6. Conclusion

It has been shown that the existence of a σ-coloration of a particular graph is a necessary and sufficient condition for the existence of a solution to the class-teacher timetable problem with unavailability constraints and preassigned meetings. The knowledge of this necessary and sufficient condition does not provide an efficient algorithm which can be applied to an arbitrary timetable problem in order to determine the existence of a solution. The necessary and sufficient condition does, however, show that existing graph coloring algorithms [1, 6, 14, 15, 16] may be applied to timetable problems with unavailability constraints and preassigned meetings.

Acknowledgment. The authors would like to thank the referees for their constructive criticism and helpful suggestions for the improvement of this paper.

Received July 1973; revised February 1974

References
12. Lions, J. Some results concerning the reduction of binary matrices. J. ACM 18, 3 (July 1971), 424-430.

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

Introduction

Knuth [1], deBruijn [2], and Eisenberg and McGuire [3] have given solutions to a concurrent programming problem originally proposed and solved by Dijkstra [4]. A simpler solution using semaphores has also been implemented [5]. These solutions have one drawback for use in a true multicomputer system (rather than a time-shared multiprocessor system): the failure of a single unit will halt the entire system. We present a simple solution which allows the system to continue to operate despite the failure of any individual component.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate despite the failure of any individual component.

Key Words and Phrases: critical section, concurrent programming, multiprocessing, semaphores

CR Categories: 4.32

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by U.S. Army Research Office--Durham, under Contract No. DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, MA 01880.
The Algorithm

Consider \( N \) asynchronous computers communicating with each other only via shared memory. Each computer runs a cyclic program with two parts—a critical section and a noncritical section. Dijkstra’s problem, as extended by Knuth, is to write the programs so that the following conditions are satisfied:
1. At any time, at most one computer may be in its critical section.
2. Each computer must eventually be able to enter its critical section (unless it halts).
3. Any computer may halt in its noncritical section. Moreover, no assumptions can be made about the running speeds of the computers.

The solutions of [1–4] had all \( N \) processors set and test the value of a single variable \( k \). Failure of the memory unit containing \( k \) would halt the system. The use of semaphores also implies reliance upon a single hardware component.

Our solution assumes \( N \) processors, each containing its own memory unit. A processor may read from any other processor’s memory, but it need only write into its own memory. The algorithm has the remarkable property that if a read and a write operation to a single memory location occur simultaneously, then only the write operation must be performed correctly. The read may return any arbitrary value!

A processor may fail at any time. We assume that when it fails, it immediately goes to its noncritical section and halts. There may then be a period when reading from its memory gives arbitrary values. Eventually, any read from its memory must give a value of zero. (In practice, a failed computer might be detected by its failure to respond to a read request within a specified length of time.)

Unlike the solutions of [1–4], ours is a first-come-first-served method in the following sense. When a processor wants to enter its critical section, it first executes a loop-free block of code—i.e., one with a fixed number of execution steps. It is then guaranteed to enter its critical section before any other processor which later requests service.

The algorithm is quite simple. It is based upon one commonly used in bakeries, in which a customer receives a number upon entering the store. The holder of the lowest number is the next one served. In our algorithm, each processor chooses its own number. The processors are named 1, \( \ldots \), \( N \). If two processors choose the same number, then the one with the lowest name goes first.

The common store consists of

\[
\text{integer array choosing [1:N], number [1:N]}
\]

Words \( \text{choosing} \) \( i \) and \( \text{number} \) \( i \) are in the memory of processor \( i \), and are initially zero. The range of values of \( \text{number} \) \( i \) is unbounded. This will be discussed below.

