A New Generalization of Dekker's Algorithm for Mutual Exclusion

Alain J. Martin

Department of Computer Science
California Institute of Technology

5195:TR:85
A New Generalization of Dekker's Algorithm for Mutual Exclusion

Alain J. Martin

Computer Science Department
California Institute of Technology

5195:TR:85

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

© California Institute of Technology, 1985
A new generalization of Dekker's algorithm for mutual exclusion

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena CA 91125
Revised October 1985

1. Introduction

Dekker's algorithm [1] is the historically first solution to the mutual exclusion problem among two processes. The only two atomic actions allowed on shared variables are read and write actions on a single shared variable and no synchronization primitives are used. Dekker's solution for two processes and a generalization to an arbitrary number of processes have been presented and proved by Edsger W. Dijkstra in [1] and [2].

Dijkstra's original generalization is not "fair": in a fair solution, any process that requests the critical section will eventually get it. The standard fair solutions are Eisenberg and McGregor's [4], Lamport's [5], and Peterson's [6].

A new generalization of Dekker's solution is proposed. Although the solution is not fair, its simplicity compared to other solutions—for n processes, n Boolean and one bounded integer are used—makes it attractive for applications in which the shared resource is not heavily used.

2. Notation

We use Dijkstra's guarded commands [3] with a slightly different syntax: *[...]* and [...] stand for do ... od and if ... fi respectively. Moreover, *[S]* and *[B]* are simplifications for *[true -> S]* and *[B -> skip]* respectively. Given the semantics of the selection-command, which requires that at least one guard be true for the selection to terminate, the semantics of *[B]* can be interpreted as "wait until *B* holds". (An equivalent implementation of waiting is the busy wait *[¬B -> skip]*.)

3. The solution

The cyclic activity of a process is an alternation of a "non-critical section" NCS and a "critical section", CS. These two actions are further left unspecified apart from the fact that they both leave the control variables unchanged, that NCS need not terminate, and that CS is guaranteed to terminate.

In Dekker's solution, both processes p1 and p2 behave similarly. Process p1, for instance, "declares its interest" for the CS by setting the Boolean variable z1 to true. It then tests z2 to check whether p2 is also interested in the CS. If p2 is not, p1 enters its CS. If p2 is, p1 withdraws its candidacy by setting z1 to false and tries again. Because both processes have the same behavior, a subtle form of deadlock may occur—called "after-you-after-you blocking"—where both processes keep trying to enter and keep withdrawing at the same time. In order to exclude this possibility,
an additional shared variable $t$—for “turn”—equal to 1 or 2, is introduced to give priority to one of the two candidates. A process leaving the $CS$ changes the value of $t$.

Our solution for $n, n > 2$, processes is a straightforward generalization of Dekker’s algorithms. Process $p(i), 1 \leq i \leq n$, uses variable $x(i)$ to declare its interest to the $CS$ and checks whether other processes are interested by the test $(Ej : j \neq i : x(j))$. Variable $t$ is used in the same way as in Dekker’s solution. But a special value 0 has to be introduced in order to reset $t$ to a “neutral” value after completion of the $CS$. Hence the solution for an arbitrary process $p(i)$.

$$
p(i) \equiv \begin{cases} \ast[NCSi; \quad s(i) := \text{true}; \\
\ast[(Ej : j \neq i : x(j)) \rightarrow x(i) := \text{false}; \\
\; [t = 0 \lor t = i]; \\
\; t := t; \\
\; x(i) := \text{true} \\
\}; \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
with $Bk \equiv (\exists i : j \neq k : x(i))$ holding at some point between the two assignments to $x(k)$, and $t = k$ holding after the wait action. From the structure of $u(k)$, we see that

1. $x(k) = $ true holds only inside $u(k)$,
2. $t = k$ holds everywhere inside $u(k)$.

From (1) and (2): $x(k) \Rightarrow t = k$, i.e. since $\neg x(j)$ holds for all processes other than the actively blocked ones:

$$x(k) \Rightarrow \neg Bk.$$ 

Which contradicts $Bk$ holding at some point between the two assignments to $x(t)$.

\[ \square \]

References


January 1985
Revised, October 1985