Copyright Notice

Variations on a theme: an open letter to C.A.R.Hoare.

Dear Tony!

For a variety of reasons I have not yet reacted to your article on Monitors [1]. For one thing: it failed to convince me --something I felt bad about, because I knew that this might have been due to the circumstance that I had been too lazy to go in detail through your more sophisticated examples--. Secondly: I was not too pleased either with the alternatives I could offer myself --my difficulty in finding good identifiers for the operations I was considering was just a symptom of my own mixed feelings--. Eventually I got interested in what one can do without mutual exclusion, and I dropped the subject --not without remorse, for I had left a task undone: I had failed to make up my mind!--.

Recently the topic was brought back to my attention by a nice technical report by Coen Bron [2], and in a Tuesday afternoon discussion with Wim Feijen, Alain Martin, Martin Rem and Liesbeth Steffens I tried, as a result, to redesign my (formerly rejected) alternative, in the hope that this time I could do a more conclusive job. This letter records the quintessence of that discussion and the following considerations.

About the microscopic delays implied by mutual exclusion.

The whole purpose of a monitor is to grant mutually exclusive access to a bunch of common variables, and this implies two things.

First: the whole monitor concept is only adequate in circumstances such that a monitor will only be "active" during a negligible fraction of time. (And, on the next higher level of abstraction, we shall indeed ignore the CPU-time spent on "monitoring"!) Second: in any multiprocessor installation, attempted monitor calls while the monitor is active imply delays, but in view of the first remark I propose to attach no significance whatsoever to the order in which such " microscopic" delays have been caused. Such microscopic delays will last until the moment when otherwise the monitor would have become inactive, and one of the micriscopically delayed processes will be granted access to the monitor. Our only (logical!) requirement is the exclusion of the (in view of our first remark highly improbable) danger of individual starvation. (Round Robin, for instance, would do!) In the following the microscopic delays will not be mentioned anymore, logically it is as if "by magic" no process attempts to call a monitor while it is active. (Early in the discussion I had failed to make a clear distinction between microscopically delayed processes eager to call the monitor, and macroscopically delayed processes that, being woken up, were eager to continue an interrupted execution of a monitor procedure --in what follows the latter class will disappear--; this confusion was so disastrous that it did not last long!)

Note.At the lowest level I expect no objection to implementing the microscopic delays by means of the busy form of waiting. (End of Note.)

About the macroscopic delays introduced by a monitor.

The further purpose of a monitor is to introduce macroscopic delays when necessary, and, ideally, a monitor is formulated in such a fashion that it does not reflect the number of partners between which the cooperation is regulated. It should describe "my" behaviour versus "the others". (In the THE-system the

cooperation was coded in a context in which all partners were individually known and explicitly referred to; in retrospect I regard that now as one of the more significant shortcomings of that system.) In order to describe the rules of cooperation in a way which was independent of the number of partners involved, I envisaged to describe it in terms of a finite number of named queues of sleeping --i.e. macroscopically delayed-- processes, where the queues themselves could be of any length, and each sleeping process would occur at exactly one queue.

Right at the start, our decision that the elements on a queue should be linearly ordered, seemed more emphatic than yours. You write: "If more than one program is waiting on a condition, we postulate that the signal operation will activate the longest waiting program. This gives a simple, neutral queueing discipline, which ensures that every waiting program will eventually get its turn." But if individual starvation is the danger you would like to exorcize, Round Robin or allowance counts would have done as well.

I propose for the linear order of the elements in each queue a role that seems to me much more fundamental: "the (sleeping) others" are known to "me" by virtue of their place in one of the queues. If they were sets instead of linearly ordered queues, the different "(sleeping) others" would have no distinct identities.

* * *

(Continued after an interlude during which I just listened to Dvorak's Serenade --mainly for wind instruments-- in D moll, opus 44: a delightful piece of music!)

I saw --you know my weakness for railroad metaphors!-- the queues as one-directional railroad tracks of a shunting yard with each "(sleeping) other" in its own carriage --sleeper, if you so desire!-- somewhere on one of the tracks of the shunting yard. Waking up a process implies that it leaves the yard and, therefore, the track on which it is waiting. But why should leaving a track imply waking up? In this view it comes quite naturally to allow that sleeping processes can be shunted from one track to another without being woken up. Thanks to this metaphor I freed myself of one of the constraints you had introduced.

Now for some terminology, in order to avoid misunderstanding. A process is "in monitor state" from the beginning of the execution of the first statement of a monitor procedure it has called until the end of the execution of the dynamically last statement of that monitor procedure, when its concurrently executable code can continue to be obeyed. For a given monitor n processes may be in monitor state. Either the monitor is inactive: in that case all n processes are sleeping somewhere on the shunting yard and each process, when woken up --i.e. removed from the shunting yard-- will continue the execution of the monitor procedure it had called at the point, where it had gone to sleep. Or the monitor is active: in that case $n-1$ processes are sleeping somewhere on the shunting yard and one of them has the special status "me", viz. the process, whose monitor call is continued to be executed. (Associating your "conditions" with my "tracks" of the shunting yard, this represents a slight departure from your proposal, in which a process that is awake --i.e. does not occur on a queue-- can wake up another by signalling: you then have more than one process being awake, but only one, whose monitor procedure execution is continued. I preferred to identify "me" with the one and only active process and to have <u>all</u> others in monitor state explicitly somewhere on the shunting yard.)

What I was looking for was a nice set of operations in terms of which I could describe the shunting, the reallocation of "me" and the leaving of "me" of the monitor state. I did not like your term "condition" as it evoked in my mind the wrong associations: it does not reflect a linearly ordered set of sleeping processes. For lack of a better name I introduced the type "fifoq" --being an acronym for "first-in-first-out-queue"-- , but this was a very grave mistake, which led me astray for more than 24 hours! It implies too much about the long-range history whereas at each moment only the current value matters! It was a mental liberation when it dawned upon me that I could stay within the shunting yard metaphor and could just call them "trains". (For a while I used the term "tracks", but that was discarded on account of its associations with drums and disks. Eventually the transitions from track to train turned out to be a blessing: whereas the "track" suggest a "place holder" or a "location", the "train" suggests a value, viz. a linearly ordered sequence of sleeping processes. It opens the way to "train expressions", which describe how new trains are composed out of the cars already on the shunting yard. It is, by the way, frightening to observe the devious and sometimes obnoxious influence of the terms I tentatively introduce! The wrong choice can drag in the wrong associations or deny you the expressive power needed to describe what you would like to think about, but then are unable to do. How does one avoid falling unaware into the trap of the inadequate metaphor? I know so many earlier instances of my falling into that trap and I honestly try to be aware of the danger: yet I did it again!)

My next problem was with "wait" and "signal"; I tried "sleep" and "wake", but quickly ran out of names for more intricate shunting operations, possibly to be combined with a redefinition of "me": I found myself forced to describe the operation in which "me" should go to sleep "somewhere", and another sleeper should take over the role of "me" instead. I even considered horrible neologisms like "slake", in order to express the combination of putting one process to sleep and waking up another. As you can imagine, I quickly ran out of descriptive names.

The way out seemed the introduction of "train expressions" and an assignment statement. The train expression would describe the new train as a concatenation of (cars of) existing trains: its "evaluation" would have as implicit side-effect taking away the cars used in the new train value from the train operands: shunting does not change the number of cars on the shunting yard! I tried to describe just shunting as an assignment to a train variable, just changing monitor activity from one process to another by an assignment to "me" and the combination of the two by a sort of concurrent assignment with at the left-hand side "me" and a train variable.

It was understood that "me" could occur as component of a train expression. And it was the idea that, by definition, the shunting yard should contain the sleeping processes, that caused the need for the concurrent assignment. Composing a new train, containing "me" could not be the first assignment statement, for then the active process would sleep before it had assigned a new value to "me", I could not invert the order either, because then I would have two "me's". Hence the idea of the concurrent assignment, which solves such problems.

It looked promising and I started to write a manuscript, but after a couple of hours at least ten pages were thrown into the waste paper basket, because, although it worked after a fashion, the code needed for the monitors became more and more tortuous as my examples became more ambitious. It was really appalling! I was coding in a conceptually nice and clean interface, but in spite of its

conceptual simplicity apparently hopelessly inadequate. It was one of those rare beautiful days in which one can work in the garden, but in spite of the shining sun I was close to desparate. There was only one thing I could do: put all papers away, pour myself a glass of beer, look into the blue sky and figure out where I had got stuck.

One glass of beer --even parts of it!-- sufficed. Although "I" have to describe "my" behaviour versus "the others", "I" am part of the whole community, and it is extremely awkward if I cannot treat "me" on the same footing as "the others". While during inactivity of the monitor, all "sleepers" occur on the shunting yard, it is rash to identify --what I had done!-- the contents of the shunting yard with "the set of sleepers": during monitor activity, "me" should be allowed to occur (obviously at most once!) on the shunting yard as well, just as one of "the others"! This has a few drastic consequences. For reasons of safety, one should insist that all semicolons of a monitor procedure fall into one of two categories: those semicolons where "me" is somewhere on the shunting yard --and placing "me" on the shunting yard is not allowed and redefining "me" implies that the old "me" remains in monitor state and goes to sleep-- and those semicolons, where "me" is not on the shunting yard --where placing "me" on the shunting yard is allowed and re-definition of "me" implies that the old "me" leaves the monitor state--. To allow during monitor activity "me" to appear --at most once!-- on the shunting yard solved all my problems. It is such an obvious generalization: during monitor inactivity, "me" does not exists and, therefore, cannot occur on the shunting yard. Yet it took me hours of following of false ideas to discover it! I shall describe my new solutions at another occasion: tomorrow is Sunday, so I am not in a hurry, but in the meantime it is past two o'clock, and I had better go to sleep. I thank you --although you must be unaware of it!-- for your patience and your inspiring "presence". My problem is, that I really like letter writing.....

* * *

(Sunday afternoon, 6th July 1975.)

Let a train expression enumerate in order from left to right the trains the cars of which are concatenated to form the new train value in order from "head" to "tail". With

tr0, tr1, tr2: train

examples of train expressions are

(tr0, tr1) : this train consists of the cars of tr0 , followed by the cars of tr1 . As a result of this train formation, the trains tr0 and tr1 have become empty, which value is indicated by "nil".

(tr2, me) forms a train one longer than tr2 , by appending "me" at the rear end.

(me, tr2) forms a train one longer than tr2 , by putting "me" in front of the train tr2.

Shunting operations I shall indicate by means of assignment statements

< train variable > := < train expression > e.g.

tr0 := (tr1, tr0) tr2 := (tr2, me) tr0 := (tr0, tr1, tr2) etc.

After evaluation of the train expression, the train assigned to must

be empty, otherwise its cars would "disappear". One way of imposing this is to require that in the train assignment the train assigned to occurs somewhere in the train expression. I shall not do so, and allow

tr0 := (tr1, tr2)

--if you so desire as abbreviation of "tr0 := (tr0, tr1, tr2)"-- in those circumstances that I can assert the initial emptyness of tr0 .

Potential change of "me" will also be indicated by an assignment statemer

me:= head(tr0) me:= nil .

When the value "nil" is assigned to "me", the monitor becomes inactive until the next call of a monitor procedure, which implicitly assigns to "me" the identity of the calling process. The evaluation of the function "head(tr0)" gives for initially non-empty tr0 as value the first element of tr0 , which is taken away from tr0 . (Note that also this is a glorious side-effect: all problems can be solved by postulating that the components of a train expression are evaluated in order from left to right.) If initially tr0 is empty, it remains so, and the value of head(tr0) equals "nil".

These two types of assignment statement enable us to separate completely shunting on the one hand and process switching on the other. Note that an assignment to "me"

1) must be a dynamically last statement of a monitor procedure when "me" does not occur on the shunting yard; the process that was "me" leaves monitor state and can continue with its concurrently executable code

2) should not be a dynamically last statement of a monitor procedure when "me" does occur on the shunting yard; the process that was "me" remains in monitor state, but remains asleep until its identity is reassigned to "me", whereafter the execution of the interrupted monitor procedure is resumed at the next statement.

Now for some examples. Let me first code your single resource monitor, which macroscopically on fifo basis grants the single resource ([1], page 550)

```
single resource: monitor
begin busy: boolean;
      nonbusy: train;
      proc acquire:
           if busy → nonbusy:= (nonbusy, me); me:= nil
            [] non busy → skip
           fi;
           busy:= true; me:= nil
      corp acquire;
      proc release:
           if busy → busy:= false; me:= head(nonbusy) fi
      corp release;
      busy:= false
end
```

(As you have seen, a call of "release" while non busy leads to abortion.) The above is a straight transliteration of your text and does not reflect too clearly, that acquire will only assign the value true to busy , when initially it is false. I offer the following alternative solution for acquire:

```
proc acquire:
     nonbusy:= (nonbusy, me); me:= head(nonbusy);
     do busy → nonbusy:= (me, nonbusy); me:= nil od;
     busy:= true; me:= nil
corp acquire
```

When you see this for the first time, it may strike you as a coding trick: depending on whether nonbusy is empty to start with "me:= head(nonbusy)" will leave "me" unaffected or not. The test on "business" is only performed by the one which was at the head of the queue, and when it finds busy true, it places itself back at the head side.

But it allows a nice generalization. Suppose that we have to synchronize the unbounded buffer, where (with $p > 0$ and $c > 0$)

prod(p): n:= n + p and cons(c): n:= n - c

have to be synchronized in such a fashion that $n \geq 0$ remains invariant. Here we go: (consumers being served on fifo basis)

```
ubb:monitor
begin n: integer;
      con: train;
      proc prod(p: integer):
           n:= n + p; me:= head(con)
      corp prod;
      proc cons(c: integer):
           con:= (con, me); me:= head(con);
           do n < c → con:= (me, con); me:= nil od;
           n:= n - c; me:= head(con)
      corp cons;
      n:= 0
end ubb.
```

Finally, the same problem, but instead of serving the consumers on fifo basis, they may try on fifo basis.

```
ubb: monitor
begin n: integer;
      con, temp: train;
      proc prod(p: integer):
           n:= n + p; temp:= (con); me:= head(temp)
      corp prod;
      proc cons(c: integer):
           if n ≥ c → n:= n - c; me:= nil
            [] n < c → con:= (con, me); me:= nil;
                       do n < c → con:= (con, me); me:= head(temp) od;
                       n:= n - c; me:= head(temp)
           fi
      corp cons;
      n:= 0
end ubb
```

This strategy has, of course the danger of individual starvation: another strategy with the same danger is to give priority to the requesting consumer with maximum value of c . The coding of that one is quite fun and I leave it as an exercise to you.

* * *

If I wanted to make a really strong case for my constructs, I should, of course, continue this letter with the coding of all your examples, but I am not going to do that now: after all, it is Sunday afternoon! For the time being I have the feeling of having done my share, and I am looking forward to your comments in particular.

You will have noticed that, for instance, in "release" I need at the end an additional "me:= nil". We could allow its omission and make the additional rule that it will be supplied by default. If you are going to suggest that as an improvement of my proposal, I promise that I shall get very cross with you (or, for that matter, with anyone else who suggests that "improvement")!

A shortcoming could be that we have only variables local to the monitor and locals of each call: if you look at "temp" it could be a local of a "monitor activity". Do we think that a serious shortcoming? It could be overcome by declaring "temp", "prod" and "cons" inside an special "inner block" of the monitor that is entered upon activation of the monitor and left at the moment the monitor becomes inactive. I think that I don't care about this refinement, but I may be overlooking a forceful argument in favour.

My dear Tony, it was as always a pleasure and a privilege to write to you. With greetings and best wishes,

yours ever

Edsger

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow

[1] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept" Comm.ACM, 17, 10 (Oct. 1974) 549 - 557
[2] Bron, C., "Description of Conditional Critical Regions in Terms of P- and V-Operations." Memorandum nr. 84, May 1975, Department of Applied Mathematics, Twente University of Technology, P.O.Box 217, Enschede, The Netherlands.

To Professor C.A.R.Hoare
Department of Computer Science
The Queen's University of Belfast
BELFAST BT7 1NN
Northern Ireland