The following is the program for processor \( i \). Execution must begin inside the noncritical section. The arguments of the maximum function can be read in any order. The relation “less than” on ordered pairs of integers is defined by \( (a,b) < (c,d) \) if \( a < c \), or if \( a = c \) and \( b < d \).

begin integer \( j \);
\[ L1: \text{choosing} [i] := 1; \]
\[ \text{number}[i] := 1 + \max \text{ (number}[1], \ldots, \text{number}[N]); \]
\[ \text{choosing} [i] := 0; \]
\[ \text{for } j = 1 \text { step 1 until } N \text { do } \]
\[ \begin{align*}
& L2: \text{if } \text{choosing}[j] \neq 0 \text{ then goto } L2; \\
& L3: \text{if } \text{number}[j] \neq 0 \text{ and } \text{number}[j], j < \text{number}[i], i \text{ then goto } L3;
\end{align*}
\]
\[ \text{end; critical section; number}[i] := 0; \]
\[ \text{noncritical section; goto } L1; \]

We allow processor \( i \) to fail at any time, and then to be restarted in its noncritical section (with \( \text{choosing} [i] = \text{number} [i] = 0 \)). However, if a processor keeps failing and restarting, then it can deadlock the system.

Proof of Correctness

To prove the correctness of the algorithm, we first make the following definitions. Processor \( i \) is said to be in the doorway while \( \text{choosing} [i] = 1 \). It is said to be in the bakery from the time it resets \( \text{choosing} \) \( i \) to zero until it either fails or leaves its critical section. The correctness of the algorithm is deduced from the following assertions. Note that the proofs make no assumptions about the value read during an overlapping read and write to the same memory location.

Assertion 1. If processors \( i \) and \( k \) are in the bakery and \( i \) entered the bakery before \( k \) entered the doorway, then \( \text{number} [i] < \text{number} [k] \).

Proof. By hypothesis, \( \text{number} [i] \) had its current value while \( k \) was choosing the current value of \( \text{number} [k] \). Hence, \( k \) must have chosen \( \text{number} [k] \geq 1 + \text{number} [i] \).

Assertion 2. If processor \( i \) is in its critical section, processor \( k \) is in the bakery, and \( k \neq i \), then \( \text{number} [i], i < \text{number} [k], k \).

Proof. Since \( \text{choosing} \) \( k \) has essentially just two values—zero and nonzero—we can assume that from processor \( i \)’s point of view, reading or writing it is done instantaneously, and a simultaneous read and write does not occur. For example, if \( \text{choosing} \) \( k \) is being changed from zero to one while it is also being read by processor \( i \), then the read is considered to happen first if it obtains a value of zero; otherwise the write is said to happen first. All times defined in the proof are from processor \( i \)’s viewpoint.

Let \( t_{1\delta} \) be the time at which processor \( i \) read \( \text{choosing} \) \( k \) during its last execution of \( L2 \) for \( j = k \), and let \( t_{2\delta} \) be the time at which \( i \) began its last execution of \( L3 \) for \( j = k \), so \( t_{1\delta} < t_{2\delta} \). When processor \( k \) was choosing its
current value of number \([k]\), let \(t_e\) be the time at which it
took to enter the doorway, \(t_w\) the time at which it finished
writing the value of number \([k]\), and \(t_c\) the time at which
it left the doorway. Then \(t_e < t_w < t_c\).

Since choosing \([k]\) was equal to zero at time \(t_{13}\), we
have either (a) \(t_{12} < t_e\) or (b) \(t_e < t_{12}\). In case (a),
assertion 1 implies that number \([i]\) < number \([k]\), so the
assertion holds.

In case (b), we have \(t_w < t_e < t_{12} < t_{13}\), so \(t_w < t_{12}\).
Hence, during the execution of statement L3 begun at time
\(t_{13}\), processor \(i\) read the current value of number \([k]\).
Since \(j\) did not execute L3 again for \(j = k\), it must have
found (number \([i]\), \(i\) < (number \([k]\), \(k\)). Hence, the
assertion holds in this case, too. \(\square\)

Assertion 3. Assume that only a bounded number of
processor failures may occur. If no processor is in its
critical section and there is a processor in the bakery
which does not fail, then some processor must eventually
enter its critical section.

