EWD 123

Cooperating sequential processes

Table of Contents.

---

5. Cooperation via Status Variables.

In sections 4.1 and 4.3 we have illustrated the use of the general semaphore. It proved an adequate tool, be it as implementation of a rather trivial form of interaction. The rules for the consumer are very simple: if there is something in the buffer, consume it. They are of the same simplicity as the behaviour rules of the wage earner who spends all his money as soon as he has been paid and is broke until the next pay day.

In other words: when a group of cooperating sequential processes have to be constructed and the overall behaviour of these processes combined has to satisfy more elaborate requirements —community, formed by them, has, as a whole, to be well-behaved in some sense— we can only expect to be able to do so, if the individual processes themselves and the ways in which they can interact will get more refined. We can no longer expect a ready made solution as the general semaphore to do the job. In general, we need the flexibility as can be expressed in a program for a general purpose computer.

We now have the raw material, we can define the individual processes, they can communicate with each other via the common variables and finally we have the synchronizing primitives. How we can compose from it what we might want is, however, by no means obvious. We must now train ourselves to use the tools, we must develop a style of programming, a style of "parallel programming" I might say.

In advance I should like to stress two points.

We shall be faced with a great amount of freedom. Interaction may imply decisions bearing upon more than one process and it is not always obvious, which of the processes should do it. If we cannot find a guiding priciple (e.g. efficiency considerations), then we must have the courage to impose some rule in the name of clarity.

Secondly, if we are interested in systems that really work, we should be able to convince ourselves and anybody else who takes the trouble to doubt, of the correctness of our constructions. In uniprogramming one is already faced with the task of program verification —a task, the difficulty of which is often underestimated— but there one can hope to debug by testing of the actual program. In our case the system will often have to work under irreproducible circumstances and from field tests we can hardly expect any serious help. The duty of verification should concern us right from the start.

We shall attack a more complicated example in the hope that this will give us some of the experience which might oe used as guiding principle.

5.1. An Example of a Priority Rule.

In section 4.3 we have used the general semaphore to couple a producer and a consumer via a bounded buffer. The solution given there is extendable to more producers and/or more consumers; it is applicable when the "portion" is at the same time a convenient unit of information, i.e. when we can regard the different portions as all being of the same size.

In the present problem we consider producers that offer portions of different sizes; we assume the size of these portions to be expressed in portions units. The consumers, again, will process the successive portions from the buffer and will, therefore, have to be able to process portions, the size of which is not given a priori. A maximum portion size, however, will be known.

The size of the portions is given in information units, we assume also that the maximum capacity of the buffer is given in information units: the question whether the buffer will be able to accomodate the next portion will therefore depend on the size of the portion offered. The requirement, that "adding a portion to" and "taking a portion from the buffer" are still conceivable operations implies that the size of the buffer is not less than the maximum portion size.

We have a bounded buffer and therefore a producer may have to wait before it can offer a portion. With fixed size portions this could only occur when the buffer was full to the brim, now it can happen, because free space in the buffer, although present, is insufficient for the portion concerned.

Furthermore, when we have more than one producer and one of them is waiting, then the other ones may go an and reach the state that they wish to offer a portion. Such a portion from a next producer may also be too large or it may be smaller and it may fit in the available free space of the buffer.

Somewhat arbitrarily, we impose on our solution the requirement, that the producer wishing to offer the larger portion gets priority over the producer wishing to offer the smaller portion to the buffer. (When two or more producers are offering portions that happen to be of the same size, we just don't care.)

When a producer has to wait, because the buffer cannot accomodate its portion, no other producers can therefore add their portions until further notice: they cannot when the new portion is larger (for then it will also not fit), they may not when the new portion is smaller, for then they have a lower priority and must leave the buffer for the earlier request.

Suppose at a moment a completely filled buffer and three producers, waiting to offer portions of 1, 2 and 3 units respectively. When a consumer now consumes a five-unit portion, the priority rule implies that the producers with the 2-unit portion and the 3-unit portion respectively will get the opportunity to go on and not the one offering the 1-unit portion. It is *not* meant to imply, that then the 3-unit portion will actually be offered before the 2-unit portion!

We shall now try to introduce so-called "status variables" for the different components of the system, with the aid of which we can characterize the state of the system at any moment. Let us try.

For each producer we introduce a variable named "`desire`": this variable will denote the number of buffer units needed for the portion it could not add to the buffer. As this number is always positive, we can attach to "`desire` = 0" the meaning, that no request from this buffer is pending. Furthermore we shall introduce for each producer a private binary "`producer semaphore`".

For the buffer we introduce the binary semaphore "`bufman`", which takes care of the mutual exclusion of buffer manipulations in the widest sense (i.e. not only the adding to and taking from the buffer, but also inspection and modification of the status variables concerned.)

Next we need a mechanism to signal the presence of a next portion to the consumers. As soon as a next portion is in the buffer, it can be consumed and as we do not care, which of the consumers takes it, we can hope, that a general semaphore "`number of queuing portions`" will do the job. (Note, that it counts portions queuing in the buffer and not number of filled information units in the buffer.)

Freecoming buffer space must be signalled back to the producers, but the possible consequences of free coming buffer space are more intricate and we cannot expect that a general semaphore will be adequate. Tentatively we introduce an integer status variable "`number of free buffer units`". Note, that this variable counts units and not portions.

Remark. The value of "`number of free buffer units`" will at most be equal to the size of the buffer diminished by the total size of the portions counted in "`number of queuing portions`", but it may be less! I refer to the program given in section 4.3; there the sum

```
number of queuing portions + number of empty positions
```

is initially (and usually) = *N*, but it may be = *N* -1, because the P-operation on one of the semaphores always precedes the V-operation on the other. (Verify, that in the program of section 4.3 the sum can even be = *N* - 2 and that this value can even be lower, when we have more producers and/or consumers.) Here we may expect the same phenomenon: the semaphore "`number of queuing portions`" will count the portions actually and completely filled and still unnoticed by the consumers, "`number of free buffer units`" will count the completely free, unallocated units in the buffer. But the units which have been reserved for filling, which have been granted to a (waiting) producer, without already being filled, will not be counted in either of them,

Finally we introduce the integer "`buffer blocking`", the value of which equals the number of quantities "`desire`"? that are positive. Obviously, this variable is superfluous; it has been introduced as a recognition of one of our earlier remarks, that as soon as one of the desires is positive, no further additions to the buffer can be made, until further notice. At the same time this variable may act as a warning to the consumers, that such a "further notice" is wanted.

We now propose the following program, written for *N* producers and *M* consumers. ("`N`", "`M`", "`Buffer size`" and all that concerns the buffer is assumed to be declared in the surroundings of this program.)

```
begin integer array desire, producer semaphore [1 : N];
      integer number of queuing portions, number of free buffer units,
              buffer blocking, bufman, loop;
      for loop:= 1 step 1 until N do
           begin desire[loop]:= 0; producer semaphore[loop]:= 0 end;
      number of queuing portions:= 0;
      number of free buffer units:= Buffer size;
      buffer blocking:= 0; bufman:= 1;
      parbegin
      producer 1: begin.........................end;
                     .
                     .
                     .
                     .
                     .
      producer n: begin integer portion size;
              again n: produce next portion and set portion size;
                       P(bufman);
                       if buffer blocking = O and
                             number of free buffer units ≥ portion size
                                 then
```

```
                           number of free buffer units:=
                              number of free buffer units - portion size
                                         else
                           begin buffer blocking:= buffer blocking - 1;
                                 desire[n]:= portion size; V(bufman)
                                 P(producer semaphore[n]); P(bufman) end;
                           add portion to buffer; V(bufman);
                           V(number of queuing portions); goto again n
                   end;
                    .
                    .
                    .
       producer N: begin................end;
       consumer 1: begin................end;
                     .
                     .
                     .
       consumer m: begin integer portion size, n, max, nmax;
                again m: P(number of queuing portions); P(bufman);
                         take portion from buffer and set portion size;
                         number of free buffer units:=
                             number of free buffer units + portion size;
                test:    if buffer blocking > 0 then
                         begin max:= 0;
                               for n:= 1 step 1 until N do
                               begin if max < desire[n] then
                                 begin max:= desire[n]; nmax:= n end end;
                               if max ≤ number of free buffer units then
                               begin number of free buffer units:=
                                        number of free buffer units - max;
                                     desire[nmax]:= 0;
                                     buffer blocking:= buffer blocking - 1;
                                     V(producer semaphore[nmax]); goto test
                               end
                         end;
                         V(bufman); process portion taken; goto again m
                   end;
                    .
                    .
                    .
       consumer M: begin................end
       parend
end
```

In the outermost block the common variables are declared and initialized; I hope -and trust that this part of the program presents no difficulties to the reader that has followed me until here.

