s}_i) = (\sigma(j))(\tilde{s}_i) - c_i(\tilde{s}_i)$, we have that $(\sigma(j))(\tilde{s}_i) \geq c_i(\tilde{s}_i) + 1$. Define $c_{i+1}(s) = c_i(s)$ for all $s \in S$ with $s \neq \tilde{s}_i$, and define $c_{i+1}(\tilde{s}_i) = c_i(\tilde{s}_i) + 1$. This completes the construction of c_{i+1} and k_{i+1} from c_i and k_i.

It is easy to see that the c_i and k_i defined in this way have properties (1) and (2). To see that (3) holds as well, let $\bar{s}$ be the semaphore variable, whose existence was argued above, such that $c_i(\bar{s})$ increases in an unbounded fashion with increasing i. It must be the case that for infinitely many i, $\bar{s}$ is the semaphore $\tilde{s}_i$. Since for each such i, $\mathit{ind}(\bar{s}, \mathrm{ncr}_{k_i}, \mathit{safe}_{k_i}) = c_i(\bar{s})$, and since $\#_{P(\bar{s})}(\mathrm{ncr}_{k_i}, \mathit{safe}_{k_i}) \geq \mathit{ind}(\bar{s}, \mathrm{ncr}_{k_i}, \mathit{safe}_{k_i})$, property (3) holds as well. □

6. *Summary and Conclusions*

The differences in power among the various types of semaphore primitives are summarized in Figure 5, where each column corresponds to various intersections of the classes NBW, SYM, and NM. Each row corresponds to a type of semaphore primitives. The entries in the table are either Y (for "yes"), indicating the existence of a solution to the starvation-free mutual-exclusion problem in the given class that uses the given type of semaphores, or N, which indicates that no such solution exists. The entry 2 indicates that there is a solution to the starvation-free mutual-exclusion problem with weak general semaphores in the class SYM ∩ NM for two processes. The solution supporting this statement was not presented in this paper but may be

	NBW SYM —	— SYM NM	NBW SYM NM
Weak binary	Y	N	N
Weak general	Y	2	N
Blocked-set binary	Y	Y	Y
Blocked-set general	Y	Y	Y

FIG. 5. Relative "power" of the various primitives.

found in [21]. Note that this solution, as well as that of Example 4.2, refutes Morris' claim in [18] that weak semaphores "obviously" cannot be used to implement starvation-free mutual-exclusion even for two processes. It is an open question whether there exist solutions of this type for more than two processes; however, a result of [21] shows that if such a solution exists, it must make use of local variables.

Note that five of the eight combinations of NBW, SYM, and NM are missing from Figure 5. All these combinations except NM and NBW ∩ NM may be filled in by noting that if a solution exists in a given class of mutual-exclusion systems, then that same solution is in any larger class. For the remaining two classes note that it is meaningless to impose the requirement of no memory in the absence of symmetry, since in an asymmetric solution global variables may be used in a disciplined fashion to simulate the effect of local variables.

The relationship among the various types of semaphore primitives may be summarized as follows. Blocked-queue semaphores are rather powerful primitives which admit a trivial solution to the starvation-free mutual-exclusion problem. Blocked-set semaphores are somewhat weaker than blocked-queue semaphores; however, it is still possible to use them to implement starvation-free mutual exclusion if we are willing to accept a somewhat more complicated solution. Weak semaphores are much weaker than either blocked-set or blocked-queue semaphores, since there are no nice solutions to the starvation-free mutual-exclusion problem using these primitives. The borderline between weak binary and weak general semaphores is quite fine, as is evidenced by the fact that weak general semaphores barely suffice to implement starvation-free mutual exclusion in a situation where weak binary semaphores do not.

The existence of weak general semaphore solutions in SYM ∩ NM, coupled with Theorem 5.1, shows a sense in which weak general semaphores are strictly more powerful than weak binary semaphores. Dijkstra [6] claims that general semaphores are superfluous given binary semaphores as primitives. At least for weak semaphores, if we are concerned about the properties of symmetry and no memory in our solution, this is not the case. This also explains why any attempt to implement weak general semaphores with weak binary semaphores, such as that in [20, p. 78], must either introduce asymmetry, violate the no-memory property, or fail.

An interesting extension to the work presented here would be the investigation of semaphore-free solutions in more detail. An unanswered question here is: How many global variables are required to implement starvation-free mutual exclusion for N processes? It is not difficult to show that two processes require at least two variables, and it seems intuitive that the number of variables should increase with the number of processes. Semaphore-free solutions are asymmetric. Consequently, such solutions are always presented in parameterized form; that is, the program run by process i depends upon the number i. The question arises: Must these solutions also be parameterized by the total number of processes N as well? It seems as though it might be possible to use the lock-step construction to prove that this must be the case. For the lock-step construction to be applicable to asymmetric solutions requires increasing the number of processes in the system large enough to match up initial segments of execution sequences for various processes.