Proof. Assume that no processor ever enters its criti-
cal section. Then there will be some time after which no
more processors enter or leave the bakery. At this time,
assume that processor \(i\) has the minimum value of (num-
ber \([i]\), \(i\)) among all processors in the bakery. Then pro-
cessor \(i\) must eventually complete the for loop and enter
its critical section. This is the required contradiction. \(\square\)

Assertion 2 implies that at most one processor can be
in its critical section at any time. Assertions 1 and 2 prove
that processors enter their critical sections on a
first-come-first-served basis. Hence, an individual pro-
cessor cannot be blocked unless the entire system is
deadlocked. Assertion 3 implies that the system can only be
deadlocked by a processor halting in its critical section,
or by an unbounded sequence of processor failures and
re-entries. The latter can tie up the system as follows. If
processor \(j\) continually fails and restarts, then with bad
luck processor \(i\) could always find choosing \([j]\) = 1,
and loop forever at \(L_2\).

Further Remarks

If there is at least one processor in the bakery,
then the value of number \([i]\) can become arbitrarily large.
This problem cannot be solved by any simple scheme of
cycling through a finite set of integers. For example,
given any numbers \(r\) and \(s\), if \(N \geq 4\), then it is possible
to have simultaneously number \((i) = r\) and number
\((j) = s\) for some \(i\) and \(j\).

Fortunately, practical considerations will place an
upper bound on the value of number \([i]\) in any real ap-
lication. For example, if processors enter the doorway
at the rate of at most one per msec, then after a year of
operation we will have number \([i]\) < \(2^n\)—assuming that
a read of number \([i]\) can never obtain a value larger than
one which has been written there.

The unboundedness of number \([i]\) does raise an inter-
esting theoretical question: can one find an algorithm for
finite processors such that processors enter their critical
sections on a first-come-first-served basis, and no pro-
cessor may write into another processor’s memory? The
answer is not known.1

The algorithm can be generalized in two ways: (i)
under certain circumstances, to allow two processors si-
multaneously to be in their critical sections; and (ii) to
modify the first-come-first-served property so that
higher priority processors are served first. This will be
described in a future paper.

Conclusion

Our algorithm provides a new, simple solution to the
mutual exclusion problem. Since it does not depend
upon any form of central control, it is less sensitive to
component failure than previous solutions.

References

1. Knuth, D.E. Additional comments on a problem in concurrent
2. deBruijn, N.G. Additional comments on a problem in con-
current programming control Comm. ACM 10, 3 (Mar. 1967),
137-138.
3. Eisenberg, M.A., and McGuire, M.R. Further comments on
Dijkstra’s concurrent programming control problem. Comm.
ACM 15, 11 (Nov. 1972), 999.
4. Dijkstra, E.W. Solution of a problem in concurrent pro-
5. Dijkstra, E.W. The structure of THE multiprogramming sys-

Computer Systems

Erratum

In “A Note on Subexpression Ordering in the Evaluation
of Arithmetic Expressions” by Peter J. Denning and G. Scott
Graham, Comm. ACM 10, 11 (Nov. 1973), 700-702, the follow-
ing erratum has been submitted by Denning.

The first two sentences in the first full paragraph on
p. 701 should read as follows:

Hu shows that an optimal list \(L_0\) for any \(m\) and any
tree (of equal-execution-time tasks) can be constructed
by taking a first appearance of each task in the se-
quence \(M_1, M_2, \ldots, M_{xx}\). Ramamoorthy and
Gonzales order the tasks of each \(M_i\) according to
decreasing execution time, then construct a list \(L\) by tak-
ing the first appearance of each task in the sequence \(M_{11},
M_{11}, M_{12}, M_{12}, M_{23}, M_{23}, \ldots, M_{xx}, M_{xx}\); they claim that \(L\) is optimal for any tree and any \(m\).

It should be noted that even for equal-execution-time
tasks, a list constructed from the latter sequence above
need not be consistent with the former sequence above
and, hence, need not be optimal for that reason alone.

We are grateful to Dr. Shimon Even for calling our
unfortunately incorrect wording to our attention.