Let us first try to understand the behaviour of the producer. When it wishes to add a new portion to the buffer, there are essentially two cases: either it can do so directly, or not. It can add directly under the combined condition:

```
buffer blocking = 0 and
number of free buffer units ≥ portion size;
```

if so, it will decrease "number of free buffer units" and —dynamically speaking in the same critical section— it will add the portion to the buffer. The two following V-operations (the order of which is immaterial) close the critical section and signal the presence of the next portion to the combined consumers. If it cannot add directly, i.e. if (either)

```
buffer blocking > 0 or
number of free buffer units < portion size"
```

(or both), then the producer decides to wait, "to go to sleep", and delegates to the combined consumers the task to wake it up again in due time. The fact that it is waiting is coded by "`desire[n]` > O", "`buffer blocking`" is increased by 1 accordingly. After all clerical operations on the common variables have been carried out, the critical section is left (by `V(bufman)`") and the producer initiates a P-operation on its private semaphore. When it has completed this P-operation, it reenters the critical section, merges dynamically with the first case and adds the portion to the buffer. (See also the consumer in the second program of section 4.2, where we have already met the cutting open of a critical section.) Note that in the case of waiting, the producer has skipped the decrease of "`number of free buffer units`". Note also, that the producer initiates the P-operation on its private semaphore at a moment, that the latter may already be = 1, i.e. this P-operation, again, is only a potential delay.

Let us now inspect, whether the combined consumers fulfill the tasks delegated to them. The presence of a next portion is correctly signalled to them via the general semaphore "`number of queuing portions`" and as the P-operation on it occurs outside any critical section, there is no danger of consumers not initiating it. After this P-operation, the consumer enters its critical section, takes a portion and increases the number of free buffer units. If "`buffer blocking` = 0" holds, the following compound statement is skipped completely and the critical section is left immediately; this is correct, for "`buffer blocking` = 0" means that none of the quantities "`desire`" is positive, i.e. that none of the producers is waiting for the free space just created in the buffer. If, however, it finds "`buffer blocking` > 0", it knows that at least one of the producers has gone to sleep and it will inspect, whether one or more producers have to be woken up. It looks for the maximum value of "`desire`". If this is not too large, it decides, that the corresponding producer has to go on. This decision has three effects:

the "number of free buffer units" is decreased by the number of units desired. Thus we guarantee that the same free space in the buffer cannot be granted to mare than one producer. Furthermore this decrease is in accordance with the producer behaviour.

"desire" of the producer in question is set to zero; this is correct, for its request has now been granted; buffer blocking is decreased by 1 accordingly.

a V-operation on the producer semaphore concerned wakes the sleeping producer.

After that, control of the consumer returns to "`test`" to inspect, whether more sleeping producers should be woken up. The inspection process can end in one of two ways: either there are no sleeping producers anymore ("`buffer blocking` = 0) or there are still sleeping processes, but the free space is insufficient to accommodate the maximum desire. The final value of "`buffer blocking`" is correct in both cases. After the waking up of the producers is done, the critical section is left.

5.2. An Example of Conversations.

In this section we shall discuss a more complicated example, in which one of the cooperating processes is not a machine but a human being, the "operator".

The operator is connected with the processes via a so-called "semi-duplex channel" (say "telex connection"). It is called a duplex channel because it conveys information in either direction: the operator can use a keyboard to type in a message for the processes, the processes can use the teleprinter to type out a message for the operator. It is called a semi-duplex channel, because it can only transmit information in one direction at a time.

Let us now consider the requirements to the total construction. (I admit, that they are somewhat simplified. I hope, that they are sufficiently complicated to pose to us a real problem, yet sufficiently simple as not to drown the basic pattern of our solution in a host of inessential details. The trees should not prevent us from seeing the forest!)

We have *N* identical processes (numbered from 1 through *N*) and essentially they can ask a single question, called "Q1", meaning "How shall I go on?", to which the operator may give one of two possible answers, called "A1" and "A2". We assume, that the operator must know, which of the processes is asking the question —as his answer might depend on this knowledge— and we therefore specify, that the *i*-th process identifies itself when posing the question; we indicate this by saying that it transmits the question "Q1(*i*)". In a sense this is a consequence of the fact, that all *N* processes use the same communication channel.

A next consequence of this channel sharing between the different processes is that no two processes can ask their question simultaneously: behind the scenes some form of mutual exclusion must see to this. If only Q1-questions are mutually exclusive, the operator may meet the following situation: a question —say "Q1(3)— is posed, but before he has decided how to answer it, a next question —say"Q1(7)"— is put to him. Then the single answer "A1" is no longer sufficient, because now it is no longer clear, whether this answer is intended for "process 7" or for "process 3". This could be overcome by adding to the answers the identification of the process concerned, say, "A(*i*)" and "A2(*i*)" with the appropriate value of *i*.

But this is only one way of doing it: an alternative solution is to make the question, followed by its answer, together a critical occurence: it relieves the operator from the task to identify the process and we therefore select the latter arrangement. So we stick to the answers "A1" and "A2". We have two kinds of conversations "Q1(*i*), A1" and "Q1(*i*), A2" and the next conversation can only be initiated when the previous one has been completed.

We shall now complicate the requirements threefold.

Firstly, the individual processes may wish to use the communication channel for single-shot messages —"M(*i*)" say— which do not require any answer from the operator.

Secondly, we wish to give the operator the possibility to postpone an answer. Of course, he can do so by just not answering, but this would have the undesirable effect,that the communication channel remains blocked for the other *N*-1 processes. We introduce a next answer "A3", meaning: "The channel becomes free again, but the conversation with the process concerned remains unfinished." Obviously, the operator must have the opportunity to reopen the conversation again. He can do so via "~A(*i*)"or "A5(*i*)", where "*i*" runs from 1 through *N* and identifies the process concerned, where "A4" indicates that the process should continue in the same way as after "A1", while "A5" prescribes the reaction as to "A2". Possible forms of conversation are now:
a) "Q1(*i*), A1"
b) "Q1(*i*), A2"
c) "Q1(*i*), AT3" - - - "A4(*i*)"
d) "Q1(*i*), AT3" - - - "A5(*i*)"
As far as process *i* is concerned a) is equivalent with c) and b) is equivalent with d).

The second requirement has a profound influence: without it —i.e. only
"A1" and "A2" permissible answers— the process of incoming message interpretation can always be subordinate to one of the *N* processes, viz. the one, that has put the question: this can wait for an answer and can act accordingly. We do not know beforehand, however, when the message "A4(*i*)" or "A5(*i*)" comes and we cannot delegate the interpretation of it to the *i*-th process, because the discovery that this incoming message is concerned with the *i*-th process is part of the message interpretation itself!

Thirdly, A4-and A5-messages must have priority over Q1- and M-messages, i.e. while the communication channel is occupied (in a Q1-or M-message), processes might reach the state, that they want to use the channel, but also the operator might come to this conclusion. As soon as the channel becomes available, we wish, that the operator can use it and that it won't be snatched away by one of the processes. This implies that the operator has a means to express this desire —a rudimentary form of input— even if the channel itself is engaged in output.

We assume that

a) the operator can give externally a

`V(incoming message)`,

which he can use to announce a message (A1, A2, A3, A4, or A5)

b) can detect by the machines reaction, whether the message is accepted or ignored.

Remark. The situation is not unlike the school teacher shouting "Now children, listen!". If this is regarded as a normal message, it is nonsensical: either the children are listening and it is therefore superfluous, or they are not listening, and therefore they do not hear it. It is, in fact a kind of "meta-message", which only tells, that a normal message is coming and which should also penetrate if the children are not listening (talking, for instance).

This priority rule may make the communication channel reserved for an announced A4 - or A5 message. By the time that the operator gets the opportunity to give it, the situation or his mood may have changed, and therefore we extend the list of answers with "A6" —the dummy opening— which enables the operator to withhold, upon further consideration, the A4 or A5.

A final feature of the message interpreter is the applicability test. The operator is a human being and we may be sure that he will make mistakes. The states of the message interpreter are such that at any moment, not all incoming massages are applicable; when a message has been rejected as non-applicable, the interpreter should return to such a state that the operator can now give the correct version.

Our attack will be along the following lines:
1) Besides the *N* processes we introduce another process, called "message interpreter"; this is done because it is difficult to make the interpretation of the messages "A4", "A5" and "A6" subordinate to one of the *N* processes.
2) Interpretation of a message always implies, besides the message itself, a state of the interpreter.(In the trivial case this is a constant state, viz. the willingness to understand the message.) We have seen that not all incoming messages are always acceptable, so our message interpreter will be in different states. We shall code them via the (common) state variable "`comvar`". The private semaphore, which can delay the action of the message interpreter, is the semaphore "`incoming message`", already mentioned.

