Copyright Notice

On two beautiful solutions designed by Martin Rem.

(In recent correspondence with dr.Martin Rem --currently at the Department of Computer Science (mail code: 256-80), California Institute of Technology, PASADENA, California 91109, U.S.A.-- he sent me two solutions which I think both so beautiful, that they deserve a wider distribution; hence their inclusion in the EWD-series; apart from some historical information and formal elaborations that have been added, and some cosmetic changes, I have essentially presented Rem's solutions.)

A P/V-implementation of conditional critical regions.

Since (by an accident of history) the P- and V-operations on semaphores have more or less acquired the status of "canonical" synchronization primitives, inventors of new synchronization concepts have related their inventions to P- and V-operations in two different ways. Either --see, for instance, Hoare[1], concerning monitors-- the new concept is shown to be equally powerful by demonstrating that it can be used to implement the P-and V-operations; or --see, for instance, Hoare [2] , when introducing the (simple) critical region "with r do S od"-- the feasibility of its implementation is argued by showing how to implement it with P- and V-operations. The latter possibility has now been demonstrated by Rem for the conditional critical region "with r when B do S od" as well. (In [2] , Hoare remarks about the simple critical region "If we assume that a Boolean semaphore mechanism is "built-in", the implementation is trivial." (as indeed it is). When in [2] Hoare introduces the conditional critical regions, he adds "Some care must be exercised in the implementation of this new feature." and follows with a two paragraph verbal sketch, explaining what has to be done with a queue of processes waiting for r . In [3] , Brinch Hansen gives a slightly more detailed sketch of an implementation involving two queues --"queues" that can be recognized in Rem's solution (if looked at abstractly enough)-- but it is still no more than a sketch. Ironically enough, Rem now solves the problem by a method --later called "splitting a binary semaphore"-- that a few years ago.... Hoare has taught us!)

In processes so-called "conditional critical regions" may occurr of the form

"with r when Bi do Si od" .

Here r denotes a shared variable --or more generally: a cluster of shared

variables-- , such that r is only accessible from within sections of the text of the form "when Bi do Si od" that are prefixed by "with r". (That this constraint is not violated is easily checked by a compiler, a circumstance that is its major justification.)

As with the simple critical regions "with r do Si od", the implementation has to ensure that the executions of the statements Si --prefixed by the same "with r"-- as they may occur in the different processes, exclude each other in time. In addition, a statement Si , --like what later would become known as "a guarded command"-- is only eligible for execution in those initial states where Bi holds. The implementation has to ensure that these constraints are met by delaying, if necessary, the further execution of the process in which Si occurs.

A further requirement is that no such delay occurs without justification, more precisely:

1) if no statement Si is under execution --i.e. the requirement of mutual exclusion would not constrain the selection of a next Si for execution--,and

2) if for one or more processes the Si of a conditional critical region is the next statement to be executed and at least one of the corresponding Bi's is true,

then the selection of such an Si with a true Bi is obligatory.

To make the implementation of this last requirement feasible, a further constraint ensures that activity of one process, but well outside its regions critical with respect to r leaves the "non Bi" for all other processes invariantly true. This further constraint is that r is the only shared variable Bi may depend upon. The whole set of constraints now ensures that the obligation to inspect whether a false Bi of a delayed process has turned true, can be concentrated at the point where the execution of an Sj (of another process!) has been carried to completion.

The technique of the "split binary semaphore" consists of the introduction of a set of binary semaphores --in this example of the three semaphores m , b1 , and b2 -- of which at most one equals 1 . This can obviously be ensured by seeing to it that in each program P- and V-operations --regardless of on which

of the three semaphores they operate-- alternate dynamically: each P-operation decreases their sum by 1 , each V-operation increases their sum by 1 . Furthermore we can assert that between each P-operation and dynamically subsequent V-operation the sum m + b1 + b2 = 0 ; hence the executions of the program sections between such a P-operation and its subsequent V-operation can be viewed as excluding each other mutually in time (if so desired by the traditional argument of Dijkstra [4]).

Rem's solution uses three semaphores m (=1), b1(=0), and b2(=0), and two counters n(=0), and nt(=0) --initial values being given between parentheses-- . The integer n counts the number of processes "eager" to perform their Si's ; during testing, the counter nt is equal to the number of Bi's , the falsity of which is not guaranteed. The whole critical activity can only end with nt = 0 --otherwise impermissible delays could result-- . When an Si has been performed --and, therefore, all Bi may have become true-- nt has to be increased until nt = n before testing can begin. In this latter process the semaphore b1 plays a signalling role; the semaphore b2 is used to admit processes to their Bi-test one at a time. With this informal sketch of meaning and function of the semaphores and variables I shall present Rem's solution without further annotation; thereafter I shall present a more formal treatment.

```
P(m); n:= n + 1;
do non Bi → if nt = 0 → V(m) [] nt > 0 → V(b2) fi;
            P(b1); nt:= nt + 1;
            if nt < n → V(b1) [] nt = n → V(b2) fi;
            P(b2); nt:= nt - 1
od;
n:= n - 1;  Si;
if n = 0 → V(m)
 [] n > 0 → if nt < n → V(b1) [] nt = n → V(b2) fi
fi
```

For our more formal treatment we introduce angle brackets in order to indicate that each action extending from an opening bracket until a next (closing) angle bracket denotes an atomic action. Atomic actions can be viewed as excluding each other in time. This is OK if each atomic action starts with a P-operation, ends with a V-operation and has no such operations in between.

