ALTERNATE SOLUTIONS TO THE CIGARETTE SMOKERS' PROBLEM WITHOUT CONDITIONALS

S.S. REDDI
Rice University, Houston, Texas 77001, U.S.A.

Received 31 May 1976, revised version received November 1976

Cigarette smoker's problem, no-conditionals case, semaphore, primitive, Petri net

Parnas in a recent paper [1] proposes a solution for Patil's Cigarette Smokers' Problem (CSP) without using conditional branches. By using semaphore arrays and performing arithmetic operations on the indices of these arrays he obtains the effect of the generalized primitives of Patil. (This method has been further generalized in [2] which gives procedures for realizing conflict free Petri nets without conditional statements.) In this paper we present two alternate solutions which do not use conditional branches and are conceptually different from that of Parnas. The first solution (see Appendix), uses a primitive called FORK which creates independent subprocesses whereas the second employs primitive D which decreases its argument by 1 and hence is very similar to the conventional V primitive. In the third solution (see Appendix), we introduce a primitive which tests the value of a semaphore.

FORK is used as follows. Consider

```
A.  FORK(A1, A2)
A1: ....        A2: ....
    ....            ....
    go to A         ....
                    go to A
```

Process A creates independent subprocesses $A1$ and $A2$. When a go to A statement is encountered in one of the subprocesses, control of the whole process is transferred to A regardless of the status of the other subprocess. Implementation of FORK involves changing the instruction counters of the subprocesses appropriately when go to statements are encountered. Solution 1 is based on a Petri net representation of the 2 out of 3 net (fig. 1). Counting each subprocess as a process we see Solution 1 has only five processes whereas Parnas's solution has six (not counting overflow preventing processes).

Solution 2 (see Appendix), uses $D(S)$ which decreases semaphore S by 1 and so is as simple to implement as V. The solution works as follows. Let semaphores a and b become 1. Then process AB proceeds and decrements semaphores y and z and increments x by 1. BC and CA are blocked at second and first statements respectively. Once x becomes 1, PAB starts and performs the required "smoking" operation. After the completion of the operation, $c1$ and $c2$ are incremented by 1 so that BC and CA can proceed. Though BC and CA increment y and z respectively

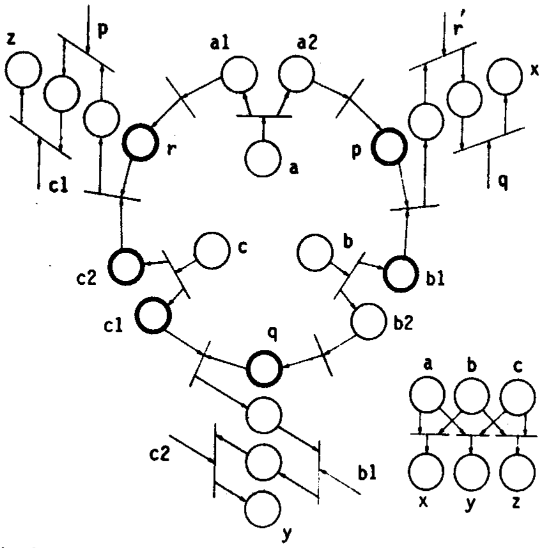

Fig. 1. Petri net representation of a 2 out of 3 net on which the first solution is based. Double circles represent shared plces.

they do not become 1 because *AB* decremented them before. When *ab, bc* and *ca* are incremented to 1, *x, y* and *z* will have values of −2, −1 and −1 respectively. *PAB* increments and makes all of them zero. It then signals to the agent completion of the "smoking" operation by performing $V(Z)$. Note *y* and *z* never become 1 to enable *PBC* and *PCA*.

In the third solution we use the primitive $T(s\ R\ i)$ $\{S1; S2; ...; Sn\}$ which when executed has the following effect. If the value of semaphore *s* and integer *i* satisfy relation *R*, statement sequence *S*1, *S*2, ..., *Sn* is executed; otherwise control is transferred to the next statement which occurs after $T(s\ R\ i)$ $\{S1; S2; ...; Sn\}$. To implement the primitive the value of the semaphore is transferred to a central register and tested whether it satisfies the relation. If the relation holds all the statements within the braces are executed. (Thus the value of the semaphore is irrelevant after it has been transferred to the central register.) It is possible to eliminate processes *A, B* and *C* is Solution 3 by suitably modifying the agent.

Operation *T* can be used sometimes to eliminate "busy waiting" (e.g., consider a processor executing the program:

```
A: T(s1 = 0) {go to B}
   P(s1)
   ....
   V(s1)
B: T(s2 = 0) {go to C}
   P(s2)
   .....
   V(s2)
```

By using *T* one avoids the situation where a processor is tied up waiting for *s*1 or *s*2 to become 1 and increases processor utilization). The operation of testing the value of a semaphore for zero can also be used to solve the synchronization problems posed by Kosaraju [3] who shows that they cannot be handled by *P* and *V* primitives.

It should be noted that Solutions 1 and 2 do not use conditional statements but Solution 3 uses *T* which is essentially a conditional. Reduction of conditionals in a program may be advantageous in a computer system with instruction look ahead circuitry.

