DISTRIBUTED MUTUAL EXCLUSION ON A RING OF PROCESSES

Alain J. Martin

Computer Science
California Institute of Technology

5080:TR:83

DISTRIBUTED MUTUAL EXCLUSION ON A RING OF PROCESSES

Alain J. Martin

Computer Science
California Institute of Technology

5080:TR:83

The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

DISTRIBUTED MUTUAL EXCLUSION ON A RING OF PROCESSES

Alain J. Martin
Computer Science
California Institute of Technology
Pasadena CA 91125 USA

April 1983 / October 1983

Abstract

A set of processes called "masters" are sharing a critical section on a mutual exclusion basis. A master communicates only with its private "server". The servers communicate with each other in a ring. Three solutions for solving the mutual exclusion problem are presented. They all rely on the presence of a unique privilege in the ring. The notation used extends CSP input and output commands with a Boolean primitive, the "probe", which makes it possible to determine whether a communication action is pending on a channel.

In the correctness proofs, the concept of "trace" is introduced, i.e. a total ordering of actions corresponding to a possible interleaving of the atomic actions of a concurrent computation.

1. Introduction

A set of N (N>1) cyclic processes, to be called "masters", are using a critical section on a mutual exclusion basis: at most one master at a time can find itself inside its critical section. When a master wants exclusive access to the critical section, it communicates with another process, its "private server". (Hence, there are N master-server pairs.) A master communicates only with its server; the servers also communicate with each other.

We present three simple solutions for the case when the communication network among the servers is a ring. All three solutions rely on the

presence of a unique "privilege" in the ring: when a server has the privilege, its master is guaranteed to have exclusive access to the critical section. (In another paper [5], we also present another type of solutions, and generalize both types for the case when the communication network is an arbitrary, connected graph.)

The solutions differ in the way the privilege is acquired by the servers. In the first solution (called "perpetuum mobile") the privilege circulates continuously. In the other two solutions the privilege is moved only on request. In the second solution ("the reflecting privilege" solution) the requests move in one direction, and the privilege in the opposite direction. In the third solution ("the drifting privilege" solution), requests and privilege move in the same direction.

The algorithms are described in a notation close to but not identical to C.A.R. Hoare's CSP [4]. To the usual pair of communication primitives, another primitive called the probe is added. It is a Boolean function on a channel, that has the same value as the predicate "a communication action is pending on the channel". (The probe is described in more detail in [6].)

Two semantic properties of the algorithms are proved: the mutual exclusion property (at any time at most one master is inside its critical section) and the absence of deadlock. For mutual exclusion, the classical invariant assertion method ([8]) is used. For the absence of deadlock, the concept of trace is introduced, i.e. a total ordering of actions corresponding to a possible interleaving of the actions of a concurrent computation.

Finally, we slightly modify the servers' programs in the last two solutions so as to make these solutions fair, i.e. each request for the critical section is eventually granted.

2. The programming language and its semantics

For the sequential part, we use guarded commands [1], with CSP syntax: [..] instead of if .. fi , and *[...] instead of do ... od. We also simplify *[true → S] to *[S].

Two communication primitives are used: an input command and an output command. Input and output commands are paired by identifiers called channels. For a channel C, C?x is an input command "on" C and C!y is an output command on C. A pair of input and output commands on the same channel fulfils the following semantic axioms.

2.1. Synchronization axioms:

An action is the execution of a command by a process. (The execution of command X by process p will often be referred to as "action p.X".)

Let cC! and cC? be the number of completed output actions and the number of completed input actions on C respectively, at an arbitrary point of the computation. Then

Axiom A0: $cC! = cC?$. □

A0 expresses that the completion of the nth input action on C coincides with the completion of the nth output action on C. The two actions are said to match.

The execution of an input or output command X results in the suspension of X when its completion in the current state of the computation would violate A0. In this case, the process executing X is said to be delayed at X. From suspension until (possible) later completion, an action is pending. The Boolean qX is equal to the predicate "an action X is pending".

Axiom A1: $\neg qC! \vee \neg qC?$. □

(For a justification of the choice of A0 and A1 as synchronization axioms, see [7].)

