A FIRST COME FIRST SERVED MUTUAL EXCLUSION ALGORITHM WITH SMALL COMMUNICATION VARIABLES

Edward A. Lycklama
Vassos Hadzilacos

Department of Computer Science
University of Toronto, Toronto, Canada

ABSTRACT

We present an algorithm for the mutual exclusion problem that satisfies the "first come first served" property and requires only five shared bits per participant. The algorithm works in a model of concurrency that does not assume atomic operations.

August 26, 1991

1. Introduction

Mutual exclusion is a basic and classic synchronisation problem. Informally it can be formulated as follows. We have a collection of N processes each having the structure illustrated in Figure 1.1.

---

repeat
  NCS
  Trying Protocol
  CS
  Exit Protocol
forever

Figure 1.1: Mutual exclusion problem

Each process iterates repeatedly through a non-critical section (NCS) and a critical section (CS). The problem is to design a Trying Protocol and and Exit Protocol which surround the critical section and synchronise access to it in the following sense:

Mutual Exclusion: No two processes should ever be inside their critical sections at the same time.

Deadlock Freedom: A set of processes wishing to enter the critical section do not forever defer to each other — with the result that none ever enters.

Deadlock freedom does not preclude the possibility of starvation: a process i wishing to enter the critical section may be forever prevented from doing so, as long as some other process j goes through the critical section infinitely often, always being deferred to by i. Thus, a stronger condition is sometimes required:

Lockout Freedom: A process wishing to enter the critical section will eventually do so.

Even lockout freedom may not be strong enough, as it does not place any bound on how many times a process waiting in the trying protocol to enter the critical section may be "overtaken" by others. To ensure strongly fair behaviour we might wish the algorithm to provide access to the critical section on a first come first served (FCFS) basis.

In this paper we are interested in mutual exclusion algorithms that work under the following assumptions about the environment:

- Processes can communicate only by reading or writing shared variables.
- Reading or writing a shared variable is *not* an atomic (i.e., indivisible) action. Thus a process may be reading a variable while another is writing into it and, in that event, the value returned to the reader is arbitrary. This is Lamport's model of non-atomic operations which will be described more formally in Section 2.
- Processes are neither dead nor infinitely fast. I.e., in an infinite amount of time each process takes infinitely many steps; while in a finite amount of time each process can take only finitely many steps. Beyond this, however, no other assumption is made regarding the speeds of processes: processes take steps in a completely asynchronous

manner.

The main result of this paper is a FCFS mutual exclusion algorithm that uses five communication bits per process. To our knowledge, this is the first algorithm to achieve these properties with a constant number of shared bits per process. Previous solutions require $O(N)$ shared bits per process.

To place our algorithm in perspective, we give a brief history of the mutual exclusion problem.† Dijkstra's seminal paper [1965] contained the first statement and solution of the mutual exclusion problem. The environment assumed was akin to the one outlined above, except that read and write operations were assumed to be atomic. However, no atomic "read-and-write" operation (such as a "test-and-set") was assumed. Dijkstra's solution satisfied mutual exclusion and deadlock freedom but was not lockout-free. Knuth [1966] pointed out this weakness and gave an algorithm that remedied it. deBruijn [1967], and Eisenberg and McGuire [1972] improved the fairness of Knuth's lockout-free algorithm.

In a landmark paper, Lamport [1974] pointed out that the previous algorithms provided mutual exclusion at a high level (critical section) assuming mutual exclusion at a lower level (read and write operations on shared variables). He presented the well-known "bakery algorithm" which works *without* atomic reads and writes and satisfies FCFS. On the negative side, that algorithm requires shared variables of unbounded size. Among other algorithms that work without atomic reads and writes are: An algorithm devised independently by Burns [1981] and Lamport [1986b] which satisfies mutual exclusion and deadlock freedom (but not lockout freedom) and uses only one communication bit per process; algorithms by Peterson [1983] and Lamport [1986b] which satisfy mutual exclusion, deadlock freedom and lockout freedom and use, respectively, two and three communication bits per process; an algorithm by Katseff [1978] which satisfies mutual exclusion, deadlock freedom and the FCFS property, and uses $2N$ communication bits per process; and a modification of this algorithm by Lamport [1986b] which satisfies the same properties and uses N bits per process.

Mutual exclusion algorithms, especially those that work in a model of non-atomic read and write operations, are notoriously subtle. Experience teaches that nothing short of a detailed and rigorous proof should be allowed to dispel scepticism about their correctness. Lamport [1986b] has devised a very successful model and methodology with which to prove the correctness of such algorithms. We make use of this methodology in our proof of correctness for the new algorithm.

The remainder of the paper is organised as follows. In Section 2 we present the formal model of concurrent executions that we shall use. In Section 3 we describe the mutual exclusion problem in terms of that model. In Section 4 we give the new algorithm and prove it has the stated properties. We conclude with Section 5.

† This account focuses only on those aspects of mutual exclusion algorithms that are addressed in this paper: the strength of the fairness properties they satisfy and the size of the communication variables they use. Other aspects — in particular, fault-tolerance — are not addressed. We briefly return to the issue of fault-tolerance in Section 5.

2. A model of concurrent executions

In this section we present a model of concurrent process executions. The model is essentially the "global time model" of Lamport [1986a]. It assumes that operations of different processes occur within a common time frame (even though no process may be "aware" of that time frame) but individual operations of different processes may overlap in time. Indeed this is so even for conflicting operations, such as reading and writing a shared variable.

We fix a set of N *processes* numbered $1, 2, \ldots, N$ and a set of *communication variables*. Each communication variable x is *owned* by exactly one process, has a finite set of possible values, called its *range*, and may have a specified *default value* in its range. A default value, if specified, is used as the communication variable's initial value; if no default value is specified, the communication variable's initial value is an arbitrary one in its range.

We further assume a set of *elementary operation types*, representing the kinds of operations that processes can perform. Among them are Read(x) and Write(x) operation types for reading and writing, respectively, communication variable x. By the term *elementary operation* we mean a particular execution of an elementary operation type by some process. There can be many distinct elementary operations that are executions of the same elementary operation type (even by the same process). A Read(x) elementary operation *returns* a value in the range of x; a Write(x) elementary operation *writes* a value in the range of x.

A *(system) execution* is a tuple $\Gamma = (O_1, \ldots, O_N, \longrightarrow, - - \rightarrow)$, where

- O_i is a countably infinite set of elementary operations. It represents the set of elementary operations of process i in execution Γ.
- If $W \in O_i$ is a Write(x) elementary operation then i must be the owner of x. That is, only the owner of a communication variable can write into it.
- $O_i \cap O_j = \emptyset$ for any $i \neq j$.
- $\longrightarrow$ and $- - \rightarrow$ are binary relations on $\bigcup_{1 \leq i \leq N} O_i$. These relations satisfy certain axioms listed below.

Informally, $\longrightarrow$ and $- - \rightarrow$ indicate the temporal ordering of elementary operations. The intention is that $a \longrightarrow b$ iff a precedes b (i.e., a ends before b begins); and that $a - - \rightarrow b$ iff a precedes or is concurrent with b (i.e., a starts no later than b ends). Formally, the properties of these relations are expressed by the following axioms (for any elementary operations a, b, c, d):

General Axioms:

A1 $\longrightarrow$ is a partial order.†

A2 If $a \longrightarrow b$ then $a - - \rightarrow b$.

A3 $a \longrightarrow b$ iff $b - \not\rightarrow\ a$.

A4 If $a \longrightarrow b - - \rightarrow c$ or $a - - \rightarrow b \longrightarrow c$ then $a - - \rightarrow c$.

† A relation is a partial order if it is irreflexive and transitive.

A5 If $a \longrightarrow b \dashrightarrow c \longrightarrow d$ then $a \longrightarrow d$.

A6 $\{e : e \dashrightarrow a\}$ is finite

A7 If a, b belong to the same process then $a \longrightarrow b$ or $b \longrightarrow a$.

Communication Axioms:

In addition, there are two axioms that pertain specifically to read and write operations. A8 states that for each communication variable there is a write operation that produces the variable's initial value. A9 asserts that a Read(x) operation that is not concurrent with any Write(x) operation must return the value last written into x. More precisely, for each communication variable x:

A8 There is a Write(x) operation W such that for any Read(x) operation R, $W \longrightarrow R$.

A9 Let R be a Read(x) operation such that for any Write(x) operation W, either $W \longrightarrow R$ or $R \longrightarrow W$. Let W^* be the $\longrightarrow$-maximum element of the set

$$X = \{W : W \text{ is a Write}(x) \text{ elementary operation and } W \longrightarrow R\}.$$

Then R must return the value written by W^*.†

Fix an execution $\Gamma = (O_1, \ldots, O_N, \longrightarrow, \dashrightarrow)$. An *operation* of process i is a non-empty $A \subseteq O_i$ with the following property: If $a, b \in A$ and there is some $c \in O_i$ such that $a \longrightarrow c \longrightarrow b$ then $c \in A$. A is a *terminating* operation if it is finite; otherwise, it is *non-terminating*.

By slight abuse of notation, we shall write a for both the elementary operation a and the operation $\{a\}$. In this sense, all elementary operations are terminating. We shall now extend the relations $\longrightarrow$ and $\dashrightarrow$ to arbitrary operations (not just elementary ones) A, B as follows:

$A \longrightarrow B$ iff $\forall a \in A, b \in B \;\; a \longrightarrow b$

$A \dashrightarrow B$ iff $\exists a \in A, b \in B \;\; a \dashrightarrow b$.

With these definitions, it is straightforward to check that axioms A1–A5 hold for arbitrary operations, not just elementary ones. In the sequel we shall refer to these axioms even when speaking of (non-elementary) operations. An operation A is a *suboperation* of operation B iff $A \subseteq B$. The following is an immediate consequence of the definitions of $\longrightarrow$ and $\dashrightarrow$.

Lemma 2.1. *Let A, B be operations in a system execution.*

(i) If $A \longrightarrow B$ then $A' \longrightarrow B'$ for any suboperations A', B' of A, B respectively.

(ii) If there are suboperations A', B' of A, B, respectively, so that $A' \dashrightarrow B'$ then $A \dashrightarrow B$. ∎

---

† We note that the W^* of A9 is well-defined: By A6 and A2, X is finite. By A8, it is non-empty. By the fact that all Write(x) operations belong to the process that owns x, A7 and A1, $\longrightarrow$ is a total order for X. Any non-empty, totally ordered, finite set has a maximum element.

We now prove two simple but useful results. Two Write(x) operations W_1 and W_2 are called *successive* if $W_1 \longrightarrow W_2$ and there is no Write(x) operation W so that $W_1 \longrightarrow W \longrightarrow W_2$.

Lemma 2.2. *Let W_1, W_2 be successive Write(x) operations, writing values $v_1 \neq v_2$ respectively, and R be a Read(x) operation returning v_2.*

(i) If $R \longrightarrow W_2$ then $R \dashrightarrow W_1$.

(ii) If $W_1 \longrightarrow R$ then $W_2 \dashrightarrow R$.

Proof: (i) Suppose, by way of contradiction, that $R \longrightarrow W_2$ but $R \not\dashrightarrow W_1$. By A3, $W_1 \longrightarrow R$. But then R is not concurrent with any Write(x) operation (because W_1 and W_2 are successive). By A9, R should return v_1, contrary to the assumption. (ii) is the contrapositive of (i) (cf. A3). ∎

Lemma 2.3. *Consider an execution with a finite number of Write(x) and an infinite number of Read(x) operations. Then all but a finite number of the Reads will return the value written by the last Write.*

Proof: Let W be the last of the (finitely many) Write(x) operations and v the value it writes. By A6 the set $X = \{R : R \text{ is a Read}(x) \text{ operation and } R \dashrightarrow W\}$ is finite. By A3, for every Read(x) operation $R \notin X$, $W \longrightarrow R$. Since W is the last Write(x) operation, every $R \notin X$ is not concurrent with any Write(x) and therefore, by A9, must return v. ∎

The model presented above is nearly identical to that proposed by Lamport [1986b], except that we incorporated in it the "global time assumption" (cf. Lamport [1986a]). More specifically, our model differs from his only in axioms A3 and A9. Lamport uses a weaker version of A3, namely,

A3′ If $A \longrightarrow B$ then $B \not\dashrightarrow A$.

but a stronger version of A9. He defines an operation A to be *effectively non-concurrent* with B if $A \dashrightarrow B$ or $B \dashrightarrow A$ but not both. Instead of A9, he requires

A9′ For any Read(x) and Write(x) operations R, W, either $R \dashrightarrow W$ or $W \dashrightarrow R$. Furthermore, if a Read(x) operation R is effectively non-concurrent with any Write(x) operation, it returns the value written into x by the last Write(x) operation W^* such that $W^* \dashrightarrow R$.

From a pragmatic point of view the net effect of strengthening A3 and weakening A9 in this manner is zero: whatever interesting property can be shown with Lamport's formulation can also be shown with ours and vice-versa. Having proved the correctness of mutual exclusion algorithms in both ways, we have concluded that ours lead to slightly shorter and more transparent proofs.

3. Mutual exclusion algorithms

An *algorithm* is a set of system executions. Informally, if we are given programs for each of the processes and certain assumptions about the environment in which these programs are to be executed (such as assumptions on synchrony) the set of system executions in question are those that follow the processes' programs and conform with the stated assumptions on the environment.

The system executions that comprise a mutual exclusion algorithm have a special structure. Let $\Gamma = (O_1, \ldots, O_N, \longrightarrow, \dashrightarrow)$ be such an execution. The set of process i's elementary operations in Γ, O_i, is partitioned into a finite or infinite set of operations $\mathrm{NCS}_i^j, \mathrm{T}_i^j, \mathrm{CS}_i^j, \mathrm{X}_i^j$ for $j = 1, 2, \ldots, k, \ldots$ such that

$$\mathrm{NCS}_i^j \longrightarrow \mathrm{T}_i^j \longrightarrow \mathrm{CS}_i^j \longrightarrow \mathrm{X}_i^j \quad \text{and} \quad \mathrm{X}_i^j \longrightarrow \mathrm{NCS}_i^{j+1} \text{ for all } j.$$

NCS_i^j, T_i^j, CS_i^j and X_i^j represent, respectively, the execution of the non-critical section, the trying protocol, the critical section and the exit protocol in the jth iteration of process i. It is assumed that,

(M1) Each CS_i^j is an elementary operation.

(M2) No communication variable read or written by a T_i^j or X_i^j operation is written by an NCS_i^j or CS_i^j operation.

(M1) reflects the fact that critical section executions must terminate and that we do not know or care what takes place inside them. The reason we don't also take the NCS_i^j's to be elementary is that an NCS may be non-terminating. (M2) states that the shared variables used by the trying and exit protocols to synchronise access to the critical section should not be altered elsewhere.

We also require that the mutual exclusion algorithm work no matter what a process does in its critical and non-critical section — as long as these operations satisfy (M1) and (M2). We can formally express this as follows:

(M3) Let $\overline{\mathrm{CS}}_i^j, \overline{\mathrm{NCS}}_i^j$ for $j = 1, 2, \ldots, k, \ldots$ be operations that satisfy (M1) and (M2), and $\overline{\mathrm{NCS}}_i^j$ be terminating iff NCS_i^j is. Then the execution $\overline{\Gamma}$ obtained from Γ by replacing each CS_i^j by $\overline{\mathrm{CS}}_i^j$ and each NCS_i^j by $\overline{\mathrm{NCS}}_i^j$ is also in the set of executions that comprise the mutual exclusion algorithm.

For FCFS algorithms we must further assume that each trying operation T_i^j begins with a bounded suboperation D_i^j, called the doorway. More formally, to say that "T_i^j begins with D_i^j" means that $\mathrm{D}_i^j \longrightarrow \mathrm{T}_i^j \setminus \mathrm{D}_i^j$. To say that the doorway operations are *bounded* means that there is a fixed number k such that $|\mathrm{D}_i^j| \le k$ for every i, j in *any* execution of the algorithm.

Given these assumptions about the system executions that make up a mutual exclusion algorithm, here are the properties we require of them. For any system execution Γ:

Mutual Exclusion (ME): For any distinct operations CS_i^j, $\mathrm{CS}_{i'}^{j'}$, *either* $\mathrm{CS}_i^j \longrightarrow \mathrm{CS}_{i'}^{j'}$ *or* $\mathrm{CS}_{i'}^{j'} \longrightarrow \mathrm{CS}_i^j$.

Deadlock Freedom (DF): If there is a non-terminating T_i^j *operation then, for some process* i', *there are infinitely many operations* $\mathrm{CS}_{i'}^{j'}$.

Exit Protocol Termination (EPT): Every X_i^j *operation is terminating.*

Satisfaction of these three properties is the minimum required of mutual exclusion algorithms. As mentioned in the Introduction, however, some additional properties may be desirable.

Lockout Freedom (LF): Every T_i^j *operation is terminating.*

First Come First Served (FCFS): If $\mathrm{D}_i^j \longrightarrow \mathrm{D}_{i'}^{j'}$ *and* $\mathrm{CS}_{i'}^{j'}$ *occurs in* Γ *then* $\mathrm{CS}_i^j \longrightarrow \mathrm{CS}_{i'}^{j'}$.

If we think of the doorway as the operation whereby a process requests access to the critical section, LF and FCFS ensure that processes gain access to the critical section in the order they made their requests. (Processes that make their requests "simultaneously", i.e., are in their doorways at the same time, may enter the critical section in any order.) It is immediate from the definitions that LF implies DF. Furthermore, Lamport [1986b, p. 330] proves the following

Theorem 3.1. *FCFS and DF imply LF.* ∎

A composition theorem

It is possible to construct FCFS mutual exclusion algorithms in a modular way by "composing" two independently designed algorithms. One, the "ME algorithm", provides exclusive (but not necessarily FCFS) access to the critical section. The other, the "FCFS algorithm", ensures FCFS (but not necessarily exclusive) access to the critical section. The composition is effected by embedding the former inside the latter, as illustrated in Figure 3.1, where ";" denotes sequential composition (for the moment think of the "I" operation in the FCFS algorithm as a critical section). That is, the trying protocol T of the composed algorithm consists of the trying protocol TF of the FCFS algorithm followed by the trying protocol TM of the mutual exclusion algorithm. Accordingly, its exit protocol X consists of the two exit protocols in reverse order: first the exit protocol XM of the ME algorithm and then the exit protocol XF of the FCFS algorithm.