We conclude the paper by noting that *D* primitive indeed adds power to semaphores and semaphore arrays. Kosaraju [3] proves the following: Let there be two consumers *C*1, *C*2, two buffers *B*1, *B*2 and two producers *P*1, *P*2. *Pi* produces an item and deposits in *Bi*; *Ci* consumes items from *Bi* one at a time. It is illegal to consume an empty buffer. *C*1 and *C*2 cannot consume simultaneously and *C*1 has priority over *C*2. (Thus *C*2 consumes only when *B*1 is empty.) Then

(1) It is not possible to solve the above synchronization problem using *only* semaphores and *P* and *V* primitives.

(2) It is not possible to solve the above problem using *only* a finite, bounded number of semaphores and semaphore arrays, *P* and *V* primitives and operations on array indices. It is assumed that a table specifies the effect of each operation on the array indices.

Thus in both instances it is not permissible to use integer variables and conditionals. Solution *K* shows that using only *P, V, D* and semaphores the above problem can be solved.

References

[1] D.L. Parnas, On a solution to the cigarette smokers' problem (without conditional statements), CACM, 18 (3) (1975) 181–183.

[2] R. Devillers and G. Louchard, Realization of petri nets without conditional statements, Information Processing Lett. 2 (4) (1973) 105–107.

[3] S. Rao Kosaraju, Limitations of Dijkstra's semaphore primitives and petri nets, Tech. Rept., Dept. of Elec. Eng., Johns Hopkins U., Baltimore.

Appendix

```
r_a: P(s)          r_b: P(s)          r_c: P(s)
     V(b)               V(a)               V(a)
     V(c)               V(c)               V(b)
     go to r_a          go to r_b          go to r_c

β_x: P(X)          β_y: P(Y)          β_z: P(Z)
     V(s)               V(s)               V(s)
     go to β_x          go to β_y          go to β_z
```

Initially *s* = 1 and *a, b, c, X, Y* and *Z* = 0.

Solution 1:

semaphore *a*1, *a*2, *b*1, *b*2, *c*1, *c*2, *p, q, r* (all initially 0) semaphore *re* (initially 0)

```
α: P(a)                β: P(b)                γ: P(c)
   V(a1)                  V(b1)                  V(c1)
   V(a2)                  V(b2)                  V(c2)
   FORK(α1, α2)           FORK(β1, β2)           go to γ

α1: P(a1)    α2: P(a2)    β1: P(re)    β2: P(b2)
    V(r)         V(p)         :            V(q)
    P(c2)        P(b1)        V(Z)         P(c1)
    P(r)         P(p)         go to β      P(q)
    P(p)         P(r)                      P(b1)
    P(c1)        P(q)                      P(c2)
    :            V(re)                     :
    V(Y)         go to α                   V(X)
    go to α                                go to β
```

Solution 2:

semaphore *a*1, *a*2, *b*1, *b*2, *x, y, z, ab, bc, ca* (all initially 0)

```
A:   P(a)        B:  P(b)        C:   P(c)
     V(a1)           V(b1)            V(c1)
     V(a2)           V(b2)            V(c2)
     go to A         go to B          go to C

AB:  P(a1)       BC: P(b1)       CA:  P(c1)
     P(b2)           P(c2)            P(a2)
     D(y)            D(x)             D(x)
     D(z)            D(z)             D(y)
     V(x)            V(y)             V(z)
     V(ab)           V(bc)            V(ca)
     go to AB        go to BC         go to CA

PAB: P(x)        PBC: P(y)       PCA: P(z)
     :                :               :
     V(c1)            V(a1)           V(b1)
     V(c2)            V(a2)           V(b2)
     P(ab)            P(ab)           P(ab)
     P(bc)            P(bc)           P(bc)
     P(ca)            P(ca)           P(ca)
     V(x)             V(y)            V(z)
     V(x)             V(y)            V(z)
     V(y)             V(x)            V(x)
     V(z)             V(z)            V(y)
     V(Z)             V(X)            V(Y)
     go to PAB        go to PBC       go to PCA
```

Solution 3:

semaphore *a*1, *b*1, *c*1, *S* (all initially 0)

```
A:   P(a)         B: P(b)          C: P(c)
     V(a1)           V(b1)            V(c1)
     V(S)            V(S)             V(S)
     go to A         go to B          go to C

ABC: P(S)
     P(S)
     T(a1 = 0) {...; P(b1); P(c1); V(X); go to ABC}
     T(b1 = 0) {...; P(a1); P(c1); V(Y); go to ABC}
     T(c1 = 0) {...; P(a1); P(b1); V(Z); go to ABC}
```

Solution K:

semaphore *p*1, *p*2, *b*1, *b*2 (all initially 0)
semaphore *b, r* (all initially 1)

```
P1   A: P(p1)         P2   C: P(p2)
        D(b)                  V(b2)
        V(b1)                 go to C
        go to A

C1   B: P(b1)         C2   D: P(b2)
        P(r)                  P(b)
        .                     P(r)
        .                     .
        V(r)                  .
        V(b)                  V(r)
        go to B               V(b)
                              go to D
```