The major contribution of this paper is that it brings the murky issues of fairness often mentioned in the synchronization literature into sharper focus. The attempt at precise definitions of the various types of semaphores helps to clear up confusion that has resulted from informal discussion. Many pages have been, and continue to be, spent in the literature in arguments over whether one program solves or does not solve a particular synchronization problem. Often such arguments are useless, since precise specifications are lacking, both for the synchronization problem itself, and for what it means to "solve" that problem. It is hoped that this paper makes a small step toward the resolution of this difficulty.

ACKNOWLEDGMENTS. I would like to thank Russ Atkinson, Sheng-Yang Chiu, Professor Irene Greif, Professor John Guttag, Maurice Herlihy, Jeff Jaffe, Craig Schaffert, Carl Seaquist, Bill Weihl, and Jeannette Wing for their valuable comments on drafts of this paper, and on my thesis [21], from which it evolved. The area editor and the referees also provided constructive suggestions for improving this paper. Carl Seaquist deserves the credit for discovering a crucial idea in the proof of Theorem 5.1.

REFERENCES

1. BEN-ARI, M. Temporal logic proofs of concurrent programs. Tech. Rep. 80-44, Tel-Aviv Univ., Tel-Aviv, Israel, Nov 1980.
2. BURNS, J.E., JACKSON, P., LYNCH, N.A., FISCHER, M.J., AND PETERSON, G L. Data requirements for implementation of N-process mutual exclusion using a single shared variable. *J. ACM 29,* 1 (Jan. 1982), 183–205
3. COFFMAN, E.G. JR , AND DENNING, P.J. *Operating System Theory.* Prentice Hall, Englewood Cliffs, N.J., 1973.
4. COURTOIS, P.J., HEYMANS, F , AND PARNAS, D.L. Concurrent control with "readers" and "writers." *Commun. ACM 14,* 10 (Oct. 1971), 667–668.
5. COURTOIS, P.J., HEYMANS, F., AND PARNAS, D Comments on "A Comparison of Two Synchronizing Concepts by P. B. Hansen." *Acta Inf. 1* (1972), 375–376.
6. DIJKSTRA, E.W. Cooperating sequential processes. In *Programming Languages,* F. Genuys, Ed , Academic Press, New York, 1968, pp. 43–112
7. DIJKSTRA, E.W Hierarchical ordering of sequential processes. *Acta Inf. 1* (1972), 115–138.
8. DOEPPNER, T W On abstractions of parallel programs. Proc. 8th ACM Symp. on Theory of Computing, Hershey, Pa., 1976, pp. 65–72.
9. HABERMANN, A.N. Synchronization of communicating processes. *Commun. ACM 15,* 3 (Mar. 1972), 171–176
10. HABERMANN, A.N. Review of article by Leon Presser on multiprogramming coordination. *Comput. Rev. 29,* 788 (Apr. 1976), 150–151
11. KELLER, R.M Formal verification of parallel programs. *Commun. ACM 19,* 7 (July 1976), 371–384.
12. KNUTH, D.E. Additional comments on a problem in concurrent programming control. *Commun. ACM 9,* 5 (1966), 321–322.
13. KOSARAJU, S.R. Limitations of Dijkstra's semaphore primitives and Petri nets. Rep. No. 25, Computer Science Dep., Johns Hopkins Univ., Baltimore, Md., 1973.
14. KWONG, Y.S. On the absence of livelocks in parallel programs. In *Semantics of Concurrent Computation,* Lecture Notes in Computer Science 70, Gilles Kahn, Ed., Springer-Verlag, New York, 1978, pp. 172–190.
15 LIPTON, R.J. On synchronization primitive systems. Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1973.
16. LIPTON, R.J Reduction· A method of proving properties of parallel programs. *Commun. ACM 18,* 12 (Dec 1975), 717–721.
17. MILLER, R.E., AND YAP, C K. Formal specification and analysis of loosely connected processes. IBM Res. Rep. RC 6716, IBM Thomas J. Watson Research Lab., Yorktown Heights, N.Y., 1977.

18. Morris, J.M. A starvation-free solution to the mutual-exclusion problem. *Inf. Proc. Lett. 8*, 2 (Feb. 1979), 76–80.
19. Presser, L. Multiprogramming coordination. *Comput. Surv. 7*, 1 (Mar. 1975), 21–44.
20. Shaw, A.C. *The Logical Design of Operating Systems.* Prentice Hall, Englewood Cliffs, N.J , 1974.
21. Stark, E.W. Semaphore primitives and starvation-free mutual exclusion. Tech. Rep. TM-158, Lab. for Computer Science, M I.T., Cambridge, Mass., 1980.

RECEIVED MAY 1980; REVISED APRIL 1981; ACCEPTED JUNE 1981