2.2 Communication axiom:

Let C?x and C!y be matching. If x,y = X,Y before the communication, then x,y = Y,Y after the communication. □

2.3 Probe

A Boolean command on channels is introduced, called the probe. For channel C, the probe $\bar{C}$ has the same value as qC!. (The probe function is more general, but this restricted definition is sufficient for the purpose of this paper. See [6].) Hence, guarded command $\bar{C} \rightarrow C?$ guarantees that action C? is not suspended. But guarded command $\neg\bar{C} \rightarrow C?$ does not guarantee that C? is suspended.

2.4 Traces of a computation

In a computation evoked by process p, to each execution of a command C corresponds a unique action - say action C_n for the nth execution of C. Hence to each computation evoked by p corresponds a (totally ordered) sequence of atomic actions of p. Such a (finite or infinite) sequence is called a trace of (a computation of) p.

2.4.1 Notations and definitions

- The predicate "action X occurs in trace T" is denoted by "$X \in T$".
- In a trace T, the sequence of atomic actions corresponding to the execution of non-atomic command A is called the "imprint" of A in T, and is denoted by i(A).
- The ordering of actions in trace T is denoted by $\prec_T$.

- $i(A) \lessdot_T i(B)$ means $\underline{A}(x,y: x \in i(A) \wedge y \in i(B): x \lessdot_T y)$.

- The part of a trace T posterior to an action X is denoted by T post X. The part of a trace T preceding an action X is denoted by T pre X.

- Two sequences T1 and T2 are called a prefix and a suffix of T respectively, if T1 is finite and T is the catenation of T1 and T2.

- The partition of trace T into a prefix and a suffix pair uniquely identifies a state of the computation, to which we will also refer as "a point in T".

2.4.2 Atomic actions

- Any program part that does not contain communication actions or probes can be considered an atomic action. Since the processes do not share variables, this assumption does not restrict the implementation freedom.

- An input or output action is an atomic action.

- The evaluation of the guards of a selection can be considered an atomic action. This is obviously true if the guards do not contain probes. It is also obviously true for the probed guards used in the solutions, which are either of the form $[\bar{U} \rightarrow \ldots \; [] \; \bar{L} \rightarrow \ldots]$ or of the form $[\bar{U} \rightarrow \ldots \; [] \neg\bar{U} \rightarrow \ldots]$.

2.4.3 Blocking

A computation may be interrupted at a communication action or at the guards of a selection when the completion of the action would violate the semantics of communication or of probes respectively. In such a case the process is said to be blocked at the action. A priori the traces of a process corresponding to all possible blocking situations - the so-called "blocked traces" - are possible traces of the process.

2.4.4 Interleaving

A computation evoked by the concurrent program $P:[P_1 \| P_2 \| \ldots \| P_n]$ is postulated to have the same net effect as a totally ordered computation whose trace T is constructed according to the interleaving rules I0, I1, and I2.

I0: There exists a set $\{t_i : 1 \leq i \leq n\}$ such that t_i is a possible trace of P_i, and T is an interleaving of $t_1, t_2, \ldots, t_n$, i.e.

- for any action x in T, there exists a unique t_i, $1 \leq i \leq n$, such that x is in t_i,
- The "projection" of T on actions of P_i, i.e. the trace obtained by removing all actions of T that do not belong to P_i, is t_i. □

(When necessary, the first imprinted action - say X - of a matching pair (X,Y) of communication actions is distinguished from the second one by denoting them by #X and Y# respectively.)

Let p be a process of P, and t be the projection of T on p.

I1: For two matching communication actions #X and Y#, we say that p is blocked at #X in T iff the last p-action in T is #X and Tpost#X contains no Y-action.

If p is not blocked at #X, and B is an action following #X in t:

$$i(B) \in T \Longrightarrow Y\# \not\leq_T i(B) \ . \quad \square$$

From I1, it follows that the pairs of actions representing successive communications on a channel follow each other in a trace (no overlapping or nesting of matching pairs).

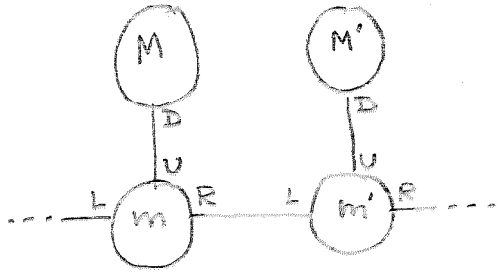
We now define qX as holding exclusively at a point of computation corresponding to a position in a trace between #X and the matching Y#. And we

define both cX and cY in any prefix of a trace to be equal to the number of complete matching (X,Y) pairs in this prefix. With these definitions, it is easy to verify that the ordering of actions in traces satisfies axioms A0 and A1.

I2: Let S be the selection action $[[i: 0 \leq i < N: B_i \rightarrow S_i]$ in process p, and let G: $[[i: 0 \leq i < N: B_i]$ denote the evaluation of the guards of S. We say that p is blocked at G in T iff the last p-action in T is G and $A(i: 0 \leq i < N: \neg B_i)$ holds everywhere in T post G. If p is not blocked at G in T, the imprint of S in T consists of G followed by $i(S_k)$, where S_k is any action of the guarded command set such that B_k holds at some point in T between G and $i(S_k)$. □

3. General properties of the processes

A master uses one channel, with the local name D (for "down"). A server uses three channels, with the local names U,R,L (for "up", "right", and "left"). For a master M and its server m : M.D = m.U. The servers are connected in a ring, i.e. for server m and its right-hand neighbor mr : m.R = mr.L. (See Figure.)



Unless otherwise specified, a current master and its server will be identified with M and m respectively.

The program of a master is

*[NCS; D!; CS; D!] .

The actions NCS and CS (for "non-critical section" and "critical section") leave all variables of the program unchanged and contain no communication action. Further, CS always terminates.

In all solutions, the programs of the servers include exactly one sequence of communication commands on U of the form:

$$\bar{U} \rightarrow \ldots; U?; U?;\ldots$$

where $\bar{U}$ is the probe of channel U.

Let D1 and D2 be the textually first and second commands D! respectively. And let U1 and U2 be the textually first and second commands U? respectively.

Theorem 1: At any point in a trace T, a master is between D1 and D2 (i.e. has completed D1, and has not yet completed D2) iff its server is between U1 and U2. □

Proof: A master M is between D1 and D2 iff $\underline{c}M.D!$ is odd. A server m is between U1 and U2 iff $\underline{c}m.U?$ is odd. By axiom A0, $\underline{c}M.D! = \underline{c}m.U?$. □

Theorem 2: Blocking of a master at D2 does not occur; blocking of a server at U2 does not occur. □

Proof: By Theorem 1, if a master is blocked at D2, its server is blocked at U2. And vice-versa. This contradicts axiom A1. □

Consequently, a server can be blocked neither at U1 (because of the guard $\bar{U}$) nor at U2.

A one-place buffer (here simply called a buffer) is a process whose activity consists in an alternation of input actions on one channel -- say L -- and output actions on another channel --say R. Hence either $0 \leq \underline{c}L? - \underline{c}R! \leq 1$ (if the first action is an input) or $0 \leq \underline{c}R! - \underline{c}L? \leq 1$ (if the first action is an output) holds.

A buffer is sending when its next communication action is an output. It is receiving when its next communication action is an input.

A directed ring of buffers is a set of buffers connected in the same way as the servers. Three simple theorems about a ring of buffers are given without proofs.

Theorem 3: In a directed ring of buffers, if one buffer is blocked, all are blocked. □

Theorem 4: The buffers of a directed ring are blocked iff they are all sending or all receiving. □

Theorem 5: In a directed ring of buffers, the number of sending buffers and the number of receiving buffers, are constant. □

We shall prove two properties of all three algorithms.

