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

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 McGuire'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 all other solutions—for n processes, n Booleans 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: $*[\ldots]$ and $[\ldots]$ stand for **do** ... **od** and **if** ... **fi** respectively. Moreover, $*[S]$ and $[B]$ are simplifications for $*[\textbf{true} \rightarrow S]$ and $[B \rightarrow \text{skip}]$ 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 $*[\neg B \rightarrow \text{skip}]$.)

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 $x1$ to **true**. It then tests $x2$ 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 $x1$ 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 $(\mathbf{E}j : j \neq i : x(j))$. 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 $p(i)$.

$$
\begin{array}{ll}
p(i) \equiv *[& NCSi; \\
& x(i) := \mathbf{true}; \\
& *[(\mathbf{E}j : j \neq i : x(j)) \rightarrow x(i) := \mathbf{false}; \\
& \qquad\qquad\qquad\qquad\quad [t = 0 \vee t = i]; \\
& \qquad\qquad\qquad\qquad\quad t := i; \\
& \qquad\qquad\qquad\qquad\quad x(i) := \mathbf{true} \\
&]; \\
& CSi; \\
& x(i), t := \mathbf{false}, 0 \\
&].
\end{array}
$$

Initially: $t = 0$, $(\mathbf{A}i : 1 \leq i \leq n : \neg x(i))$.

4. Proof of correctness

i) Mutual exclusion.
We introduce the ghost integer variable m, initialized to zero, and for each process $p(i)$, the ghost integer variable $c(i)$, initialized to zero. In the program, each action $x(i) := \mathbf{true}$ is replaced by the atomic sequence

$$X(i) : \langle x(i) := \mathbf{true}; c(i) := m;\ m := m + 1\rangle.$$

For each process $p(i)$, let us determine a precondition Pi of CSi. Since $x(i)$ holds as precondition of CSi, Pi contains the conjunct $x(i)$.

Let $L(i)$ be the last $X(i)$ executed before CSi. Such an $L(i)$ exists. By definition of the semantics of the repetition, for each j, $j \neq i$, $x(j)$ has been evaluated to **false** after $L(i)$. Hence, either $\neg x(j)$ holds as precondition of CSi or an $X(j)$ has been executed after $L(i)$, i.e., $c(j) > c(i)$ holds. Hence:

$$P(i) \equiv (x(i) \wedge (\mathbf{A}j : j \neq i : \neg x(j) \vee c(j) > c(i))).$$

For $i \neq \ell$, $(P(i) \wedge P(\ell)) \equiv \mathbf{false}$, which guarantees mutual exclusion. □

ii) Absence of deadlock.
A deadlock situation is one in which a non-empty subset of processes—the "blocked" processes—are inside their repetition $Ri : *[(\mathbf{E}j : j \neq i : x(j)) \rightarrow \ldots]$, Ri does not terminate, and all other processes are in NCS. Since initially $t = 0$ and any process leaving the CS performs $t := 0$, in a deadlock situation $t \neq k$ for any process $p(k)$ in NCS. In other words, in a deadlock situation $t = 0 \vee t = i$, where $p(i)$ is a blocked process. Hence not all blocked processes are blocked at the wait-action $[t = 0 \vee t = i]$.

Let $p(k)$ be "actively blocked" i.e. Rk does not terminate for $p(k)$, but $p(k)$ is not blocked at the wait-action. Starting from a deadlock state, any activity of $p(k)$ consists of an infinite repetition of the sequence $u(k)$ of actions:

$$t := k; x(k) := \mathbf{true}; \{Bk\}; x(k) := \mathbf{false}; [t = 0 \vee t = k]\{t = k\}$$

with $Bk \equiv (\mathbf{E}j : j \neq k : x(j))$ 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 $x(t)$.

□

References

[1] E.W. Dijkstra, Cooperating Sequential Processes, in: *Programming Languages*, (Academic Press, London, 1968).

[2] E.W. Dijkstra, Solution of a problem in concurrent programming control, *Communications of the ACM II* (3) (1968) 147-148.

[3] E.W.Dijkstra, *A Discipline of Programming*, (Prentice Hall, Englewood Cliffs, NJ 1976).

[4] M.A. Eisenberg and M.R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem," *Communications of the ACM*, 15, (11), (1972), 999.

[5] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, 17, (8), (1974), 453-455.

[6] G.L. Peterson, Myths about the mutual exclusion problem, *Information Processing Letters*, 12, (3) (1981) 115-116.

January 1985

Revised, October 1985