AN IMPLEMENTATION OF P AND V

Eric C.R. HEHNER
Computer Systems Research Group, University of Toronto, Toronto M5S 1A1, Canada

R.K. SHYAMASUNDAR
National Centre of Software Development and Computing Techniques, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, India

Received October 1980; revised version received March 1981

Synchronization, mutual exclusion, concurrency, P and V

1. Introduction

The semaphore operations P and V are used for synchronization and mutual exclusion of concurrent processes. An implementation of P and V is offered, with the following merits.

(1) It is very simple, and has a simple proof.

(2) No variable is written by more than one process; thus each variable can be associated with the process that writes it.

(3) If mutually exclusive access is granted to individual variables, then mutually exclusive access will be granted to critical regions properly enclosed by P and V operations. No other requirement of atomicity is made.

(4) The implementation cannot deadlock. This remains true even if some processes fail. Failed processes can be restarted outside their critical regions.

(5) If fair access is granted to individual variables, then fair access is granted to critical regions properly enclosed in P and V operations. No other queuing assumption is made.

The problem is artificial in the sense that we assume it to be already solved on one level as a basis for its solution on another; we assume non-deadlocking mutually exclusive and fair access is granted to individual variables in order to provide the same for critical regions of processes. The problem is non-trivial, however, and is something of a classic, being posed as early as 1962. Its first published solution is [1]. The present solution is offered both for its simplicity and for the style of presentation.

2. Implementation

The implementation requires, for each semaphore s, an array A of variables such that

(a) there is one element of A for each process;

(b) all elements of A can be read by all processes;

(c) only process i sets A[i];

(d) reading and writing an element of A are considered atomic; these are the only actions considered atomic;

(e) the values of elements of A are non-negative integers, adjoined by a special value ∞ such that $\forall i \in integers: i < \infty$;

(f) the initial values of A are ∞.

Number the processes arbitrarily from 1 to N. Then P(s) in the i^{th} process becomes

```
A[i] := 0;
A[i]:= 1 + FINITEMAX(A);
for j := 1 to N do
   while G(i, j) do skip od
od
```

and V(s) in the i^{th} process becomes

$A[i] := \infty$,

where j is a fresh local variable, and G (mnemonic for

0020-0190/81/0000–0000/$02.50

"greater than") is defined as

$$G(i,j) = (A[i] > A[j]) \vee (A[i] = A[j] \wedge i > j),$$

and *FINITEMAX*(A) is

```
fm := 0;
for j := 1 to N do
    aj := A[j];
    if fm < aj ∧ aj < ∞ then fm := aj fi
od
```

with result fm, where fm, j, and aj are local variables.

Our implementation of P and V is a version of Lamport's bakery algorithm [2], in which a customer wishing service takes a number to represent his place in a queue. Because "taking a number", i.e. the assignment "A[i] := 1 + *FINITEMAX*(A)" is not a mutually exclusive activity, it is possible for two processes to take the same number; in that case, the lower-numbered process is deemed to have priority. The boolean G(i, j) has the informal meaning "process j currently has priority over process i".

3. Verification

Our proof technique is that of Gries and Owicki [3].

To prove that this provides mutually exclusive access to critical regions, we introduce auxiliary boolean variables B[m, n] such that

$$I: (\forall m, n) \quad B[m, n] \vee \neg G(m, n) \vee A[n] = 0$$

is invariantly true. The B[m, n] are initially true. Now "P(s); *CRITICAL-REGION*; V(s)" becomes

```
(1)  A[i] := 0;
(2)  ai := 1 + FINITEMAX(A);
(3)  A[i] := ai;
(4)  for j := 1 to N do
(5)      while G(i, j) do skip od;
(6)      B[i, j] := false
(7)  od;
(8)  CRITICAL-REGION;
(9)  for j := 1 to N do
(10)     B[i, j] := true
(11) od;
(12) A[i] := ∞.
```

Note. Because the B[i, j] are only assigned and never used, they may be removed without changing the values of variables or flow of control. Therefore this program is equivalent to the former program.

To prove that I is invariant, we must check that it is true initially (obvious), and that assignments to A and B leave it true.

Line 1 establishes A[i] = 0. From the definition of G we see that it also establishes $(\forall j)\ \neg G(i, j) \vee A[j] = 0$. Checking the two cases m = i and n = i we see that I is maintained.