3) For the *N* processes we shall introduce an array "*procsem*" of private semaphores and an array "`procvar`" of state variables, through which the the different processes can communicate with each other, with the message interpreter and vice versa.
4) Finally we introduce a single binary semaphore "`mutex`" which caters for the mutual exclusion during inspection and/or modification of the common variables.
5) We shall use the binary semaphore "`mutex`" only for the purpose just described and never, say, will "`mutex` = 0" be used to code, that the channel is occupied. Such a convention would be a dead alley in the sense that the technique used would fall into pieces as soon as the *N* processes would have two channels (and two operators) at their disposal. We aim to make the critical sections, governed by "`mutex`" rather short and we won't shed a tear if some critical section is shorter than necessary.

Well, the above five points, articles of faith, I might say, are of some help and I hope that in view of our previous experiences they seem a set of reasonable principles. I do one part of my job if I present a solution along the lines just given and show that it is correct. I would do a better job if I could show as well, how such a solution is found. Admittedly by trial and error, but even so, we could try to make the then prevailing guiding principle (in mathematics usually called "The feeling of the genius") somewhat more explicit. For we are still faced with problems:
a) what structure should we give to the *N* + 1 processes?
b) what states should we introduce (i.e. how many possible values should the state variables have and what should be their meanings)?

The problem (both in constructing and in presenting the solution) is, that the two points just mentioned are interdependent. For the values of the state variables have only an unambiguous, describable meaning, when "`mutex` = 1" holds, i.e. none of the processes is inside a critical section, in which they are subject to change. In other words: the conditions under which the meaning of the state variable values should be applicable is only known, when the programs are finished, but we can only make the programs if we know what inspections of and operations on the state variables are to be performed. In my experience one starts with a rough picture of both programs and state variables, one then starts to enumerate the different states and then tries to build the programs. Then two different things may happen: either one finds that one has introduced too many states or one finds that —having overlooked a need for cutting a critical section into parts— one has not introduced enough of them. One modifies the states and then the program and with luck and care the design process converges. Usually I found myself content with a working solution and I did not bother to minimize the number of states introduced.

In my experience it is easier to conceive first the states (being statically interpretable) and then the programs. In conceiving the states we have to bear three points in mind.

a) State variables should have a meaning when `mutex` is = 0; on the other hand a process must leave the critical section before it starts to wait for a private semaphore. We must be very keen on all those points where a process may have to wait for something more complicated than permission to complete "`P(mutex)`",

b) The combined state variables specify the total state of the system. Nevertheless it helps a great deal if we can regard some state variable as "belonging to that and that process". If some aspect of the total state increases linearly with *N*, it is easier to conceive that part as equally divided among the *N* processes.

c) If a process decides to wait on account of a certain (partial) state, each process that makes the system leave this partial state should inspect whether on account of this change, some waiting process should go on. (This is only a generalization of the principle, already illustrated in The Sleeping Barber.)

The first two points are mainly helpful in the conception of the different states, the last one is an aid to make the programs correct.

Let us now try to find a set of appropriate states. We starts with the element "`procvar[i]`", describing the state of process `i`.

```
procvar[i] = 0
```

This we call "the homing position". It will indicate that none of the following situations applies, that process `i` does not require any special service from either the message interpreter or one of the other processes.

```
procvar[i] = 1
```

"On account of non-availability of the communication channel, process `i` has decided to wait on its private semaphore." This decision can be taken independently in each process, it is therefore reasanable to represent it in the state of the process. Up till now there is no obvious reason to distinguish between waiting upon availability for a M-message and for a Q1-question, so let us try to do it without this distinction.

```
procvar[i] = 2
```

"Question "`Q1(i)`" has been answered by "`A3`", viz. with respect to process `i` the operator has postponed his final decision." The fact of the postponement must be represented because it can hold for an undefinitely long period of time (observation a); it should be regarded as a state variable of the process in question as it can hold in *N*-fold (observation b). Simultaneously, "`procvar[i] = 2`" will act as applicability criterion for the operator messages "`A4[i]`"and "`A5[i]`".

```
procvar[i] = 3
```

""`Q1[i]`" has been answered by "`A1`" or by "`A3`"- - -"`A4[i]`"."

```
procvar[i] =
```

""`Q1[i]`" has been answered by "`A2`" or by "`A3`"- - -"`A5[i]`"."

First of all we remark, that it is of no concern to the individual process, whether the operator has postponed his final answer or not. The reader may wonder, however, that the answer given is coded in "`procvar`", while only one answer is given at a time. The reason is that we do not know how long it will take the individual process to react to this answer: before it has done so, a next process may have received its final answer to the `Q1`- question.

Let us now try to list the possible states of the communication organisation. We introduce a single variable, called "`comvar`" to distinguish between these states. We have to bear in mind three different aspects
1) availability of the communication possibility for M-messages, Q1-questions and the spontaneous message of the operator.
2) acceptability —more general: interpretability— of the incoming messages.
3) operator priority for incoming messages.
In order not to complicate matters immediately too much, we shall start by ignoring the third point. Without operator priority we can see the following states.

```
comvar = 0
```

"The communication facility is idle", i.e. equally available for both processes and operator. For the processes "`comvar = 0`" means that the communication facility is available, for the message interpreter it means that an incoming message need not be ignored, but must be of type A4, A5 or A6.

```
comvar = 1
```

"The communication facility is used for a M-message or a Q1-question".
In this period of time the value of "`comvar`" must be ≠ 0, because the communication facility is not available for the processes; for the message interpreter it means, that incoming messages have to be ignored.

```
comvar = 2
```

"The communication facility is reserved for an A1-,A2- or A3-answer." When the M-message has been finished, the communication facility becomes available again, after a Q1-question, however, it must remain reserved. During this period, characterized by "`comvar = 2`", the message interpreter must know to which process the operator answer applies. At the end of the answer, the communication facility becomes again available.

Let us now take the third requirement into consideration. This will lead to a duplication of (certain) states. When "`comvar = 0`" holds, an incoming message is accepted, when "`comvar = 1`", an incoming message must be ignored. This occurence must be noted down, because at the end of this occupation of the communication facility, the operator must get his priority. We can introduce a new state:

```
comvar = 3
```

"As "`comvar = 1`" with operator priority requested."

When the transition to "`comvar = 3`" occurred during a M-message, the operator could get his opportunity immediately at the end of it; if, however, the transition to "`comvar = 3`" took place during a Q1-question, the priority can only be given to the operator after the answer to the Ql-question. Therefore, also state 2 is duplicated:

```
comvar = 4
```

"As "`comvar = 2`", with operator priority requested."

Finally we have the state:

```
comvar = 5
```

"The communication facility is reserved for, or used upon instigation of the operator." For the processes this means non-availability, for the message interpreter the acceptability of the incoming messages of type A4, A5 and A6. Usually, these messages will be announced to the message interpreter while "`comvar`" is = 0. If we do not wish that the entire collection and interpretation of these messages is done within the same critical section, the message interpreter can break it open.It is then necessary, that "`comvar`" is ≠ 0. We may try to use the same value 5 for this purpose: for the processes it just means non-availability, while the control of the message interpreter knows very well, whether it is waiting for a spontaneous operator message (i.e. "reserved for..") or interpreting such a message (i.e. "used upon instigation of. .").

Before starting to try to make the program, we must bear in mind point c: remembering that availability of the communication facility is the great (and only) bottleneck, we must see to it, that every process that ends a communication facility occupation decides upon its future usage. This is in the processes at the end of the M-message (and not so much at the end of the Q1-question, for then the communication facility remains reserved for the answer) and in the message interpreter at the end of each massage interpretation.

The proof of the pudding is the eating, let us try, whether we can make the program. (In the program, the sequence of characters starting with "`comment`" and up to and including the first semicolon are inserted for explanatory purposes only. In ALGOL 60, such a comment is only admitted only immediately after "`begin`" but I do not promise, to respect this (superfluous) restriction. The following program should be interpreted to be embedded in a universe in which the operator, the communication facility and the semaphore "incoming message" —initially = 0— are defined.

```
begin integer mutex, comvar, asknum, loop;
      comment The integer "asknum" is a state variable of the message
      interpreter, primarily during interpretation of the answers A1, A2
      and A3. It is a common variable, as its value is set by the asking
      process.;
      integer array procvar, procsem [1 : N]
      for loop:= 1 step 1 until N do
      begin procvar[loop]:= 0; procsem[loop]:= 0 end;
      comvar:= 0; mutex:= 1;
      parbegin
process 1: begin...................end;
               .
               .
               .
process n: begin integer i; comment The integer "i" is a local variable,
                 very much like "loop".;
                   .
                   .
                   .
      M message:P(mutex);
                if comvar = 0 then
                begin comment When the communication facility is available,
                      it is taken.;
                      comvar:= 1; V(mutex)end
                              else
                begin comment Otherwise the process books itself as sleeping
                      and goes to sleep.;
                      procvar[n]:= 1; V(mutex); P(procsem[n])
                      comment At the completion of this P-operation,
                      "procsem[n]" will again be = 0, but comvar -still
                      untouched by this process- will be =1 or =3.; end;
                send M message;
                comment Now the process has to analyse, whether the operator
                (first!) or one of the other processes should get the commu-
                nication facility or not.; P(mutex);
                if comvar = 3 then comvar:= 5
                              else
                begin comment Otherwise "comvar = 1" will hold and process n
                      has to look whether one of the other processes is waiting.
                      Note that "procvar[n] = 0" holds.;
                      for i:= 1 step 1 until N do
```

```
                          begin if procvar[i]= 1 then
                                begin procvar[i]:= 0; V(procsem[i]); goto ready
                                end
                          end;
                          comvar:= 0
                    end
             ready: V(mutex);
                    .
                    .
                    .
       Q1 Question: P(mutex);
                    if comvar = 0 then
                    begin comvar:= 1; V(mutex) end
                                  else
                    begin procvar[n]:= 1; V(mutex); P(procsem[n]) end;
                    comment This entry is identical to that of the M message.
                    Note that we are out of the critical section, nevertheless
                    this process will set "asknum". It can do so safely, for no
                    other process, nor the message interpreter, will access
                    "asknum" as long as "comvar = 1 " holds.;
                    asknum:= n; send question Q1(n);
                    P(mutex);
                    comment "comvar" will be = 1 or = 3.;
                    if comvar = 1 then comvar:= 2 else comvar:= 4;
                    V(mutex); P(procsem[n]);
                    comment After completion of this P-operation, procvar[n]
                    will be = 3 or = 4. This process can now inspect and reset
                    its procvar, although we are outside a critical section.;
                    if procvar[n] = 3 then Reaction 1 else Reaction 2;
                    procvar[n]:= 0;
                    comment This last assignment is superfluous.;
                    .
                    .
                    .
               end;
               .
               .
               .
 process N: begin....................end;
 message interpreter:
               begin integer i;
               wait: P(incoming message);
                     P(mutex);
                     if comvar = 1 then comvar:= 3;
                     if comvar = 3 then
                     begin comment The message interpreter ignores the incoming
                     message, but in due time the operator will get the
                     opportunity.;
                     V(mutex); goto wait end;
                     if comvar = 2 or comvar = 4 then
                     begin comment Only A1, A2 and A3 are admissible. The inter-
                           pretation of the message need not be done inside a
                           critical section;
                           V(mutex);
                           interpretation of the message coming in;
                           if message = A1 then
                           begin procvar[asknum]:= 3; V(procsem[asknum]);
```

```
                              goto after correct answer end;
                        if message = A2 then
                        begin procvar[asknum]:= 4; V(procsem[asknum]);
                              goto after correct answer end;
                        if message = A3 then
                        begin procvar[asknum]:= 2; goto after correct answer end;
                        comment The operator has given an erroneous answer
                        and should repeat the message; goto wait;
after correct answer: P(mutex);
                        if comvar = 4 then
                        begin comment The operator should now get his opportunity;
                              comvar:= 5; V(mutex); goto wait end;
perhaps comvar to zero:for i:=1 step 1 until N do
                        begin if procvar[i] = 1 then
                              begin procvar[i]:= 0; comvar:= 1;
                                    V(procsem[i]); goto ready end
                        end;
                        comvar:= 0;
                 ready: V(mutex); goto wait
                  end;
                  comment The cases "comvar = 0" and "comvar = 5" remain.
                  Messages A4, A5 and A6 are admissible.;
                  if comvar = 0 then comvar:= 5;
                  comment See Remark 1 after the program.;
                  V(mutex);
                  interpretation of the message coming in
                  P(mutex);
                  if message = A4[process number] then
                  begin i:= "process number given in the message";
                        if procvar[i] = 2 then
                        begin procvar[i]:= 3; V(procsem[i]);
                              goto perhaps comvar to zero end;
                        comment Otherwise process not waiting for postponed
                        answer.; goto wrong message
                  end;
                  if message = A5[process number] then
                  begin i:= "process number given in the message";
                        if procvar[i] = 2 then
                        begin procvar[i]:= 4; V(procsem[i]);
                              goto perhaps comvar to zero end;
                        comment Otherwise process not waiting for postponad
                        answer.; goto wrong message
                  end;
                  if message = A6 then goto perhaps comvar to zero;
wrong message:    comment"comvar = 5" holds, giving priority to the operator
                  to repeat his message.;
                  V(mutex); goto wait
            end
      parend
end
```

Remark 1. If the operator, while "`comvar = 0`" or "`comvar = 5`" originally holds, gives an uninterpretable (or inappropriate) message, the communication facility will remain reserved for his next trial.

Remark 2. The final interpretation of the A4 and A5 messages is done within the critical section, as their admissibility depends on the state of the process concerned. If we have only one communication channel and one operator, this precaution is rather superfluous.

Remark 3. The for-loops in the program scan the processes in order, starting by process 1; by scanning them cyclically, starting at an arbitrary process (selected by means of a (pseudo) random number generator) we could have made the solution more symmetrical in the *N* processes.

Remark 4. In this sectinn we have first given a rather thorough exploration of the possible states and then the program. The reader might be interested to know that this is the true picture —"a life recording"— of the birth of this solution. When I started to write this section, the problem posed was for me as new as for the reader: the program given is my first version, constructed on account of the considerations and explorations given. I hope that this section may thus give a hint as how one may find such solutions.

5.2.1. Improvements of the Previous Program.

In section 5.2 we have given a first version of the program; this version has been included in the text, not because we are content with it, but because its inclusion completes the picture of the birth of a solution. Let us now try to embellish, in the name of greater conciseness, clarity and, may be, efficiency. Let us try to discover in what respects we have made a mess of it.

Let us compare the information flows from a process to the message interpreter and vice versa. In the one direction we have the common variable "`asknum`" to tell the message interpreter, which process is asking the question. The setting and the inspection of "`asknum`" can safely take place outside the critical sections, governed by "`mutex`", because at any moment at most one of the *N* + 1 processes will try to access "`asknum`". In the inverse information flow, where the message interpreter has to signal back to the *i*-th process the nature of the final operator answer, this answer is coded in "`procvar`". This is mixing things up, as is shown
a) by the "`procvar`"-inspection (whether `procvar` is = 3 or = 4), which is suddenly allowed to take place outside a critical section
b) by the superfluity of its being reset to zero.

The suggestion is to introduce a new

```
    integer array operanswer[1:N]  ,
```

the elements of which will be used in a similar fashion as "`asknum`". (An attractive consequence is that the number of possible values of "`procvar`" —the more fundamental quantity(see below)— does not increase any more, if the number of possible answers to the question Q1 is increased.)

I should like to investigate whether we can achieve a greater clarity by separating the common variables into two (or perhaps more?) distinct groups, in order to reflect an observable hierarchy in the way in which they are used. Let us try to order them in terms of "basicness".

The semaphore "`incoming message`" seems at first sight a fairly basic one, being defined by the surrounding universe. This is, however, an illusion: within the parallel compound we should have programmed (as *N* + 2nd process) the operator himself, and the semaphore "`incoming message`" is the private semaphore for the message interpreter just as "`procsem[i]`" is for the *i*-th process.)

Thus the most basic quantity is the semaphore "`mutex`" taking care of the mutual exclusion of the critical sections.

Then come the state variables "`comvar`" and "`procvar`" which are inspected and can be modified within the critical sections.

The quantities just mentioned share the property that their values must be set before entering the parallel compound. This property is also shared by the semaphores "`procsem`" (and "`incoming message`" see above), if we stick to the rules that parallel statements will access common semaphores via P- and V-operations exclusively.

(Without this restriction, request for the communication facility by process n could start with

```
P(mutex);

if comvar = 0 then
begin comvar:= 1; V(mutexj end
              else
begin procvar[n]:= 1; procsem[n]:= 0;
      V(mutexj; P(procsem[n]) end .
```

We reject this solution on the further observation, that the assignment "`procsem[n]`" is void, except for the first time that it is executed; the initialization of `procsem`'s outside the parallel compound seems therefore appropriate).

For the common variables, listed thus far I should like to reserve the name "status variables", to distinguish them from the remaining ones, "`asknum`" and "`operanswer`", which I should like to call "transmission variables".

I call the latter "transmission variables" because, whenever one of the processes assigns a value to such a variable, the information just stored is destinated for a well known "receiving party". They are used to transmit information between well-known parties.

Let us now turn our attention from the common variables towards the programs. Within the programs we have learnt to distinguish the so-called "critical sections", for which the semaphore "`mutex`" caters for the mutual exclusion. Besides these, we can distinguish regions, in which relevant actions occur, such as:

in the i-th process:
Region 1: sending an M-message
Region 2: sending a `Q1(i)`-question
Region 3: reacting to `operanswer[i]` (this region is somewhat open-ended)

and in the message interpreter:
Region 4: ignoring incoming messages
Region 5: expecting `A1`, `A2` or `A3`
Region 6: expecting `A4(i)`, `A5(i)` or `A6`

We come now to the following picture. In the programs we have critical sections, mutually excluded by the semaphore "`mutex`". The purpose of the critical sections is to resolve any ambiguity in the inspection and modification of the remaining state variables, inspection and modification performed for the purpose of more intricate "sequencing patterns" of the regions, sequencing patterns, that make the unambiguous use of the transmission variables possible. (If one process has to transmit information to another, it can now do so via a transmission variable, provided that the execution of the assigning region is always followed by that of the inspecting region before that of the next assigning region!)

In the embellished version of the program we shall stick to the rule that the true state variables will only be accessed in critical sections (if they are not semaphores) or via P- and V-operations (if they are semaphores), while the transmission variables will only be accessed in the regions. (In more complicated examples this rule might prove too rigid and duplication might be avoided by allowing transmission variables at least to be inspected within the critical section. In this example, however, we shall stick to it.)

The remaining program improvements are less fundamental.

Coding goes more smoothly if we represent the fact of requested operator priority not in additional values of "`comvar`" but in an additional two-valued state variable:

Boolean operator priority

(Quantities of type "Boolean" can take on the two values denoted by "true" and "false" respectively, viz. the same domain as "conditions" such as we have met in the if-clause.)

Furthermore we shall introduce two procedures; they are declared outside the compound and therefore at the disposal of the different constituents of the parallel compound.

We shall first give a short description of the new meanings of the values of the state variables "`procvar`" and "`comvar`":

`procvar[i] = 0`	homing position
`procvar[i] = 1`	waiting for availability of the communication facility for `M` or `Q1(i)`
`procvar[i] = 2`	waiting for the answer "`A4(i)`" or "`A5(i)`".
`comvar = 0`	homing position (communication facility free)
`comvar = 1`	communication facility for `M` or `Q1`
`comvar = 2`	communication facility for `A1`, `A2` or `A3`
`comvar = 3`	communication facility for `A4`, `A5` or `A6`.

We give the program without comments and shall do it in two stages: first the program outside the parallel compound and then the constituents of the parallel compound.

```
begin integer mutex, comvar, asknum, loop;
      Boolean operator priority;
      integer array procvar, procsem, operanswer[1:N];
      procedure M or Q entry(u); value u; integer u;
      begin P(mutex);
            if comvar = 0 then
            begin comvar:= 1; V(mutex) end
                          else
            begin procvar[u]:= 1; V(mutex); P(procsem[u]) end
      end;
      procedure select new comvar value;
      begin integer i;
```

```
            if operator priority then
            begin operator priority:= false; comvar:= 3 end
                                   else
            begin for i:=1 step 1 until N do
                  begin if procvar[i] = 1 then
                        begin procvar[i]:= 0; comvar:= 1;
                              V(procsem[i]); goto ready end
                  end;
                  comvar:= 0;
     ready: end
      end;

      for loop:= 1 step 1 until N do
         begin procvar[loop]:= 0; procsem[ioop]:= 0 end;
      comvar:= 0; mutex:= 1; operator priority:= false;
      parbegin
      process 1: begin....................end;
                   .
                   .
                   .
                   .
                   .
      process N: begin....................end;
      message interpreter:
                 begin....................end;
      parend
end
```

Here the *n*-th process will be of the form

```
process n: begin
             .
             .
             .
             .
             .
M message: M or Q entry(n);
Region 1: send M message;
          P(mutex); select new comvar value; V(mutex);
            .
            .
            .
            .
            .
Q1 question:M or Q entry(n);
Region 2:   asknum:= n;
            send Q1(n);
            P(mutex); comvar:= 2; V(mutex); P(procsem[n]);
Region 3:   if operanswer[n] = 1 then Reaction 1
                                 else Reaction 2;
              .
              .
              .
              .
              .
            end
```

When the message interpreter decides to enter Region 6 it copies, before doing so, the array "procvar": if an answer A4(i) should be acceptable, then "procvar[i] = 2" should already hold at the moment of announcement of the answer.

```
      message interpreter:
      begin integer i; integer array[1:N];
wait:       P(incoming message); P(mutex);
            if comvar = 1 then
Region 4:   begin operator priority:= true;
leave:            V(mutex); goto wait; end
            if comvar ≠ 2 then goto Region 6;
Region 5:   V(mutex); collect message;
            if message ≠ A1 and message ≠ A2 and message ≠ A3 then goto wait;
            i:= asknum;
            if message = A1 then operanswer[i]:= 1 else
            if message = A2 then operanswer[i]:= 2;
            P(mutex);
            if message = A3 then procvar[i]:= 2 else
signal to i:   V(procsem[i]);
preleave:   select new comvar value; goto leave;
Region 6:   if comvar = 0 then comvar:= 3;
            for i:= 1 step 1 until N do pvcopy[i]:= procvar[i];
            V(mutex); collect message;
            if message = A6 then begin P(mutex); goto preleave end;
            if message ≠ A4(prmcess number) and message ≠ A5(process number) then
                                                                   goto wait;
            i:= "process number given in the message";
            if pvcopy[i] ≠ 2 then goto wait;
            operanswer[i]:= if message = A4 then 1 else 2;
            P(mutex); procvar[i]:= 0; goto signal to i
      end
```

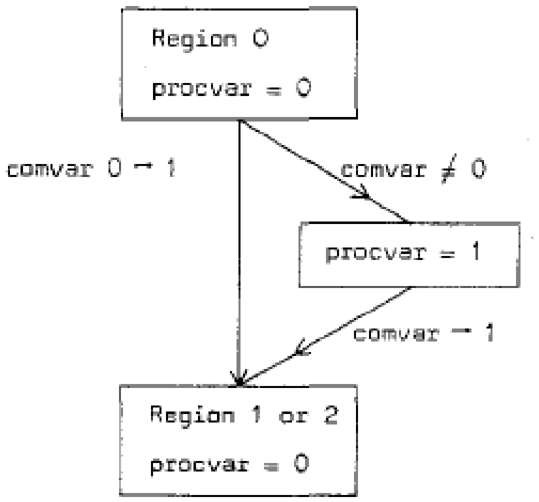
As an exercise we leave to the reader the version, where pending requests for Q1-questions have priority over those for M-messages. As a next extension we suggest a two console configuration with the additional restriction that an A4- or A5-message is only acceptable via the console over which the conver- sation has been initiated. (Otherwise we have to exclude simultaneous, contradicting messages "A4(i)" and "A5(i)" via the two different consoles. The solution without this restriction is left for the really fascinated reader.)

5.2.2. Provinq the Correctness.

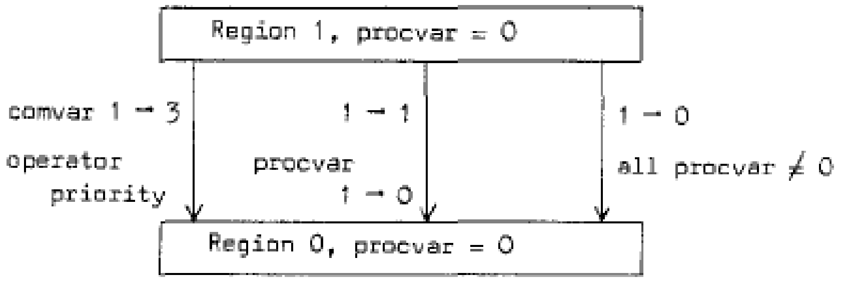
In this section title I have used the word "proving" in an informal way. I have not defined what formal conditions must be satisfied by a "legal proof" and I do not intend to do so. When I can find a way to discuss the program of section 5.2.1, by which I can convince myself—and hopefully anybody else that takes the trouble to doubt!—of the correctness of the overall performance of this aggregate of processes, I am content.

In the following "state picture" we make a diagram of a1 the states in which a process may find itself "for any length of time", i.e. outside sections, critical to mutex. In arrows we describe the transitions taking place within the critical sections; accompanying these arrows, we give the modifications of comvar or the conditions, under which the transition from one state to another is made.

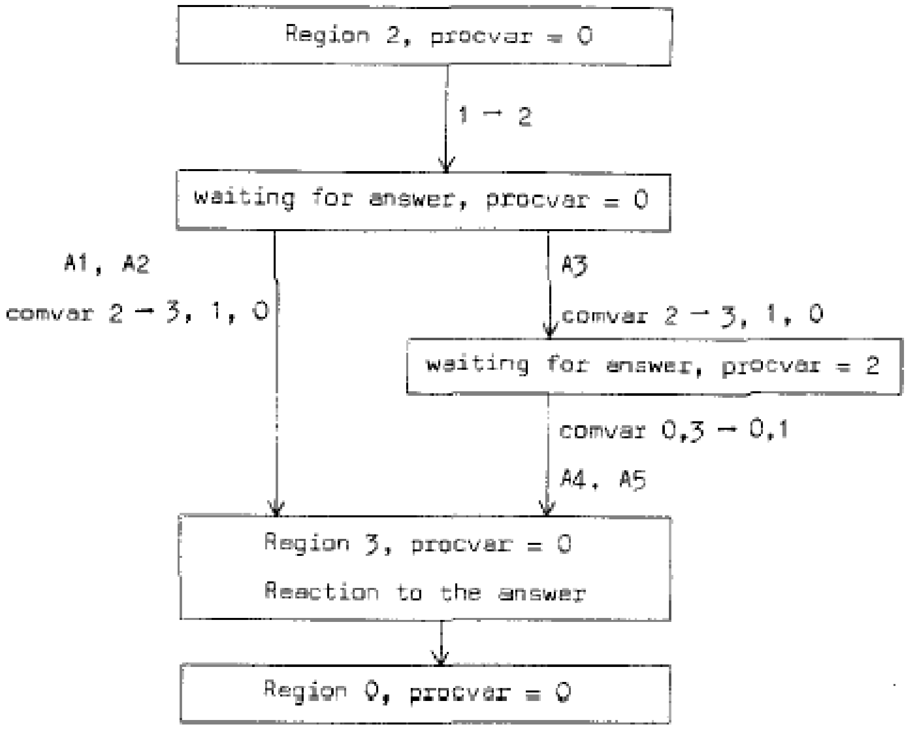
Calling the neutral region of a process before entry into a Region 1 or Region 2: "Region O", we can give the state picture



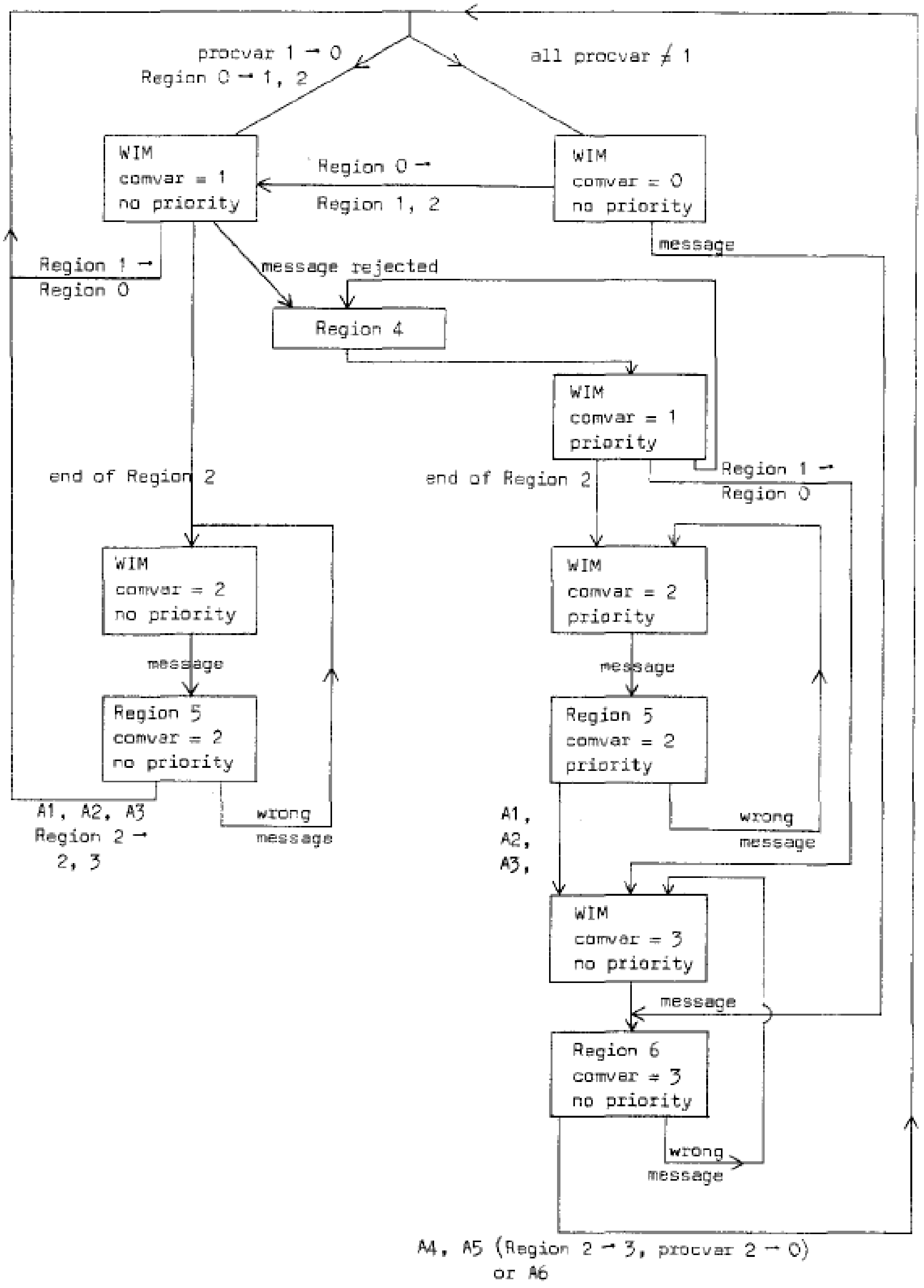
Leaving Region 1 can be pictured as:



Leaving Region 2, with the possibility of a delayed answer, can be pictured as:



We can try to do the same for the message interpreter. Here we indicate along the arrows the relevant occurrences, such as changes of a procvar and the kind of message. We use "WIM" as abreviation for "Waiting for Incoming Message".



These schemes, of course, teach us nothing new, but they may be a powerful aid in the program inspection.

We verify first, that "`comvar = 0`" represents indeed the homing position of the communication facility, i.e. available for either entrance into Region 1 or Region 2 (by one of the processes) or entrance into Region 6 (by the message interpreter, as result of an incoming message for which it is waiting).

If `comvar` = 0 and one of the processes wants to enter Region 1 or Region 2, or a message comes from the operator, Region 1, 2 or 6 is entered; furthermore this entrance is accompanied by either "`comvar:= 1`" or "`comvar:= 3`" and in this way care is taken of the mutual exclusion of the Regions 1, 2 and 6.

The mutual exclusion implies that processes may fail to enter Region 1 or 2 immediately, or that an incoming message must be rejected, coming at an inacceptable moment. In the first case, the process sets "`procvar:= 1`", in the second case (in Region 4) the message interpreter sets "`operator priority:=` <u>`true`</u>".

These assignments are only performed under the condition "`comvar ≠ 0`"; furthermore the assignment "`comvar:= 0`" —only occurring in the procedure "select new comvar value"— is only performed provided "<u>`non`</u> `operator priority` <u>`and`</u> `all procvar ≠ 1`". From these two observations and the initial values, we can conclude:

"`comvar = 0`" excludes "`operator priority`" as well as the occurrence of one
or more "`procvar = 1`".

As all ends of occupation of the communication facility (i.e. the end of Region 1, 5 and 6) call "`select new comvar value`" we have established
a) that entrance into the Region 1, 2 and 6 is only delayed when necessary
b) that such a delay is guaranteed to be resolved at the earliest opportunity.

The structure of the message interpreter shows clearly that
a) it can execute Region 3 only if "`comvar = 2`"
b) it can only execute Region 5 if "`comvar = 2`"
c) execution of Region 5 is the only way to make `comvar` again ≠ 2.

The only assignment "`comvar:= 2`" occurs at the end of Region 2. As a result each Region 2 can only be followed by a Region 5 and, conversely, each Region 5 must be preceded by a Region 2. This sequencing allows us to use the transmission variable "`asknum`", which is set in Region 2 and inspected in Region 5.

For the uses of the transmission variables "`operanswer`" an analogous analysis can be made. Region 2 will be followed by Region 5 (see above); if here the final answer (A1 or A2) is interpreted, `operanswer[i]` is set before "`V(procsem[i])`", so that the transmission variable has been set properly before the process can (and will) enter Region 3, where its "`operanswer`" will be inspected. If in Region 5 the answer A3 is detected, the message interpreter set for this process "`procvar[i]:= 2`", thus allowing *once* in Region 6 the answer A4 or A5 for this process. Again "`V(procsem[i])`" is only performed after the assignment to `operanswer`. Thus we have verified that
a) `operanswer` is only set once by the message interpreter after a request in Region 2.
b) this `operanswer` will only be inspected in the following Region 3 after the request to set it has been fulfilled (in Region 5 or Region 6).
This completes the soundness of the use of the transmission variables "operanswer".

Inspection of the message interpreter (particularly the scheme of its states) shows
a) that a rejected message (Region 4) sooner or later is bound to give rise to Region 6
b) that wrong messages are ignored, giving the operator the opportunity to correct.

By the above analysis we hope to have created sufficient confidence in the correctness of our construction. The analysis fallowed the steps already hinted at in section 5.2.1: after creation of the critical sections (with the aid of `mutex`), the latter are used to sequence Regions properly, thanks to which sequencing the transmission variables can be used unambiguously.

6. The Problem of the Deadly Embrace.

In the introductory part of this section I shall draw attention to a rather logical problem that arises in the cooperation between various processes, when they have to share the same facilities. We have selected this problem for various reasons. Firstly it is a straightforward extension af the sound principle that no two persons can use a single compartment of a revolving door simultaneously. Seondly, its solution, which I regard as non-trivial and that will be given in section 6.1, gives us a nice example of more subtle cooperation rules than we have met before. Thirdly. it gives us the opportunity to illustrate (in section 6.2) a programming technique by which a further gain in clarity can be achieved.

Let me first give an example of the kind of sharing I have in mind.

As "processes" we might take "programs", describing some computational process to be performed by a computer. Execution of such a computational process takes time, during which information must be stored in the computer. We restrict ourselves to thoses processes of which is known in advance
1) the maximum demand on storage space and
2) that the computational process will end, provided that storage space requested by the process will be put at the disposal of the computational process. The ending of the computational process will imply that its demand on storage space will reduce to zero.

We assume that the available store has been subdivided into fixed size "pages" which, from the point of view of the programs can be regarded as equivalent.

The actual demand on storage space, needed by a process, may be a function varying in time as the process proceeds —subject, of course, to the a priori known upper bound. We assume that the individual processes request from and return to "available store" in single page units. With "equivalence" (see the last word of the previous paragraph) is meant that a process, requiring a new page only asks for "a new page" but never for a special one or one out of a special group.

We now request that a process, once initiated, will get the opportunity —sooner or later— to complete its action and reject any organization in which it may happen that a process may have to be killed half way its activity, thereby throwing away the computation time already invested in it.

If the computer has to perform the different processes one after the other, the only condition that must be satisfied by a process is that its maximum demand does not exceed the total storage capacity.

If, however, the computer can serve more than one process simultaneously, one can adhere to the rule that one only admits programs as long as the sum of their maximum demands does not exceed the total storage capacity. This rule, safe though it is, is unnecessarily restrictive, for it means that each process effectively occupies its maximum demand during the complete time of its execution. When we consider the following table (in which we regard the processes as "borrowing" pages from available store)

process	maximum demand	present loan		further claim
P1	80	40		40
P2	60	20 +		40
	available store = 100	- 60	= 40	

(a total store of 100 pages is assumed), we have a situation in which is still nothing wrong. If, however, both process request their next page and they should both get it, we should get the following situation:

process	maximum demand	present loan		further claim
P1	80	41		39
P2	60	21 +		39
	available store = 100	- 62	= 38	

This is an unsafe situation, for both processes might want to realize their full further claim before returning a single page to available store. So each of them may first need a further 39 pages, while there are only 38 available.

This situation, when one process can only continue provided the other one is killed first, is called "The Deadly Embrace". The problem to be solved is: how can we avoid the danger of the Deadly Embrace without being unrecessarily restrictive.

6.1. The Banker's Algorithm.

A banker has a finite capital expressed in florins. He is willing to accept customers, that may borrow florins from him on the following conditions.

1. The customer makes the loan for a transaction that will be completed in a finite period of time.

2. The customer must specify in advance his maximum "need" for florins for this transaction.

3. As long as the "loan" does not exceed the "need" stated in advance, the customer can increase or decrease his loan florin wise.

4. A customer may not complain, if he asks for an increase of the current loan and receives from the banker the answer "If I gave you the florin you ask for you would not exceed your stated need and therefore you are entitled to a next florin. At present, however, it is somewhat inconvenient for me to pay you, but I promise to send you the florin in due time."

5. His guarantee that this moment will indeed arrive is founded on the banker's cautiousness and the fact that his co-customers are subjected to the same condition as he: that as soon as a customer has got the florin he asked for he will proceed with his transactions at a non-zero speed, i.e. within a finite period of time he will ask for a next florin or will return a florin or will finish the transaction, which implies that his complete loan has been returned (florin by florin).

The primary questions are

a) under which conditions can the banker make the contract with a new customer?

b) under which conditions can the banker pay a (next) florin to a requesting customer without running into the danger of the Deadly Embrace?

The answer to question a) is simple: he can accept any customer, whose stated need does not exceed the banker's capital.

To answer question b) we introduce the following terminology.

The banker has a fixed "capital" at his disposal; each new customer states in advance his maximum "need" and for each customer will hold

```
need[i] ≤ capital
```
(for all `i`).

The current situation for each customer is characterized by his "`loan`". Each `loan` is initially = 0 and shall satisfy at any instant

```
0 ≤ loan[i] ≤ need[i]
```
(for all `i`).

A useful quantity to be derived from this is the maximum further "`claim`", given by

```
claim[i] = need[i] - loan[i]        (for all i).
```

Finally the banker notes the amount in "cash", given by

```
cash = capital - sum of the loans.
```

Obviously

```
0 ≤ cash ≤ capital
```

has to hold.

In order to decide, whether a requested florin can be paid to the customer, the banker essentially inspects the situation that would arise if he had paid it. If this situation is "safe", then he pays the florin, if the situation is not "safe", he has to say: "Sorry, but you have to wait.".

Inspection, whether a situation is safe amounts to inspection, whether all customer transactions can be guaranteed to be able to finish. The algorithm starts to investigate whether at least one customer has a claim not exceeding `cash`. If so, this customer can complete his transactions and therefore the algorithm investigates the remaining customers as if the first one had finished and returned its complete loan. Safety of the situation means, that all transactions can be finished, i.e. that the banker sees a way of getting all his money back.

If the customers are numbered from 1 through *N*, the routine inspecting a situation can be written as follows:

```
integer free money; Boolean safe; Boolean array finish doubtful[1:N];
free money:= cash;
for i:= 1 step 1 until N do finish doubtful[i]:= true;
L:for i:= 1 step 1 until N do
begin if finish doubtful[i] and claim[i] ≤ free money then
      begin finish doubtful[i]:= false;
```

```
                free money:= free money + loan[i]; goto L
        end
end;
if free money = capital then safe:= true else safe:= false".
```

The above routine inspects any situation. An improvement of the Algorithm has been given by L.Zwanenburg, who takes into account that the only situations to be investigated are those, where, starting from a safe situation, a florin has been tentatively given to `customer[j]`. As soon as "`finish doubtful[j]:= false`" can be executed the algorithm can decide directly on safety of the situation, for apparently this tempted payment was reversible: This short cut will be implemented in the program in the next section.

6.2. The Banker's Algorithm Applied.

In this example, the florins are processes as well. (Each florin, say, represents the use of a magnetic tape deck; the loan of a florin is then the permission to use one of the tape decks.)

We assume, that the customers are numbered from 1 through *N* and that the florins are numbered from 1 through *M*. Each customer has a variable "florin number" in which, after each granting of a florin, it can find the number of the florin it has just borrowed; also each florin has a variable "customer number" in which it can find by which customer it has been borrowed.

Each customer has a state variable "`cusvar`", where "`cusvar` = 1" means "I am anxious to borrow." (otherwise "`cusvar` = 0); each florin has a state variable "`flovar`", where "`flovar` = 1" means "I am anxious to get borrowed, i.e. I am in cash." (otherwise "`flovar`= 0"). Each customer has a binary semaphore "`cussem`", each florin has a binary semaphore "`flosem`", which will be used in the usual manner.

We assume that each florin is borrowed and returned upon customer indication, but that he cannot finish the loan of a florin immediately. After the customer has indicated that he has no further use for this florin, the florin may not be instantaneously available for a next use. It is, as if the customer can say to a borrowed florin "run home to the banker". The actual loan will only be ended after the florin has indeed returned into cash: of its return into the banker's cash it will signal the customer from which it came via a customer semaphore "`florin returned`". A P-operation on this semaphore should guard the customer for an inconscious overdraft. Before each florin request the customer will perform a P-operation on its "`florin returned`"; the initial value of "`florin returned`" will be "= `need`".

We assume that the constant integers "`N`" and "`M`" (=`capital`) and the constant integer array "`need`" are declared and defined in the universe in which the following program is embedded.

The procedure "`try to give to`" is made into a Boolean procedure, the value of which indicates whether a delayed request for a florin has been granted. In the florin program it is exploited that returning a florin may at most give rise to a single delayed request now being granted. (If more than one type of facility is shared under control of the banker, this will no longer hold. Jumping out of the for loop to the statement labeled "leave" at the end of the florin program is then not permissible.)

```
begin integer array loan, claim, cussem, cusvar, florin number, florin
                    returned[1 :N],
                    flosem, flovar, customer number[1:M];
        integer mutex, cash, k;
        Boolean procedure try to give to (j); value j; integer j;
```

```
      begin if cusvar[j] = 1 then
            begin integer i, free money;
                  Boolean finish doubtful[l:N];
                  free money:= cash - 1;
                  claim[j]:= claim[j] - 1; loan[j]:= loan[j] + 1;
                  for i:= 1 step 1 until N do finish doubtful[i]:= true;
              LO: for i:= l step l until N do
                  begin if finish doubtful[i] and claim[i] [lte] free money then
                        begin if ≠ j then
                              begin finish doubtful[i]:= false;
                                    free money:= free money + loan[i];
                                    goto LO
                              end
                                     else
                              begin comment Here more sophisticated ways for
                                    selecting a free florin may be implemented;
                                    i:= 0;
                                L1: i:= i + 1; if flovar[i] = 0 then goto L1;
                                    florin number[j]:= i;
                                    customer number[i]:= j;
                                    cusvar[j]:= 0; flovar[i]:= 0:
                                    cash:= cash - 1;
                                    try to give to:= true;
                                    V(cussem[j]); V(flosem[i]); goto L2
                              end
                        end
                  end;
                  claim[j]:= claim[j] + 1; loan[j]:= loan[j] -1
            end;
            try to give to:= false;
L2:   end;
      mutex:= 1; cash:= M;
      for k:= 1 step 1 until N do
      begin loan[k]:= 0; cussem[k]:= 0; cusvar[k]:= 0; claim[k]:= need[k];
            florin returned[k]:= need[k]
      end;
      for k:= 1 step 1 until M do
      begin flosem[k]:= 0; flovar[k]:= 1 end;
      parbegin
customer 1: begin...................end;
              .
              .
              .
              .
              .
customer N: begin...................end;
florin 1:   begin...................end;
              .
              .
              .
              .
              .
florin M: begin...................end;
      parend
end
```

In customer "n", the request for a new florin consists of the following sequence of statements:

```
          P(florin returned[n]);
          P(mutex);
          cusvar[n]:= 1; try to give to (n);
          V(mutex);
          P(cussem[n]) ;
```

after completion of the last statement "`florin number[n]`" gives the identity of the florin just borrowed, the customer has the opportunity to use it and the duty to return it in due time to the banker.

The structure of a florin is as follows:

```
florin m:
begin integer h;
start:P(flosem[m]);
      "Now "customer number[m]" identifies the customer that has borrowed it.
      The florin can serve that customer until it has finished the task
      required from it during this loan. To return itself to the cash, the
      florin proceeds as follows:"

      claim[customer number[m]]:= claim[customer number[m]] + 1;
      loan[customer number[m]]:= loan[customer number[m]] - 1;
      flovar[m]:= 1 ; cash:= cash + 1;
      V(florin returned[customer number[m]]);
      for h:= 1 step 1 until N do
            begin if try to give to(h) then goto leave end;
leave:V(mutex);
      goto start
end
```

Remark. Roughly speaking a successful loan can only take place when two conditions are satisfied: the `florin` must be requested and the `florin` must be available. In this program the mechanism of `cusvar` and `cussem` is also used (by the customer), when the requested `florin` is immediately available, likewise the mechanism of `flovar` and `flosem` is also used (by the `florin`) if, after its return to cash, it can immediately be borrowed again by a waiting customer. This programming technique has been suggested by C.Ligtmans and P.A.Voorhoeve, and I mention it because in the case of more intricate rules of cooperation it has given rise to a simplification that proved to be indispensable. The underlying cause of this increase in simplicity it that the dynamic way through the topological structure of the program no longer distinguishes between an actual delay or not, just as in the case of the P-operation itself.

7. Concluding Remarks.

In the literature one sometimes finds a sharp distinction between "concurrent programming" —more than one central processor operating on the same job— and "multiprogramming" —a single processor dividing its time between different jobs—. I have always felt that this distinction was rather artificial and therefore confusing. In both cases we have, macroscopically speaking, a number of sequential processes that have to cooperate with each other and our discussions on this cooperation apply equally well to "concurrent programming" as to "multiprogramming" or any mixture of the two. What in concurrent programming is spread out in space (e.g. equipment) is in multiprogramming spread out in time: the two present themselves as different implementations of the same logical structure and I regard the development of a tool to describe and form such structures themselves, i.e. independent of these implementational differences, as one of the major contributions of the work from which this monograph has been born. As a specific example of this unifying train of thought I should like to mention —for those that are only meekly interested in multiprocessors, multiprogramming and the like— the complete symmetry between a normal sequential computer on the one hand and its periferal gear on the other (as displayed, for instance, in Section 4.3: "The Bounded Buffer").

Finally I should like to express, once more, my concern about the correctness of programs, because I am not too sure, whether all of it is duly reflected in what I have written.

If I suggest methods by which we could try to attain a greater security, then this is of course more psychology than, say, mathematics. I have the feeling that for the Human Mind it is just terribly hard to think in terms of processing evolving in time and that our greatest aid in controling them is by attaching meanings to the values of identified quantities. For instance, in the program section

```
    i:= 10;
LO: x:= sqrt(x); i:=i - 1;
    if i > 0 then goto LO
```

we conclude that the operation "`x:=sqrt(x)`" is repeated ten times, but I have the impression that we can do so by attaching to "`i`" the meaning of "the number of times that the operation "`x:=sqrt(x)`" still has to be repeated". (I know that in discussing progran' verification, Dr.P.Maur has introduced the term "the general snapshot"; in all probability we have here a trivial example of it.) But we should be aware of the fact that such a timeless meaning (a statement of fact or relation) is not permanently correct: immediately after the execution of "`x:=sqrt(x)`" but before that of the subsequent "`i:= i - 1`" the value of "`i`" is "one more than the number of times that the operation "`x:= sqrt(x)`" still has to be repeated". In other words: we have to specify at what stages of the process such a meaning is applicable and, of course, it must be applicable in every situation where we rely on this meaning in the reasoning that convinces us of the desired overall performance of the program.

In purely sequential programming, as in the above example, the regions of applicability of such meanings are usually closely connected with places in the program text (if not, we have just a tricky and probably messy program). In multiprogramming we have seen —in particular in Section 5.2.1— that it is a worth-while effort to create such regions of applicability of meaning very consciously. The recognition of the hierarchical difference between the presence of a message and the message itself, here forced upon us, might give a clue even to clearer uniprogramming.

For example. if I am married to one out of ten wives, numbered from 1 through 10, this fact may be represented by the value of a variable "*wife number*", associated with me. If I may also be single, it is a commonly used programmer's device to code the state of the bachelor as an eleventh value, say "*wife number* = 0". The meaning of the value of this variable then becomes 'If my *wife number* is = 0, then I am single, otherwise it gives the number of my wife." The moral is that the introduction of a separate Boolean variable "*married*" might have been more honest.

We know that the von Neumann type machine derives its power and flexibility from the fact that it treats all words in store on the same footing. It is often insufficiently realized that, thereby, it gives the user the duty to impose structure wherever recognizable.

Sometimes it is. It has often been quoted as The Great Feature of the Von Neumann type machine that it can modify its own instructions, but most modern algorithmic translators, however, create an object program that remains in its entire execution phase just as constant as the original source text. Instead of chaotically modifying its own instructions just before or after their execution, creation of instructions and execution of these instructions now occur in different sequenced regions: the translation phase and the execution phase. And this for the benefit of us all.

It is my firm belief that in each process of some complexity the variables occurring in it admit analogous hierarchical orderings and that, when these hierarchies are clearly recognizable in the program text, the gain in clarity of the program and in efficiency of the implementation will be considerable. If this monograph gives any reader a clearer indication of what kind of hierarchical ordering can be expected to be relevant, I have reached one of my dearest goals. And may we not hope, that a confrontation with the intricacies of Multiprogramming gives us a clearer understanding of what Uniprogramming is all about?

---

transcribed by Nick James and Ham Richards
revised Tue, 4 Dec 2007