Property 1: At most one master at a time is inside its CS. □

From Theorem 1, proving Property 1 amounts to proving that at most one server at a time is between U1 and U2. We attach to each server a ghost variable cs, which is true between U1 and U2 and false otherwise. Proving property 1 amounts to proving that at any point in a trace T:

$$\underline{N}(m{:}m.cs) \leq 1.$$ [1]

Property 2: If a master is blocked in a trace T, T contains infinitely many CS actions, i.e. any suffix T′ of T contains a CS action. □

[1] $\underline{N}(x{:}P(x))$ denotes the number of different values of x for which P(x) holds.

4. First Solution: "Perpetuum Mobile"

Let S be the program

*[L?; [$\bar{U}$ → U?; U? [] ¬$\bar{U}$ → skip]; R!].

The program of one server is R!; S . The program of all other servers is S.

Proof of Property 1:

|== {topology of servers}

N(m:m.cs) ≤ N(m:m is sending) (0)

|== {Servers are buffers, theor. 5, and initial state}

N(m:m is sending) = 1 (1)

|== {(0) and (1)}

N(m:m.cs) ≤ 1 . □

Corollary 0: A server is not blocked. □

Proof: Immediate from (1), and theor. 4 and 5. □

Proof of Property 2: We prove a stronger property: no master can be blocked, i.e. the solution is fair.

Assume M is blocked at a certain action #D1 in T and let T′ be the part of T posterior to that action. $\bar{U}$ holds at any point in T′, and T′ does not contain m.U?. But since m is not blocked, by the topology of m, T′ contains the imprint of [$\bar{U}$ → U?; U?...], and since $\bar{U}$ holds at any point, by I2, T′ contains m.U?. A contradiction. □

5. Second Solution: "The Reflecting Privilege"

```
m :: *[[Ū → [s → s:= false ▯ ¬s → R!]; U?; U?; s:= true
       ▯L̄ → [s → s:= false ▯ ¬s → R!]; L?
      ]] .
```

Initially: N(m:m.s) = 1 .

In this solution and the next one, the program of a server consists of two main guarded commands. The textually first one will be called GC1, the textually second one GC2.

Proof of Property 1:

A ghost Boolean variable p is introduced in each server in the following way:

```
initially p is false
{s  ∧¬p } < s:= false ; p:= true  > {¬s ∧ p }
{¬s ∧ p } < s:= true  ; p:= false > { s ∧ ¬p}
{¬s ∧¬p} < R!         ; p:= true  > {¬s ∧ p }
{¬s ∧ p } < L?        ; p:= false > {¬s ∧ ¬p} .
```

It can be verified that the assertions between braces hold in m as pre- and postconditions of the action they enclose, and thus in any trace T of the concurrent computation since no variables are shared.

|== {from the above pre- and postconditions relations and initialization}
N(m:m.s) + N(m:m.p) = 1 (2)

|== {topology of m}
N(m:m.cs) ≤ N(m:m.p)
==> {from (2)}
N(m:m.cs) ≤ 1 . □

Proof of Property 2:

Property 2 is obviously implied by the two following lemmas.

Lemma 1: In any trace T, if a master is blocked, at least one server is not blocked. □

Lemma 2: In any trace T, either all servers are blocked or any suffix of T contains a CS action. □

Proof of Lemma 1: Observe that since L? is guarded by $\bar{L}$, a server can be blocked only at the guard evaluation $[\bar{U}[\bar{L}]$ or at R?. In any trace T:

M is blocked
==> {definition of probe and I2}
m is not blocked at $[\bar{U}[\bar{L}]$
==> {if all servers are blocked, they are either all blocked at $[\bar{U}[\bar{L}]$ or all blocked at R?}
one server is not blocked $\vee$ all servers are blocked at R!
==> {$(\neg s \wedge \neg p)$ holds as precondition of R! and (2)}
one server is not blocked $\vee$ false. □

Proof of Lemma 2:

Lemma 3: m.L? $\in$ T ==> m.L? $\not\overset{}{\underset{T}{<}}$ CS. □

Proof of Lemma 3: Let $m_0, m_1, \ldots m_k$ be a finite sequence of consecutive servers in the ring, such that m_0 is chosen arbitrarily and m_{i+1} is the left neighbor of m_i.

In the following proof, L_i^n denotes the nth L? action of m_i, R_i^n denotes the nth R! action of m_i, and CS_i denotes a m_i.CS action. Further, $\prec$ stands for $\prec_T$.

$L_0^n \in T$

==> {"$\bar{L}$ → ... L?" and definition of probe}

$R_1^n \prec L_0^n$

==> {according to whether R_1^n belongs to GC1 or GC2 respectively, and by I1}

$R_1^n \prec L_0^n \prec CS_1 \ \vee \ R_1^n \prec L_0^n \prec L_1^n$

==> {repeating the argument for L_1^n}

$L_0^n \prec CS_1 \ \vee \ L_0^n \prec L_1^n \prec CS_2 \ \vee \ L_0^n \prec L_1^n \prec L_2^n$

==> {since the argument can be repeated only a finite number of times}

$\underline{E}(k: 0 < k < N-1 : L_0^n \prec L_1^n \prec \ldots \prec CS_k)$. □

If a server m is not blocked in trace T, then by the topology of m, an arbitrary suffix T′ of T contains either m.CS or m.L?. By Lemma 3, T′ contains a CS action in both cases. □

6. Third Solution: "The Drifting Privilege"

```
m :: *[[Ū → *[¬s → R!s; L?s]; U?; U?
       ▯L̄ → L?r; r,s:= (r ∨ s),false; R!r
      ]] .
```

Initially: N(m:m.s) = 1 .

Proof of Property 1: A ghost Boolean variable p is introduced in each server in the following way:

initially p is false

{¬p} < L?r ; p:= r > {p=r}

{p=r} < r,s:= (r ∨ s),false; p:= r > {¬s ∧ p=r}

{¬s ∧ p=r} < R!r ; p:= false > {¬s ∧ ¬p} .

It can then be verified that the assertions between braces hold in m as pre- and postconditions of the action they enclose.

It is also easy to verify that ¬s ∧ ¬p holds as pre- and postcondition of R!s in GC1.

- Let V be N(m:m.s) + N(m:m.p). R!s and <r,s:= (r ∨ s), false; p:= r > leave V invariant. An input action whose input value is false leaves V invariant. An input action whose input value is true increases V by one. But in this case {¬s ∧ p} R!r {¬s ∧ ¬p} holds for the matching output action, i.e. the matching output action decreases V by one. Hence V is left invariant. Since (2) holds initially, (2) is invariantly true.

- From the topology of the server:

|== {s holds as precondition of U?}
N(m:m.cs) ≤ N(m:m.s)
==> {from (2)}
N(m:m.cs) ≤ 1 . □

Proof of Property 2:

(L1 and R1, resp. L2 and R2, denote L? and R! actions from GC1, resp. GC2.)

Proof of Lemma 1: In any trace T:

M is blocked
==> {definition of probe}
m is not blocked at $[\bar{U}\|\bar{L}]$
==> {synchronization axiom}
one server is not blocked $\vee$ all servers are blocked at R! or L?.

all servers blocked at R! or L?
==> {theorem 4}
all servers blocked at R! $\vee$ all servers blocked at L?
==> {at any point: cL? = cR!}
all servers blocked at R1 $\vee$ all servers blocked at L2
==> {"$\bar{L} \rightarrow$ L2"}
all servers blocked at R1
==> {$\neg s \wedge \neg p$ holds as precondition of R1 and (2)}
false . □

Proof of Lemma 2:

- If a server is not blocked in a trace T, no server is blocked in T.

- Assume that no server is blocked in T, and let m be privileged at a given point in T, i.e. m.s $\vee$ m.p holds. After the next m.R! action, the right neighbor of m becomes privileged. Hence the

Lemma 3: If no server is blocked in trace T, each server becomes privileged an unbounded number of times in T. □

- When a server m is between L? and R! in GC2 : cm.L? > cm.R! holds. When a server m is between R! and L? in GC1: cm.R! > cm.L? holds. Since cL? = cR! at any time, when a process is between L? and R! in GC2, another process is between R! and L? in GC1. Hence the

Lemma 4: If no server is blocked in T, any suffix T′ of T contains a GC1-sequence and a GC2-sequence. □

From Lemma 4, any suffix T′ contains a GC1-sequence. From Lemma 3, the server in this GC1-sequence becomes privileged in T′. Hence the GC1-sequence terminates, i.e. T′ contains a CS-action. □

7. Fairness

Obviously, the second and third solutions are not fair: a particular master can be blocked at D1 if its server constantly selects GC2. But it is easy to remedy the situation. Let $*[[\bar{U} \rightarrow C1 \,[]\, \bar{L} \rightarrow C2]]$ be the initial program of m. The fair version is

$$
\begin{aligned}
m' ::\, & *[[\bar{U} \rightarrow C1;\ [\bar{L} \rightarrow C2 \,[]\, \neg\bar{L} \rightarrow skip] \\
& \quad []\, \bar{L} \rightarrow C2;\ [\bar{U} \rightarrow C1 \,[]\, \neg\bar{U} \rightarrow skip] \\
& \quad]] \,.
\end{aligned}
$$

8. Conclusion

The second solution is the most elegant and efficient of the three. In the first and second solutions, only empty messages are transmitted, while the third sends Boolean messages. This means that only the synchronization axiom of the communication primitives is used in the first and second solutions, whereas also the communication axiom is used in the third solution. (It is used for proving the invariance of (2).)

We also observe that for m simultaneous requests, m messages are transmitted on each channel in the third solution. One empty message per channel is transmitted in the first solution. In the second solution, one empty message is transmitted on each channel located between a request and the privilege, and no communication takes place on the other channels.

The second solution is superior to the first one. First, there is no unnecessary communication in the absence of requests for the critical section, and second, when a unique master requests the critical section several times in a row, the requests are granted without communication with other servers, whereas in the first solution, the privilege has to circulate every time.

The fact that in the second solution the completion of the pending requests is used as transmission of the privilege reduces the communication traffic to its minimum. (My first version of this solution, see [3], included an explicit transmission of an empty privilege.)

The use of the probe function facilitates the coding and the proofs but is not absolutely necessary: all solutions can be recoded (albeit less elegantly and less efficiently) in a "pure CSP" style. I am convinced that the introduction of probe function is a significant improvement. (See [6].)

I discovered these solutions several years ago, but refrained from publishing them because I was not satisfied with my way of presenting them. Whether I have succeeded now is left to the appreciation of the reader. Anyway, I decided it was time to let other people study them, and perhaps enjoy them, for they are seminal case studies in the design and verification of distributed computations.

While writing this paper, I received a report by Edsger W. Dijkstra [2] in which he constructs the second solution using regular expressions. It is his second proof of this solution. The first one appeared in [3].

Acknowledgments. I am grateful to David Gries, Martin Rem, and Jan van de Snepscheut for their comments and criticisms. Acknowledgments are also due to Edsger W. Dijkstra and the members of the Tuesday Afternoon Club for their helpful criticisms (conveyed by Wim Feijen) on the first draft of this paper. Mike Schuster suggested a simplification in the coding of the first solution.

References

[1] E.W. Dijkstra, A Discipline of Programming. (Prentice-Hall, Inglewood Cliffs, 1976).

[2] E.W. Dijkstra, Reducing control traffic in a distributed implementation of mutual exclusion. EWD851b. (1983).

[3] E.W. Dijkstra, On the correctness of a design by Alain J. Martin. EWD668. (1978).

[4] C.A.R. Hoare, Communicating Sequential Processes, Comm. ACM, August (1978), 666-677.

[5] A.J. Martin, Distributed Mutual Exclusion Algorithms, Caltech technical report 5047:TR:82. (1982).

[6] A.J. Martin, The Probe: An Addition to Communication Primitives. Caltech technical report 5124:TR:84, (1984).

[7] A.J. Martin, An Axiomatic Definition of Synchronisation Primitives, Acta Informatica, vol. 16 (1981) 219-235.

[8] S. Owicki, D. Gries, An Axiomatic Proof Technique for Parallel programs, Acta Informatica, vol.6, fasc.4, (1976) 319-340.