Line 3 increases A[i] at a time when $(\forall j)$ B[i, j]. For m = i, it is the latter fact that maintains I; for n = i it is the former fact, since G(m, n) cannot be changed from false to true.

On line 6, consider that "B[i, j] := *false*" is performed at the same instant that G(i, j) is discovered to be false on line 5. Then it is obvious that I is maintained.

Note. It does not matter that we have no mechanism to perform the assignment at that instant, since B[i, j] does not appear in the original program.

Line 10 obviously maintains I.

For line 12, for m = i we have B[m, n] (from Lines 9–11), and for n = i we have ¬G(m, n).

We now prove mutual exclusion from critical regions. If process i is in its critical region, then $A[i] \neq 0 \wedge (\forall j)\ \neg B[i, j]$. Choosing a particular j, we may say $A[i] \neq 0 \wedge \neg B[i, j]$. If this process j is also in its critical region, we have $A[j] \neq 0 \wedge \neg B[j, i]$. Rearranging, $(\neg B[i, j] \wedge A[j] \neq 0) \wedge (\neg B[j, i] \wedge A[i] \neq 0)$. Using the invariant I with each of the major conjuncts, we find $\neg G(i, j) \wedge \neg G(j, i)$. But because G is antisymmetric, we must conclude i = j. Therefore two different processes cannot both be in their critical regions at the same time.

We now prove that there is no deadlock. Process i can be delayed only by the truth of G(i, j). But because G is transitive and irreflexive, no cycle of delayed processes can exist. Therefore there is no deadlock. The failure of process i will not deadlock the system if the following are enforced;

(a) upon failure, A[i] := ∞ is executed;

(b) during down-time, A[i] must remain accessible to other processes;

(c) process i must not be restarted within a sequence "P(s); *CRITICAL-REGION*; V(s)".

We now prove that this implementation of P and V is fair. This means that no process j can enter a

critical region twice while a process i waits to enter a critical region once. To suppose the contrary is to suppose that

(a) A[i] is finite and unchanging;

(b) at least once, "A[j] := 1 + FINITEMAX(A)" is executed and subsequently G(j, i) is found to be false. The contradiction is apparent.

4. Discussion

We have proven that our implementation has the merits listed in the Introduction. Our implementation also has the following demerits.

(1) There is no bound on the values of the elements of A. Whenever there is no process in or waiting to enter its critical region, the values of the elements of A for subsequent processes entering critical regions will begin again at 1; in practice then, this demerit may not be a problem. But there is no guarantee against an arbitrarily long run of values. The second author has an implementation without this demerit, giving up only a little of merits 1 and 3 (expression evaluation must be atomic to achieve mutual exclusion).

(2) Process i performing P(s) can be delayed by process j performing P(s) even if process i precedes process j. This occurs when process j is computing the time-consuming function *FINITEMAX*. This demerit can be lessened, at the expense of merit 2, by introducing global variable CTR, initially 0, and replacing "A[i] := 1 + *FINITEMAX*(A)" by "CTR := CTR + 1; A[i] := CTR".

The number of processes can change, as long as an unused process number i satisfies A[i] = ∞, exactly as for a failed process. Alternatively, the algorithm can be modified from "**for** j := 1 **to** N" to "**for** j ∈ {current process numbers}" if a means of implementing this change is available.

Acknowledgement

We thank Nigel Horspool, Edsger Dijkstra and David Gries for pointing out an error, and Jim Horning for criticism.

References

[1] E.W. Dijkstra, Solution of a problem in concurrent programming control, Comm. ACM 8 (9) (1965).

[2] L. Lamport, A new solution of Dijkstra's concurrent programming problem, Comm. ACM 17 (8) (1974) 453–455.

[3] S.S. Owicki and D. Gries, An axiomatic proof technique for parallel programs, Acta Inform. 6 (1976) 319–340.

Addendum

After submission of this paper, a paper by Ricart and Agrawala titled "An Optimal Algorithm for Mutual Exclusion in Computer Networks" appeared in *Comm. ACM* 24 (1) (1981). Their paper contains an algorithm similar to the one suggested in point 2 of our Discussion, but using network terminology. In contrast to our proof, their "proof" is long and incomplete; they argue from examples of possible situations, and diagrams with three processes. They do, however, make points that we do not, particularly about optimality.