This technique seems to have first been employed by Katseff [1978]. Lamport [1986b, p. 343] makes the general observation that if each of the two algorithms being composed satisfies its respective property (independently of the other) then their composition satisfies both properties; furthermore, if each is deadlock free then so is their composition. In this subsection we formalise this observation by stating the precise conditions under which the composition works properly.

repeat	**repeat**	**repeat**
NCS	NCS	NCS
TM (ME trying)	TF (FCFS trying)	T = TF; TM
CS	I	CS
XM (ME exit)	XF (FCFS exit)	X = XM; XF
forever	**forever**	**forever**
A_1 : ME algorithm (i)	A_2 : FCFS algorithm (ii)	A_3 : Embedded algorithm (iii)

Figure 3.1: Embedded Algorithm Structure

First, we must insist that the ME and FCFS algorithms do not use the same (local or communication) variable for conflicting purposes. For, if they did, even though each algorithm by itself works correctly, when we compose them one might find "its" variables' values altered by the other and misbehave. A simple solution to this problem is to require that the two algorithms use disjoint sets of variables.

There is a further complication. The proof that the composition works requires us to regard an execution of the composed algorithm as an execution of the FCFS algorithm, by considering the embedded ME algorithm as the critical section of the FCFS algorithm. This abstraction is not legitimate if the ME algorithm is not lockout-free, because then the trying protocol of the ME algorithm may be non-terminating while a critical section operation *must* be terminating. This issue is relevant because our own algorithm uses a ME algorithm that is not lockout-free.

The resolution of this problem requires some technical maneuvering. In particular, we must allow the "critical section" of the FCFS algorithm to be a potentially non-terminating operation. More precisely, the FCFS algorithm consists of executions that have the structure of mutual exclusion algorithm executions as described at the beginning of this section, except that we no longer require condition (M1) — and (M3) is modified to reflect that. To emphasise this point we shall call the operation that comes between the trying and exit protocols in the FCFS algorithm an *interior* rather than a critical section; instances of this operation for process i will be denoted as I_i^j, instead of CS_i^j.

Now consider the FCFS algorithm in isolation — cf. Figure 3.1(ii). The FCFS property for such an algorithm is exactly the same as given before (for mutual exclusion algorithms) except that CS_i^j is replaced by I_i^j. The Deadlock Freedom property should be amended to account for the possibility of a non-terminating interior operation. In particular, deadlock freedom for FCFS algorithms is:

Deadlock Freedom: If there is a non-terminating T_i^j *operation then for some process* i' *either there are infinitely many interior operations* $\mathrm{I}_{i'}^{j'}$ *or a non-terminating interior operation* $\mathrm{I}_{i'}^{j'}$.

To highlight the difference from the deadlock freedom property as it applies to mutual exclusion algorithms (with terminating critical sections), we shall refer to this property as "DF′", instead of "DF". We are now ready to state the composition theorem.

Theorem 3.2. *Let A_1, A_2 and A_3 be the algorithms depicted in Figure 3.1. If*

(i) A_1 satisfies ME, EPT and DF;

(ii) A_2 satisfies FCFS, EPT and DF′; and

(iii) the set of variables used in the trying and exit protocols of A_1 (TM and XM in Figure 3.1) is disjoint from the set of variables used in the trying and exit protocols of A_2 (TF and XF in Figure 3.1)

then A_3 satisfies ME, DF, EPT, LF and FCFS.

Proof sketch: We may view any execution of A_3 as an execution of either A_1 or as an execution of A_2: By hiding the FCFS subalgorithm of A_3 (consisting of TF and XF) inside the NCS, we get an execution of A_1; by hiding the ME subalgorithm of A_3 (consisting of TM and XM) inside the CS we get an execution of A_2. Assumption (iii), in conjunction with (M3), ensures that by this "hiding" of pieces of A_3 inside others, we actually obtain legitimate executions of A_1 and A_2. It is now easy to see that an assumed violation of ME by A_3 implies a violation of ME by A_1 in contradiction to assumption (i). Similarly, an assumed violation of FCFS by A_3 implies a violation of FCFS by A_2. An assumed violation of EPT by A_3 implies a violation of EPT by A_1 or A_2, contradicting (i) or (ii). And an assumed violation of DF by A_3 implies a violation of DF by A_1 or of DF′ by A_2, contradicting (i) or (ii). This implies that A_3 satisfies ME, DF, EPT and FCFS. By Theorem 3.1, it will also satisfy LF.

The details of this proof may be found in Lycklama [1987]. ∎

This theorem helps simplify the correctness proofs of FCFS mutual exclusion algorithms. Instead of considering the entire algorithm at once (and hence the interaction of the two subalgorithms) it is enough — the theorem assures us — to consider each subalgorithm separately.

4. The new algorithm

In this section we present a FCFS mutual exclusion algorithm which requires five bits of shared variables per process. The algorithm has the structure mentioned in the previous section, namely it consists of an ME algorithm embedded inside a FCFS algorithm. Thus we shall make use of the composition theorem in its proof.

The section is divided in four parts. In the first we give the ME algorithm that we embed in our new FCFS algorithm. In the second we give the complete algorithm and informally explain how and why it works. In the third part we define some notation and in the last part we prove the correctness of the algorithm.

4.1 The Burns-Lamport algorithm

To provide mutual exclusion, we use an algorithm that was devised independently by Burns [1981] (see also Burns and Lynch [1980]) and Lamport [1986b]. This algorithm uses just one communication bit per process. The basic idea is this: Each process sets its bit to indicate its interest in entering the critical section and tests the bits of lower-numbered processes. If it finds any of them interested in the critical section, it gives up and restarts. If, on the other hand, none of the lower-numbered processes has its bit set, it proceeds to the next phase where it tests the bits of higher-numbered processes, waiting for each one of them that it finds interested in entering the critical section (but now merely waiting, without giving up and starting over). Surprisingly, this simple idea works.

```
communication var:
    x_i: boolean := false

local var:
    t: 1..N

repeat
    NCS        % non-critical section %
L:  x_i := true
    for t := 1 to i - 1 do
        if x_t then
        begin
            x_i := false
            while x_t do skip
            goto L
        end
    for t := i + 1 to N do
        while x_t do skip
    CS         % critical section %
    x_i := false
forever
```

Figure 4.1: The Burns-Lamport Algorithm

The algorithm executed by process i is given in Figure 4.1. The description begins with a listing of the communication variables owned by i (in this case only x_i), including their type and initial value — if one needs to be specified. The declaration of communication variables is followed by a listing of the process' local variables and then the "code". The linguistic constructs used are standard with the possible exception of the instruction **skip** by which we denote the "null" (do-nothing) operation.

Theorem 4.1. *(Lamport [1986b]) The Burns-Lamport algorithm satisfies ME, EPT and DF.* ∎

```
communication var:
    c_i, v_i, x_i: boolean := false
    turn_i: array [0..1] of 0..1†

local var:
    j, t: 1..N
    bit: 0..1
    S: array [1..N] of array [0..1] of 0..1

   repeat
       NCS            % non-critical section %
       c_i := true    % doorway begins %
       for j := 1 to N do
           S[j] := turn_j
       bit := 1 − bit
       turn_i[bit] := 1 − turn_i[bit]
       v_i := true    % doorway ends %
       c_i := false
       for j := 1 to N do
           while c_j or (v_j and S[j] = turn_j) do skip
L:     x_i := true    % Burns-Lamport algorithm begins %
       for t := 1 to i − 1 do
           if x_t then
           begin
               x_i := false
               while x_t do skip
               goto L
           end
       for t := i + 1 to N do
           while x_t do skip
       CS             % critical section %
       x_i := false   % Burns-Lamport algorithm ends %
       v_i := false
   forever
```

Figure 4.2: The New Algorithm

† Recall that if no explicit value is specified for some communication variable, we assume that it is initialised to some arbitrary value in its range. This is needed to ensure satisfaction of axiom A8.

4.2 The new algorithm

Each process i executes the algorithm in Figure 4.2. Lines 12–23 constitute the Burns-Lamport algorithm which provides mutual exclusion and is considered as the "interior" of an algorithm designed to enforce the FCFS property. The trying protocol of this latter algorithm consists of lines 3–11 and the exit protocol is line 24.

In our algorithm, each process i owns a communication variable $turn_i$ which, in some sense, indicates the iteration that process i is currently executing. More specifically, $turn_i$ is a two-bit variable whose value is modified in each iteration of process i by alternatingly complementing the value of *one* of its bits (i.e., in each iteration we complement the bit that was not complemented in the previous iteration). Thus, the value of $turn_i$ cycles through 00, 01, 11, 10 (or, reversing the direction of the cycle, 00, 10, 11, 01).† We think of each such modification of $turn_i$ as "incrementing" its value. Its initial value does not matter.

Recall that the trying protocol of a FCFS algorithm begins with a bounded operation, the doorway, followed by a waiting operation. In the doorway of our algorithm (lines 3–8), process i reads other processes' $turn_j$ variables and stores their values in a local array S (where $S[j]$ contains the value of $turn_j$ observed by i). It then increments its $turn_i$ to indicate that this is a new iteration, and signals its intention to enter the interior by setting a Boolean variable v_i to true. The doorway is surrounded by a Boolean variable c_i which is set to true on entry (line 3) and to false after exiting the doorway. Thus, c_i = true indicates that process i is in the doorway.

In the waiting operation of the trying protocol (lines 9–11), process i waits for a process j for one of two reasons:

(a) j is in the doorway (i.e., c_j = true). This is done to ensure that a doorway operation of i overlaps with *at most one* doorway operation of j.

(b) FCFS requires i to wait for j. This is the case if

  i. j has left the doorway before i entered it; and

  ii. j has not left the interior yet.

  These mean that v_j is still true and that $turn_j$ has not changed its value since i read and recorded it in $S[j]$ when it was in its doorway. In other words, i must wait for j if it finds v_j = true and $S[j] = turn_j$.

In the exit protocol (line 24) process i simply signals that it is leaving the interior by setting v_i to false.

Point (b) above provides an informal explanation as to why the algorithm satisfies the FCFS property. Unfortunately, it is not nearly as easy to see why it is deadlock free. We next give an informal proof of deadlock freedom.

Ignoring what happens in the ME algorithm (lines 12–23), a deadlock of the FCFS algorithm can only arise if there is a cyclic sequence of processes, each forever waiting for

† The direction of the cycle is determined by the initial value of the local variable *bit*, which can be arbitrarily chosen to be either 0 or 1.

the next in line 11. If k is forever waiting for l while both processes are in line 11 (and thus c_l = false and v_l = true) it must be that the value returned when k last read $turn_l$ was precisely the value that l last wrote into $turn_l$. Since every process *first* reads the others' $turn$ variables and *then* writes its own, at least one of the waiting processes, say k which is waiting for l, must have last read $turn_l$ *before* l last wrote into $turn_l$.

Refer now to Figure 4.3. The five boxes correspond to the last five Write($turn_l$) operations. The width of a box indicates the duration of the corresponding operation and the value written by that operation is shown inside the box. The specific values chosen in the figure are not important — what is significant is that each write operation only changes one bit of $turn_l$ (indicated by an asterisk below the bit that was changed).

Figure 4.3: Deadlock freedom of new algorithm

Let R stand for k's last Read($turn_l$) operation. By assumption, R returns 00, the value last written into $turn_l$. As we have just argued, R cannot have occurred within interval A (cf. Figure 4.3), because it must precede the last write of $turn_l$. Now we argue that it cannot have occurred within interval B either. First of all, note that R cannot have occurred inside any of the intervals B1, B2 or B3 — for, if so, it would have returned 01, 11 or 10, respectively. Thus if R occurred within B it must have overlapped with either the second last or the third last write of $turn_l$. It cannot have overlapped with both since a doorway of k can overlap at most one doorway of l (cf. point (a) above). If R overlapped with the second last write of $turn_l$, it would have read the correct value of $turn_l[0]$ (the bit *not* changed by that write) which at that time is 1. Similarly, if R overlapped with the third last write of $turn_l$, it would have read the correct value of $turn_l[1]$ which at that time is also 1. Both cases contradict the assumption that R returns 00. We conclude that R did not occur within interval B. Hence, R must overlap with or precede the *fourth last* write of $turn_l$. Since a doorway of k can overlap with at most one doorway of l, the last doorway of k must precede the *third last* doorway of l. The FCFS property then implies that k must complete its last interior — and therefore its last trying protocol — before l even enters its third last interior. But if k completes what is supposedly its last trying protocol, it is not, after all, deadlocked!

4.3 Notation

To give a rigorous proof of the algorithm's properties we shall need some notation that will allow us to identify operations, especially Read and Write operations, in executions of the algorithm. The notation must enable us to specify: the process where the operation occurs, whether the operation is a Read or Write, the variable and the value read or written, the line number (in terms of Figure 4.2) where the Read or Write operation comes from,† and a number indicating the iteration (through the **repeat-forever** loop) in which the operation occurred. For example, process k's operation which writes the value true into c_k in line 3 of iteration 17 will be denoted as $[k.3 : W(c_k, \text{true})]^{17}$, and the Read of x_l by process k returning, say, value true in line 14 of iteration 13 will be denoted as $[k.14 : R(x_l, true)]^{13}$.

This notation is rather convoluted and in many cases not all the detail it conveys is needed. Usually we shall be indifferent as to the iteration and shall therefore omit the superscript, thinking of the operation as occurring in the "current iteration". However, in some instances we'll be interested in iterations relative to the "current" one; these will be indicated with relative superscripts (in particular, $+1$ for the "next" iteration). Also, the value read or written will be omitted except when its inclusion enhances clarity. Finally, when a line contains a single operation, we can identify the operation simply by indicating the executing process and the line where the operation occurs. For example, the first of the two examples given above can be identified simply as $k.3$.

4.4 Proof of correctness

Theorem 4.2. *The algorithm in Figure 4.2 satisfies ME, DF, EPT, FCFS and LF.*

Proof: The proof is based on the composition theorem (Theorem 3.2). We consider the algorithm in Figure 4.2 to be an instance of a FCFS algorithm (as in Figure 3.1(ii)) with trying protocol TF consisting of lines 3–11 and exit protocol XF consisting of line 24. Lines 12–23 comprise the interior. It is obvious that the FCFS algorithm satisfies EPT, since its exit protocol is bounded. We'll show that it also satisfies FCFS and DF′ (cf. Lemmata 4.1 and 4.3 below). Theorem 4.1 says that the interior is an algorithm satisfying ME, EPT and DF. The composition theorem then implies Theorem 4.2. ∎

Lemma 4.1. *The algorithm in Figure 4.2 satisfies FCFS.*

Proof: Fix an arbitrary execution of the algorithm. We must show that for any processes k and l, if $\mathrm{DW}_k \longrightarrow \mathrm{DW}_l$ and I_l occurs in the execution then $\mathrm{I}_k \longrightarrow \mathrm{I}_l$. Suppose, to the contrary, that $\mathrm{I}_k \not\longrightarrow \mathrm{I}_l$. Since I_l appears in the execution, $l.11$ terminates and, in particular, $l.11$ must find either $v_k = false$ or $turn_k \neq S[k]$.

Case 1: $l.11$ finds $v_k = false$. Then we claim that $[l.11 : R(v_k, \text{false})] \dashrightarrow [k.8 : W(v_k, \text{true})]$. To see this consider two subcases.

† This is necessary in order to distinguish operations that occur in more than one place in the algorithm (such as reading $turn_j$ that occurs in line 5 and in line 11) and convenient so that the reader can quickly refer to the algorithm.

Subcase a: Operation $k.24$ appears in the execution. Then so does I_k and because, by assumption, $\mathrm{I}_k \not\longrightarrow \mathrm{I}_l$, A3 implies that $\mathrm{I}_l \dashrightarrow \mathrm{I}_k$. Then

$$\begin{array}{lcl} \text{A5} & \Rightarrow & [l.11 : R(v_k, \text{false})] \longrightarrow [k.24 : W(v_k, \text{false})] \\ \text{Lemma 2.2(i)} & \Rightarrow & [l.11 : R(v_k, \text{false})] \dashrightarrow [k.8 : W(v_k, \text{true})], \text{ as wanted.} \end{array}$$

Subcase b: Operation $k.24$ does not appear in the execution. Then $[k.8 : W(v_k, \text{true})]$ is the last Write(v_k) operation. Then

$$\begin{array}{lcl} \text{A9} & \Rightarrow & [k.8 : W(v_k, \text{true})] \not\longrightarrow [l.11 : R(v_k, \text{false})] \\ \text{A3} & \Rightarrow & [l.11 : R(v_k, \text{false})] \dashrightarrow [k.8 : W(v_k, \text{true})], \text{ as wanted.} \end{array}$$

Since $[l.11 : R(v_k, \text{false})] \dashrightarrow [k.8 : W(v_k, \text{true})]$, we have:

$$\begin{array}{lcl} \text{Lemma 2.1(ii)} & \Rightarrow & \mathrm{DW}_l \dashrightarrow \mathrm{DW}_k \\ \text{A3} & \Rightarrow & \mathrm{DW}_k \not\longrightarrow \mathrm{DW}_l \end{array}$$

contrary to the assumption that $\mathrm{DW}_k \longrightarrow \mathrm{DW}_l$.

Case 2: $l.11$ finds $turn_k \neq S[k]$. By assumption, $\mathrm{DW}_k \longrightarrow \mathrm{DW}_l$ and thus:

$$\begin{array}{lcl} \text{Lemma 2.1(i)} & \Rightarrow & [k.7 : W(turn_k)] \longrightarrow [l.5 : R(turn_k)]. \\ \text{In addition,} & & [l.5 : R(turn_k)] \longrightarrow [l.11 : R(turn_k)]. \end{array}$$

Hence

$$[k.7 : W(turn_k)] \longrightarrow [l.5 : R(turn_k)] \longrightarrow [l.11 : R(turn_k)] \qquad (*)$$

We claim that $l.5$ and $l.11$ return the same value for $turn_k$. To see this consider two subcases.

Subcase a: Operation $[k.7 : W(turn_k)]^{+1}$ does not appear in the execution. Then $[k.7 : W(turn_k)]$ is the last Write($turn_k$) operation. A9 and (*) imply that $l.5$ and $l.11$ return the same value for $turn_k$, namely that written by $[k.7 : W(turn_k)]$.

Subcase b: Operation $[k.7 : W(turn_k)]^{+1}$ appears in the execution. Then so does I_k. Since, by assumption, $\mathrm{I}_k \not\longrightarrow \mathrm{I}_l$, A3 implies that $\mathrm{I}_l \dashrightarrow \mathrm{I}_k$. By A5, we have that $[l.11 : R(turn_k)] \longrightarrow [k.7 : W(turn_k)]^{+1}$. Then A9 and (*) imply that $l.5$ and $l.11$ return the same value for $turn_k$, namely that written by $[k.7 : W(turn_k)]$.

But if $l.5$ and $l.11$ return the same value for $turn_k$, we get a contradiction to the assumption of Case 2, namely that $l.11$ finds $turn_k \neq S[k]$. ∎

Because the values of $turn_l$ change cyclically, it is possible for k to read a value of $turn_l$ equal to what l assigns to $turn_l$ *before* that write takes place. This is a potentially dangerous state of affairs, for it could lead to processes k and l waiting for each other. The following Lemma shows that this situation cannot arise and is a crucial component in the proof of deadlock freedom.

Lemma 4.2. *Consider iterations m_k and m_l of processes k and l respectively. If $[k.5 : R(turn_l, v)]^{m_k} \longrightarrow [l.7 : W(turn_l, v)]^{m_l}$ then $m_l > 2$ and $[\mathrm{I}_k]^{m_k} \longrightarrow [\mathrm{I}_l]^{m_l - 1}$.*†

---

† It is actually true that when the hypotheses of the lemma hold, $m_l > 3$ and $[\mathrm{I}_k]^{m_k} \longrightarrow [\mathrm{I}_l]^{m_l - 2}$. However, the weaker statement of the lemma is easier to prove and enough for our goal of proving deadlock freedom.

Proof: Let $v_0 v_1$ be the values of the two bits in v. Recall that writing into $turn_l$ is done by writing only one of its two bits. Suppose that $[l.7]^{m_l}$ writes bit b of $turn_l$ (where b is 0 or 1). Since reading $turn_l$ requires reading both bits, by the hypothesis $[k.5 : R(turn_l, v)]^{m_k} \longrightarrow [l.7 : W(turn_l, v)]^{m_l}$ and Lemma 2.1(i), we have

$$[k.5 : R(turn_l[b], v_b)]^{m_k} \longrightarrow [l.7 : W(turn_l[b], v_b)]^{m_l} \qquad (**)$$

We claim that iteration m_l cannot be the first in which l writes v_b into $turn_l[b]$. For, if it was, the initial value of $turn_l[b]$ would have been $1 - v_b$. Then by A8, A9 and (**), $[k.5 : R(turn_l[b])]^{m_k}$ would have returned $1 - v_b$, not v_b. Since iteration m_l is not the first in which bit b of $turn_l$ is written and the bit written alternates in each iteration, it follows that $m_l \neq 1$ and $m_l \neq 2$ — so, $m_l > 2$.

Since successive writes to the same bit of $turn_l$ are two iterations apart and write complementary values to that bit, from (**) and Lemma 2.2(i) we get

$$\begin{array}{lll}
 & & [k.5 : R(turn_l[b]), v_b]^{m_k} \dashrightarrow [l.7 : W(turn_l[b], 1 - v_b)]^{m_l - 2} \\
\text{A5} & \Rightarrow & [k.3 : W(c_k, \text{true})]^{m_k} \longrightarrow [l.11 : R(c_k, \text{false})]^{m_l - 2} \\
\text{Lemma 2.2(ii)} & \Rightarrow & [k.9 : W(c_k, \text{false})]^{m_k} \dashrightarrow [l.11 : R(c_k, \text{false})]^{m_l - 2} \\
\text{A5} & \Rightarrow & [\text{DW}_k]^{m_k} \longrightarrow [\text{DW}_l]^{m_l - 1} \\
\text{Lemma 4.1} & \Rightarrow & [\text{I}_k]^{m_k} \longrightarrow [\text{I}_l]^{m_l - 1},
\end{array}$$

as wanted. ∎

We are now ready to prove,

Lemma 4.3. *The FCFS algorithm in Figure 4.2 satisfies DF′.*

Proof: Suppose, by way of contradiction, a deadlock occurs. Then some process must be executing a non-terminating trying protocol and thus a non-terminating line 11 (all other operations of the FCFS trying protocol are terminating), while no process is executing a non-terminating interior or an infinite number of interior operations.

Hence if process l is not executing a non-terminating $l.11$ then l must be executing a non-terminating NCS. Since no process k in a non-terminating $k.11$ can be waiting for a process l in NCS_l (because $c_l = v_l = \text{false}$ while l is in NCS), there must be a set of processes $k_1, k_2, \ldots, k_r$ for some $2 \leq r \leq N$, with each process k_j waiting forever in line 11 for $k_{j \oplus 1}$.† (Note that $r \neq 1$ because in line 11 each k_j finds $c_{k_j} = \text{false}$ and $S[k_j] \neq turn_{k_j}$.) Without loss of generality we may renumber these processes as $1, 2, \ldots, r$. Thus every $1 \leq k \leq r$ is executing a non-terminating $k.11$ with $j = k \oplus 1$.

While a process l is executing a non-terminating $l.11$, $v_l = \text{true}$ and $c_l = \text{false}$. Thus, Lemma 2.3 implies that every process $1 \leq k \leq r$ will eventually be finding $c_{k \oplus 1} = \text{false}$ and $v_{k \oplus 1} = \text{true}$ forever. We shall show that it is impossible for every $1 \leq k \leq r$ to continually find $turn_{k \oplus 1} = S[k \oplus 1]$ and hence that one such process k will eventually complete $k.11$, contradicting our assumption that every $1 \leq k \leq r$ is executing a non-terminating $k.11$.

† We write $j \oplus 1$ as a shorthand for $(j \bmod r) + 1$, for any $1 \leq j \leq r$.

Suppose each $k = 1, 2, \ldots, r$ is executing a non-terminating $k.11$ in iteration m_k. If $[k.11]^{m_k}$ continually reads $turn_{k\oplus 1} = S[k \oplus 1]$, Lemma 2.3 implies that $k.5$ must have written into $S[k \oplus 1]$ the value, say v, written into $turn_{k\oplus 1}$ by $k \oplus 1.7$ in iteration $m_{k\oplus 1}$ (i.e., the last write operation of $k \oplus 1$ into $turn_{k\oplus 1}$ is $[k \oplus 1.7 : W(turn_{k\oplus 1}, v)]^{m_{k\oplus 1}}$ and process k executed $[k.5 : R(turn_{k\oplus 1}, v)]^{m_k}$). There are two cases to consider:

Case 1: If for some $1 \le k \le r$,

$$\begin{array}{lll} & & [k.5 : R(turn_{k\oplus 1}, v)]^{m_k} \longrightarrow [k \oplus 1.7 : W(turn_{k\oplus 1}, v)]^{m_{k\oplus 1}} \\ \text{Lemma 4.2} & \Rightarrow & m_{k\oplus 1} > 2 \text{ and } [\mathrm{I}_k]^{m_k} \longrightarrow [\mathrm{I}_{k\oplus 1}]^{m_{k\oplus 1}-1} \end{array}$$

contradicting that process k is executing a non-terminating $k.11$ in iteration m_k.

Case 2: If for every $1 \le k \le r$,

$$\begin{array}{lll} & & [k.5]^{m_k} \not\longrightarrow [k \oplus 1.7]^{m_{k\oplus 1}} \\ \text{A3} & \Rightarrow & [k \oplus 1.7]^{m_{k\oplus 1}} \dashrightarrow [k.5]^{m_k} \\ & \Rightarrow & 1.5 \longrightarrow 1.7 \dashrightarrow r.5 \longrightarrow r.7 \dashrightarrow \ldots \\ & & \ldots \dashrightarrow 2.5 \longrightarrow 2.7 \dashrightarrow 1.5 \\ \text{A5} & \Rightarrow & 1.5 \longrightarrow 2.7 \text{ and } 2.7 \dashrightarrow 1.5, \end{array}$$

which contradicts A3.

We therefore conclude that at least one process $1 \le k \le r$ will not be continually finding $turn_{k\oplus 1} = S[k \oplus 1]$ and will therefore not be executing a non-terminating $k.11$. This contradicts the supposition that every process k, $1 \le k \le r$, is executing a non-terminating line 11. Hence, the assumption that there is a deadlock must be wrong, and the algorithm satisfies DF$'$. ∎

We have now established all the ingredients needed in the proof of Theorem 4.2.

5. Conclusion

The number of communication bits required by our algorithm can actually be reduced from five to four per process. This is achieved by replacing variable x_i in the Burns-Lamport ME protocol by c_i. Unfortunately, we can no longer use the composition theorem in proving the correctness of this modification because now the ME and FCFS algorithms that are composed share a communication variable. The ME and FCFS properties for this modification are not hard to prove. For DF we appeal to the lack of deadlock in the ME and FCFS algorithms when considered separately and note that no process in the ME algorithm can be waiting indefinitely for a process in the FCFS algorithm, since it must eventually find c_j false. A detailed proof of correctness for this modification will not be given here.

Lamport [1986b] considered mutual exclusion algorithms that are robust with respect to various types of "abnormal" behaviour. The mildest form of such behaviour is process termination which comes in two varieties: *shutdown* and *abortion*. The former amounts to a process halting forever, at an arbitrary point of its execution, but after having reset its communication variables to their default values (if such values are specified). In our algorithm (as in most others) a process shutdown is indistinguishable from a process executing a non-terminating NCS and the algorithm is therefore correct in the face of shutdowns.

Abortion is like a shutdown except that the process does not halt after resetting its communication variables, but may enter the trying protocol or abort again. Our algorithm is subject to deadlock in the presence of abortions, since abortions allow a process i to change its value of $turn_i$ several times without going through the critical section the same number of times. As a result, Lemma 4.2, which is key to deadlock freedom, is no longer true.

A more severe form of abnormal behaviour is process breakdown which also comes in two varieties: *failure* and *transient malfunction*. In both cases a process behaves arbitrarily for some finite period of time. It may assign arbitrary values to its communication and local variables. In a process failure, the process eventually recognises its erratic behaviour and promptly aborts, whereas in a transient malfunction the process merely stops misbehaving and resumes "normal" execution at any point in its program (with its communication and local variables in the state its erratic behaviour left them). Algorithms that are able to weather the storm of transient malfunctions and eventually bring the system to a state where the global properties that the algorithm is set to achieve are actually re-established, are said to be *self-stabilising*. This concept was first studied by Dijkstra [1974].

Since our algorithm is prone to deadlock in the presence of abortions, it also deadlocks in the presence of process breakdown. We have attempted to devise modifications to the algorithm that will withstand abortions and failures but have not been successful. The algorithm by Lamport [1986b] which uses N communication bits per process, satisfies ME, DF and FCFS and tolerates abortions and failures. Lamport also devised an algorithm that has these properties and is self-stabilising but uses $N!$ communication bits per process. It is an interesting open question whether the communication space cost of these algorithms is inherent in the fault-tolerance they provide.†

Acknowledgement

We thank Kaarel Truuvert and the anonymous referees for their valuable comments on earlier drafts of this paper. This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada.

† While this paper was in the refereeing process, Kaarel Truuvert provided the answer to this question. By embellishing our algorithm with a complex deadlock detection and resolution mechanism, he constructed a self-stabilising, FCFS mutual exclusion algorithm that uses 17 bits per process. (Kaarel Truuvert, M.Sc. thesis, Department of Computer Science, University of Toronto, 1989.)

Bibliography

1. Burns, J.E. [1981]. *Complexity of communication among asynchronous parallel processes.* Ph.D. thesis, Georgia Institute of Technology, Jan. 1981.
2. Burns, J.E. and N.A. Lynch [1980]. "Mutual exclusion using indivisible reads and writes". *Proc. of the 18th Allerton Conf. on Communication, Control and Computing.* Oct. 1980, pp. 833–342.
3. deBruijn, N.G. [1967]. "Additional comments on a problem in concurrent programming control". *CACM* 10(3):137–138 (March 1967).
4. Dijkstra, E.W. [1965]. "Solution of a problem in concurrent programming control". *CACM* 8(9):569 (Sept. 1965).
5. Dijkstra, E.W. [1974]. "Self-stabilizing systems in spite of distributed control". *CACM* 17(11):643–644 (Nov. 1974).
6. Eisenberg, M.A. and M.R. McGuire [1972]. "Further comments on Dijkstra's concurrent programming control problem". *CACM* 15(11):999 (Nov. 1972).
7. Katseff, H.P. [1978]. "A new solution to the critical section problem". *Proc. of the 10th ACM Symp. on the Theory of Computing.* 1978, pp. 86–88.
8. Knuth, D.E. [1966] "Additional comments on a problem in concurrent programming control". *CACM* 9(5):321–322 (May 1966).
9. Lamport, L. [1974]. "A new solution to Dijkstra's concurrent programming problem". *CACM* 17(8):453–455 (Aug. 1974).
10. Lamport, L. [1986a]. "On inter-process communication — Parts I and II". *Distributed Computing* 1(1986):77–101.
11. Lamport, L. [1986b] "The mutual exclusion problem — Parts I and II". *JACM* 33(2):313–348 (Apr. 1986).
12. Lycklama, E.A. [1987]. *A first-come first-served solution to the critical section problem using five bits.* M.Sc. thesis, Department of Computer Science, University of Toronto, October 1987.
13. Peterson, G.L. [1983]. "A new solution to Lamport's concurrent programming problem using small shared variables". *ACM TOPLAS* 5(1):56–65 (Jan. 1983).