For each process we introduce two boolean ghost-variables ai ("in the antichambre") and wi ("in the waitingroom"). They are initially false; we shall use the notations (N j: aj) and (N j: wj) respectively to denote the number of processes for which ai and wi respectively are true. Furthermore we introduce a global ghost-boolean c --initally false-- , the truth of which marks the states in which the implications aj => non Bj need not hold. Labels have been inserted for later discussion. The annotated text of the program is as follows:

```
L0: < P(m) {non c and 0 = nt ≤ n}; n:= n + 1 {non c and 0 ≤ nt < n};
    do non Bi → {non c and 0 ≤ nt < n and non Bi} ai:= true;
                if nt = 0 → {non c and 0 = nt ≤ n} V(m)
                [] nt > 0 → {non c and 0 < nt ≤ n} V(b2)
                fi >;
          L1: < P(b1) {c and 0 ≤ nt < n}; ai:= false; wi:= true;
                nt:= nt + 1  { c and 0 < nt ≤ n};
                if nt < n → { c and 0 ≤ nt < n} V(b1)
                 [] nt = n → c:= false; {non c and 0 < nt ≤ n} V(b2)
                fi >;
          L2: < P(b2) {non c and 0 < nt ≤ n}; wi:= false;
                nt:= nt - 1 {non c and 0 ≤ nt < n}
    od;
    n:= n - 1 {Bi and 0 ≤ nt ≤ n};
    Si; c:= (nt < n);
    if n = 0 → {non c and 0 = nt ≤ n} V(m)
     [] n > 0 → if nt < n → {c and 0 ≤ nt < n} V(b1)
                 [] nt = n → {non c and 0 < nt ≤ n} V(b2)
                fi
    fi >
L3:
```

Indicating atomic actions by start- and end-label, we can denote the five atomic actions we have to consider as follows: L0-L1, L0-L3, L1-L2, L2-L1, and L2-L3. With the initialization m = 1 , b1 = b2 = 0 , we readily establish for all five the invariance of

P0: m + b1 + b2 = 1 .

This establishes the property of the "split boolean semaphore" and tells us that, indeed, we are entitled to regard the five actions --each of which starts with a P-operation on one of the three semaphores and ends (dynamically) with a V-operation on one of the semaphores-- as "atomic". In particular it guarantees that the Si are executed under mutual exclusion and under the initial truth of Bi .

Having established the atomicity, and taking the further initial values nt = n = 0 and c = false into account, we next establish the invariant truth of

P1: $(m = 1 \Rightarrow (\underline{non}\ c\ \underline{and}\ 0 = nt \leq n))$ and
$(b1 = 1 \Rightarrow (c\ \underline{and}\ 0 \leq nt < n))$ and
$(b2 = 1 \Rightarrow (\underline{non}\ c\ \underline{and}\ 0 < nt \leq n))$

The invariance of P1 is easily established, as is indicated by the assertions that annotate the program text. (Note that it seems to be the function of the ghost-boolean c to make the three consequents mutually exclusive.)

With the further knowledge that initially all the wi are false, we easily establish the invariant truth of

P2: $(\underline{N}\ j:\ wj) = nt$.

Because $(\underline{N}\ j:\ wj)$ = the number of processes at L2, ready to perform P(b2) , we conclude now that on account of the third implication of P1 , a deadlock cannot occur after the execution of V(b2) .

With the further knowledge that initially all the ai are false, we easily establish the invariant truth of

P3: $(\underline{N}\ j:\ aj) = n - nt$.

Because $(\underline{N}\ j:\ aj)$ = the number of processes at L1, ready to perform P(b1) , we conclude now that on account of the second implication of P1 , a deadlock cannot occur after the execution of V(b1) .

(A "temporary" or "partial" deadlock can occur after the execution of V(m) ; then, however, the state m = 1 holds, and the assumption is that sooner or later another process will "join the game" via L0 .)

Finally we establish the invariant truth of

P4: (A j: aj => (non Bj or c)) ,

which holds initially because then all antecedents are false. We shall check its invariance explicitly.

L0-L3 and L2-L3 could make all Bj's true as a result of Si's modification of r ; the assignment c:= (nt < n) , however, makes all implications of P4 hold: if c is established by it, all consequents are true, if non c is established by it, we conclude nt = n , and P3 then tells us, that all antecedents are false; in both cases all implications of P4 hold vacuously.

L0-L1 and L2-L1 could only affect the i'th implication, but they don't do so as ai:= true is executed under the truth of its consequent, viz. non Bi.

In L1-L2 , the assignment ai:= false strengthens an antecedent, and therefore is safe; the assignment c:= false may strengthen any consequent, but --see P3-- is executed under falsity of all antecedents and, therefore, is safe as well. This concludes our demonstration of the invariance of P4 .

Combining (the first implication of) P1, P3, and P4 we conclude

m = 1 => ((N j: aj) = n and (A j: aj => non Bj)) ,

thus expressing that no avoidable delay is introduced.

* * *

[1] Hoare, C.A.R. "Monitors: an Operating System Structuring Concept", STAN-CS-73-401, November 1973

[2] Hoare, C.A.R. "Towards a Theory of Parallel Programming", in Operating Systems Techniques, C.A.R.Hoare and R.H.Perrott (Eds.) London and New York, Academic Press, 1972

[3] Brinch Hansen, Per, "Operating System Principles", Englewood Cliffs, Prentice-Hall, 1973

[4] Dijkstra, Edsger W., "Hierarchical Ordering of Sequential Processes" in Operating System Techniques, C.A.R.Hoare and R.H.Perrott (Eds.) London and New York, Academic Press, 1972

Note. I have changed my mind and postpone the other solution's presentation to a later EWD-report. (End of note.)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow