A tutorial on the split binary semaphore.

The purpose of this note is threefold. It has been written to introduce the student to

1) the technique of the so-called "split binary semaphore" --originally discovered, but at the same time not recommended, by C.A.R.Hoare--

2) the use of formal techniques in the development of multiprograms

3) the now canonical example of "The readers and the writers", which the student must know anyhow; it will be used as a carrier for the other two purposes. The problem of the readers and the writers was designed by D.L. Parnas.

We consider two classes of cyclic processes, called "readers" and "writers" respectively. With "ncs" standing for "non-critical section", they can be described by the programs

```
reader:   do true → ncs; READ od                                (1)
writer:   do true → ncs; WRITE od
```

respectively. Here "READ" and "WRITE" denote their respective critical sections, critical in the sense that when a writer is engaged in its critical section, it must be the only process engaged in its critical section. This problem can be solved in many different ways --the problem is canonical in the sense that everybody proposing new synchronization primitives has solved it in his way-- , we shall now solve it using a split binary semaphore.

Our first step is the introduction of variables, in terms of which we express our synchronization requirement formally; we call them "ar" and "aw" --short for "number of active readers" and "number of active writers" respectively-- and consider the following multiprogram:

```
initial state:   ar = 0, aw = 0, mutex = 1                      (2)
```

[cont.(2)]

```
reader:  do true → ncs;
                  P(mutex); ar:= ar + 1; V(mutex);
                  READ;
                  P(mutex); ar:= ar - 1; V(mutex)
         od
writer:  do true → ncs;
                  P(mutex); aw:= aw + 1; V(mutex);
                  WRITE;
                  P(mutex); aw:= aw - 1; V(mutex)
         od
```

For multiprogram (2) the invariance of

P0: $ar \geq 0$ and $aw \geq 0$

follows immediately in the usual way "from the topology of the program". (Associate with $reader_i$ an additional variable c_i, initially = 0, and satisfying $0 \leq c_i \leq 1$; replace ar:= ar + 1 by c_i, ar := 1, ar + 1 and ar:= ar - 1 by c_i, ar := 0, ar - 1 and observe the invariance of $ar = \sum c_i$, etc.)

Note. Looking at program (2) and interpreting it operationally, one might be tempted to say: clearly $ar \geq$ the number of readers engaged in READ , and as "a number of readers" is never negative, $ar \geq 0$ follows immediately, and similarly for $aw \geq 0$. I prefer not to do so, to prove the invariance of P0 from the uninterpreted text (2), and to conclude from P0 that it does not prohibit the interpretation of ar and aw as natural numbers. (End of Note.)

In terms of ar and aw I propose the required invariance of

P1: $aw = 0$ or ($aw = 1$ and $ar = 0$)

as a suitable formal expression of our operational requirement "when a writer is engaged in its critical section, it must be the only process engaged in its critical section". The term $aw = 0$ is intended to cover the situation when no writer is engaged in its critical section, the term $aw = 1$ and $ar = 0$

should cover the case when a writer is engaged in its critical section.

Note. The attentive reader will already have decided that he should not expect more eloquence from me on the "suitability" of the proposal captured by P1 . (End of Note.)

In order to prevent $ar:= ar + 1$ from violating P1 we can make it into a guarded command of the form

$$wp(\text{"}ar:= ar + 1\text{"}, P1) \rightarrow ar:= ar + 1 \quad .$$

The axiom of assignment gives for this guard

$$aw = 0 \ \underline{\text{or}}\ (aw = 1 \ \underline{\text{and}}\ ar + 1 = 0)$$

but on account of P0 the second term is false, and we simplify

$$aw = 0 \rightarrow ar:= ar + 1 \quad .$$

In order to prevent $aw:= aw + 1$ from violating P1 we consider

$$wp(\text{"}aw:= aw + 1\text{"}, P1) \rightarrow aw:= aw + 1 \quad .$$

The axiom of assignment gives for this guard

$$aw + 1 = 0 \ \underline{\text{or}}\ (aw + 1 = 1 \ \underline{\text{and}}\ ar = 0)$$

but, again, P0 admits simplification of this guard --its first term is false-- and we find

$$aw = 0 \ \underline{\text{and}}\ ar = 0 \rightarrow aw:= aw + 1 \quad .$$

The decrease $ar:= ar - 1$ is similarly guarded

$$wp(\text{"}ar:= ar - 1\text{"}, P1) \rightarrow ar:= ar - 1 \quad ,$$

and we get for the guard with the axiom of assignment

$$aw = 0 \ \underline{\text{or}}\ (aw = 1 \ \underline{\text{and}}\ ar - 1 = 0) \quad ,$$

which, thanks to P1 'can be simplified as

$$aw = 0 \rightarrow ar:= ar - 1 \quad .$$

The known invariance of P0 tells us that the precondition of $ar:= ar - 1$

implies wp("ar:= ar - 1", P0) , i.e. implies $ar - 1 \geq 0$; on account of P1 this implies $aw = 0$, and, therefore, we can simplify to

```
        true → ar:= ar - 1
```

i.e. the decrease ar:= ar - 1 need not be guarded at all.

The verification that also aw:= aw - 1 need not be guarded is left to the reader.

Inserting the guards as derived we get

```
initial state:  ar = 0, aw = 0, mutex = 1                              (3)

reader:  do true → ncs;
§                    P(mutex); if aw = 0 → ar:= ar + 1 fi; V(mutex);
                     READ;
                     P(mutex); ar:= ar - 1; V(mutex)
         od

writer:  do true → ncs;
§§                   P(mutex); if aw = 0 and ar = 0 → aw:= aw + 1 fi; V(mutex);
                     WRITE;
                     P(mutex); aw:= aw - 1; V(mutex)
         od
```

Multiprogram (3) has been designed so as to leave P0 and P1 invariant, and that is fine. It has, however, a major drawback: both alternative constructs, in the lines marked § and §§ respectively, may lead to abortion! The so-called split binary semaphore provides us with a technique for preventing the selection of the critical sections marked § and §§ respectively under those circumstances in which they would lead to abortion.

We replace the single binary semaphore mutex by three, also binary, semaphores --m , r , and w say-- , related to the original mutex by

$$mutex = m + r + w \quad ,$$

and replace in multiprogram (3) each P(mutex) by P(m), P(r), or P(w)

--the three ways of decreasing mutex by 1-- and each V(mutex) by V(m), V(r), or V(w) --the three ways of increasing mutex by 1-- . (Because m, r, and w are semaphores, we thus guarantee $0 \leq \text{mutex} \leq 1$.) More precisely: we replace the P(mutex) marked by § by P(r) , replace the P(mutex) marked by §§ by P(w) , and the P(mutex) that opens a critical section that cannot lead to abortion by P(m) .

We can now avoid selection of an aborting critical section by guarding V(r) by aw = 0 and by guarding V(w) by aw = 0 _and_ ar = 0 , because the precondition of a V-operation on a component of a split binary semaphore can be taken as the postcondition of the corresponding P-operation. Our analysis so far would suggest that it suffices to replace V(mutex) everytime by

```
Q:     if true → V(m) ▯ aw = 0 → V(r) ▯ aw = 0 and ar = 0 → V(w) fi     (4)
```

This, however, is too naive. To start with: how do we translate the initialization mutex = 1 ? The initialization m = 0, r = 1, w = 0 is too restrictive: if all readers remain in their ncs , no writer could perform WRITE. The initialization m = 0 , r = 0, w = 1 has to be rejected on similar grounds. In order to make the only remaining possible initialization m = 1, r = 0, w = 0 acceptable, readers and writers should encounter as first P-operation one that cannot lead to abortion. We can satisfy this requirement by inserting in both readers and writers of (3) after ncs

```
        P(mutex); V(mutex);
```

before performing the substitution described above. This would lead to the following multiprogram

```
initial state:  ar = 0, aw = 0, m = 1, r = 0, w = 0                    (5)

reader:  do true → ncs; P(m); Q;
                   P(r) {aw = 0}; ar:= ar + 1; Q;
                   READ;
                   P(m); ar:= ar - 1; Q
         od
```

```
writer:  do true → ncs; P(m); Q;                                   [cont.(5)]
                   P(w) {aw = 0 and ar = 0}; aw:= aw + 1; Q;
                   WRITE;
                   P(m); aw:= aw - 1; Q
         od
```

Multiprogram (5) is, however, still too naive: the non-determinacy, that has been introduced by Q as given in (4), has lead to a system with the danger of deadlock. The recipe for its prevention is, however, universal.

1) At initialization and at each V-operation, the "type of the next P-operation" --i.e. the component of the split binary semaphore on which this program will perform its next P-operation-- must be uniquely defined.
Our program (5) satisfies this condition, as do all programs without initial non-determinacy nor non-determinacy between a V-operation and the dynamically next P-operation. If a program does not satisfy this condition we can always make it satisfying it by introducing one or more extra components of the split binary semaphore, and by replacing essentially

```
      if true → P(component 1);...
       [] true → P(component 2);...
      fi
```

by

```
      if true → P(extra component); Q; P(component 1);...
       [] true → P(extra component); Q; P(component 2);...
      fi
```

<u>Task</u>. Prove that for this substitution process only a finite number of different extra components is needed. (End of Task.)

2) With each component of the binary semaphore we associate a counter, initialized to the number of processes for which the first P-operation is of the corresponding type.

3) After each P-operation we insert a decrease by 1 of the counter associated with its type.

4) Before each Q we insert an increase by 1 of the counter associated with the type of the dynamically next P-operation. (Thanks to step 1 , this is a well-defined counter.)

Note. For the operationally minded: each counter can be interpreted as the number of processes "ready" or "headed" for a P-operation on the corresponding component. (End of Note.)

5) Strengthen in Q the guarding of each V(component) by the requirement that the corresponding counter is positive.

Associating with the semaphore components m , r , and w the counters bm , br , and bw respectively, carrying out the above transformations on program (5) leads to

```
initial state:  ar = 0, aw = 0, bm = number of processes,  br = 0, bw = 0   (6)
                m = 1, r = 0, w = 0

reader:  do true → ncs;
                P(m); bm:= bm - 1; br:= br + 1; Q;
                P(r); br:= br - 1; ar:= ar + 1; bm:= bm + 1; Q;
                READ;
                P(m); bm:= bm - 1; ar:= ar - 1; bm:= bm + 1; Q
         od

writer:  do true → ncs;
                P(m); bm:= bm - 1; bw:= bw + 1; Q;
                P(w); bw:= bw - 1; aw:= aw + 1; bm:= bm + 1; Q;
                WRITE;
                P(m); bm:= bm - 1; aw:= aw - 1; bm:= bm + 1; Q
         od
```

with Q short for

```
Q:    if bm > 0 → V(m)                                              (6')
       [] aw = 0 and br > 0 → V(r)
       [] aw = 0 and ar = 0 and bw > 0 → V(w)
      fi
```

Note. The transformation of the introduction of the counters and of the strengthening of the guards in Q by the requirement that the corresponding counters be positive, excludes the danger of deadlock. If the original requirement --in our case: the invariance of P1-- entailed intrinsically the danger of deadlock, this danger is now made manifest by the danger of abortion in Q . The systematic procedure for dealing with that situation falls outside the scope of this tutorial. (End of Note.)

Our original system was free from deadlock, hence we must --see above note-- be able to prove the absence of the danger of abortion in Q . The precondition of Q implies everywhere in (6)

$$bm + br + bw = \text{number of readers} + \text{number of writers}$$
$$ar + br \leq \text{number of readers}$$
$$aw + bw \leq \text{number of writers}$$

Task. Verify the above three assertions. (End of Task.)

From the above we conclude

$$ar + aw \leq bm \quad \text{, and hence}$$
$$bm > 0 \ \underline{\text{or}}\ (ar = 0 \ \underline{\text{and}}\ aw = 0) \quad .$$

Assuming the number of processes to be larger than zero --otherwise the danger of abortion is absent anyhow!-- , we also have

$$bm > 0 \ \underline{\text{or}}\ br > 0 \ \underline{\text{or}}\ bw > 0$$

and from the last two relations it follows that at least one of the guards of Q as given in (6') is true.

* * *

The above form of Q as given in (6') is still very non-deterministic: we have the strategic freedom of strengthening the guards as long as we avoid the danger of abortion. As it stands, our solution does not exclude that readers or a writer are denied access to their critical section READ or WRITE without reason. Within Q we can give "priority" to V(r) or V(w) by strengthening the guard of V(m) to the conjunction of the complements of the other two guards:

Am: $(aw > 0$ <u>or</u> $br = 0)$ <u>and</u> $(aw > 0$ <u>or</u> $ar > 0$ <u>or</u> $bw = 0)$

Denoting by Ar and Aw respectively:

Ar: $aw = 0$ <u>and</u> $br > 0$

Aw: $aw = 0$ <u>and</u> $ar = 0$ <u>and</u> $bw > 0$

we can now substitute for Q in (6)

Q: <u>if</u> Am → V(m) [] Ar → V(r) [] Aw → V(w) <u>fi</u> . (7')

All (now superfluous) operations on bm can be omitted; using the post-conditions of the P-operations, the substitution instances of Q as given by (7') can be simplified. Thus we derive from (6) the multiprogram

initial state: ar = 0, aw = 0, br = 0, bw = 0, m = 1, r = 0, w = 0 (7)

```
reader:
do true → ncs;
    P(m); br:= br + 1; if aw > 0 → V(m) [] aw = 0 → V(r) fi;
    P(r); br, ar := br - 1, ar + 1; if br = 0 → V(m) [] br > 0 → V(r) fi;
    READ;
    P(m); ar:= ar - 1; if ar > 0 or bw = 0 → V(m)
                       [] ar = 0 and bw > 0 → V(w)
                       fi
od

writer:
do true → ncs;
    P(m); bw:= bw + 1; if aw > 0 or ar > 0 → V(m)
                       [] aw = 0 and ar = 0 → V(w)
                       fi;
    P(w); bw, aw := bw - 1, aw + 1; V(m);
    WRITE;
    P(m); aw:= aw - 1; if br = 0 and bw = 0 → V(m)
                       [] br > 0 → V(r)
                       [] bw > 0 → V(w)
                       fi
od
```

Remark. An inspection of the alternative constructs in (7) shows that only the very last one is non-deterministic: here we have, therefore, still a strategic freedom. Investigate the consequences of replacing the last guarded command $bw > 0 \rightarrow V(w)$ by $bw > 0$ and $br = 0 \rightarrow V(w)$. (End of Remark.)

* * *

In the above we have derived our final program as the end of a sequence of successive versions. We have done so for educational reasons, with the intent of introducing the different aspects of programs synchronized with a split binary semaphore one after the other. This is not meant as a suggestion that in the case of actual program design one should write down all those successive versions! The experienced programmer knows that "outside the critical sections" as delineated by the P - V pairs, we have an invariant of the form

$$(m = 0 \text{ or } Am) \text{ and } (r = 0 \text{ or } Ar) \text{ and } (w = 0 \text{ or } Aw)$$

and focusses his attention on the A's; having chosen them he derives the final code. I make this remark because so-called "program transformations" are sometimes suggested as a practical way of program derivation --not by me, for as a rule it leads to an undue amount of writing-- .

5th March 1979
Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow