1

COOPERATING SEQUENTIAL PROCESSES

EDSGER W. DIJKSTRA

(1965)

INTRODUCTION

This chapter is intended for all those who expect that in their future activities they will become seriously involved in the problems that arise in either the design or the more advanced applications of digital information processing equipment; they are further intended for all those who are just interested in information processing.

The applications are those in which the activity of a computer must include the proper reaction to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in process control, traffic control, stock control, banking applications, automatization of information flow in large organizations, centralized computer service, and, finally, all information systems in which a number of computers are coupled to each other.

The desire to apply computers in the ways sketched above has often a strong economic motivation, but in this chapter the not unimportant question of efficiency will not be stressed too much. Logical problems which arise, for example, when speed ratios are unknown, communication possibilities restricted, etc., will be dealt with much more. This will be done in order to create a clearer insight into the origin of the difficulties one meets and into the nature of solutions. Deciding whether under given circumstances

E. W. Dijkstra, Cooperating sequential processes. Technological University, Eindhoven, The Netherlands, September 1965. Reprinted in *Programming Languages*, F. Genuys, Ed., Academic Press, New York, 1968, 43–112.

P. B. Hansen (ed.), *The Origin of Concurrent Programming*

the application of our techniques is economically attractive falls outside the scope of this chapter.

There will not be a fully worked out theory, complete with Greek letter formulae, so to speak. The only thing that can be done under the present circumstances is to offer a variety of problems, together with solutions. And in discussing these we can only hope to bring as much system into it as we possibly can, to find which concepts are relevant, as we go along.

1 ON THE NATURE OF SEQUENTIAL PROCESSES

Our problem field proper is the co-operation between two or more sequential processes. Before we can enter this field, however, we have to know quite clearly what we call "a sequential process". To this preliminary question the present section is devoted.

To begin, here is a comparison of two machines to do the same example job, the one a non-sequential machine, the other a sequential one.

Let us assume that of each of four quantities, named `a[1]`, `a[2]`, `a[3]`, and `a[4]` respectively, the value is given. Our machine has to process these values in such a way that, as its reaction, it "tells" us which of the four quantities has the largest value. E.g. in the case:

```
a[1] = 7, a[2] = 12, a[3] = 2, a[4] = 9
```

the answer to be produced is `a[2]` (or only `2`, giving the index value pointing to the maximum element).

Note that the desired answer would become incompletely defined if the set of values were—in order—`7, 12, 2, 12`, for then there is no unique largest element, and the answer `a[2]` would have been as good (or as bad) as `a[4]`. This is remedied by the further assumption that of the four values given, no two are equal.

Remark 1. If the required answer would have been the maximum value occurring among the given ones, the last restriction would have been superfluous, for the answer corresponding to the value set `7, 12, 2, 12` would then have been `12`.

Remark 2. Our restriction "Of the four values no two are equal" is still somewhat loosely formulated, for what do we mean by "equal"? In the processes to be constructed pairs of values will be compared with one another, and what is really meant is that every two values will be sufficiently different, so that the comparator will unambiguously decide which of the two

is the larger one. In other words, the difference between any two must be large compared with "the resolving power" of our comparators.

We shall first construct our non-sequential machine. When we assume our given values to be represented by currents we can imagine a comparator consisting of a two-way switch, the position of which is schematically controlled by the currents in the coils of electromagnets, as in Figs. 1 and 2.

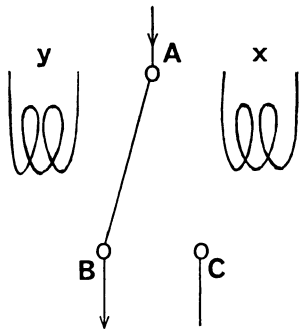


Fig. 1. $x<y$

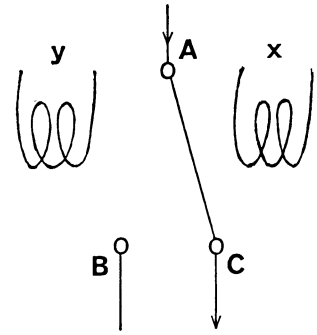
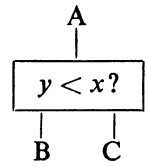


Fig. 2. $y<x$

When current y is larger than current x, the left electromagnet pulls harder than the right one and the switch switches to the left (Fig. 1) and the input A is connected to output B; if current x is the larger one we shall get the situation (Fig. 2), where the input A is connected to output C.

In our diagrams we shall omit the coils and shall represent such a comparator by a small box



only representing at the top side the input and at the bottom side the two outputs. The currents to be led through the coils are identified in the question written inside the box, and the convention is that the input will be connected to the right-hand side output when the answer to the question is "Yes", to the left-hand side output when the answer is "No".

Now we can construct our machine as indicated in Fig. 3. At the output side we have drawn four indicator lamps, one, and only one, of which will

light up to indicate the answer.

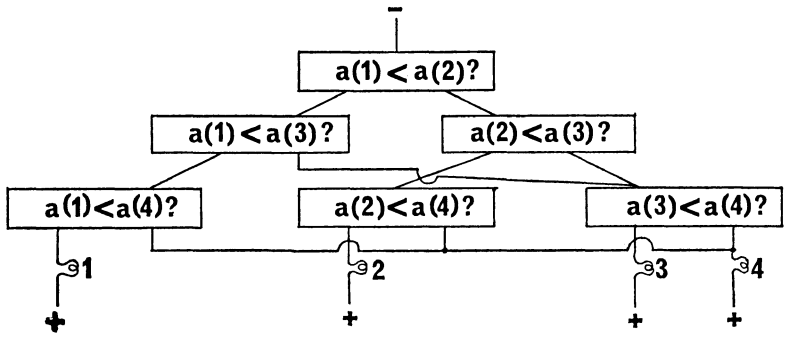


Fig. 3

In Fig. 4 we indicate the position of the switches when the value set 7, 12, 2, 9 is applied to it. In the boxes the positions of the switches are indicated, wires not connected to the input are drawn dotted.

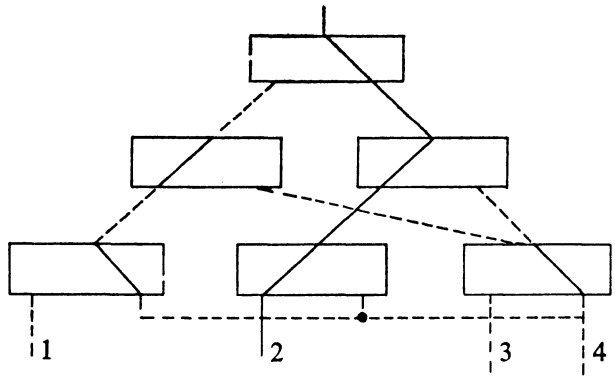


Fig. 4

We draw the reader's attention to the fact that now only the positions of the three switches that connect output 2 to the input matter; the reader is invited to convince himself that the position of the other three switches is indeed immaterial.

It is also worthwhile to give a moment's attention to see what happens in time when our machine of Fig. 3 is fed with four "value currents". Obviously it cannot be expected to give the correct answer before the four value currents start going through the coils. But one cannot even expect it to indicate the correct answer as soon as the currents are applied, for the switches must get

into their correct position, and this may take some time. In other words, as soon as the currents are applied (simultaneously or the one after the other) we must wait a period of time —characteristic for the machine—and only after that the correct answer will be shown at the output side. What happens during this waiting time is immaterial, provided that the interval is long enough for all switches to find their final position. They may start switching simultaneously, the exact order in which they attain their final position is immaterial, and therefore we shall no longer pay any attention to it.

From the logical point of view the switching time can be regarded as a marker on the time axis: before it the input data have to be supplied, after it the answer is available.

In the use of our machine the progress of time is only reflected in the obvious "before-after" relation, which tells us that we cannot expect an answer before the question has been properly put. This sequence relation is so obvious (and fundamental) that it cannot be regarded as a characteristic property of our machine. And our machine is therefore called a "non-sequential machine" to distinguish it from the kind of equipment—or processes that can be performed by it—to be described now.

Up till now we have interpreted the diagram of Fig. 3 as the (schematic) picture of a machine to be built in space. But we can interpret this same diagram in a very different manner if we place ourselves in the mind of the electron entering at the top input and wondering where to go. First, it finds itself faced with the question whether `a[1] < a[2]` holds. Having found the answer to this question, it can proceed. Depending on the previous answer, it will enter one of the two boxes `a[1] < a[3]` or `a[2] < a[3]`, i.e. it will only know what to investigate next, after the first question has been answered. Having found the answer to the question selected from the second line, it will know which question to ask from the third line and, having found this last answer, it will now know which bulb should start to glow. Instead of regarding the diagram of Fig. 3 as that of a machine, the parts of which are spread out in space, we have regarded it as rules of behaviour, to be followed in time.

With respect to our earlier interpretation two differences are highly significant. In the first interpretation all six comparators started working simultaneously, although finally only three switch positions were relevant. In the second interpretation only three comparisons are actually evaluated—the wondering electron asks itself three questions—but the price of this gain

is that they have to be performed the one after the other, as the outcome of the previous one decides what to ask next. In the second interpretation three questions have to be asked in *sequence*, the one after the other. The existence of such an order relation is the distinctive feature of the second interpretation, which in contrast to the first one is therefore called "a sequential process". We should like to make two remarks.

Remark 3. In actual fact, the three comparisons will each take a finite amount of time ("switching time", "decision time", or, in the jargon, "execution time"), and as a result the total time taken will at least be equal to the sum of these three execution times. We stress once more that for many investigations these executions can be regarded as ordered markers on a scaleless time axis and that it is only the relative ordering that matters from this (logical) point of view.

Remark 4. As a small side line we note that the two interpretations (call them "simultaneous comparisons" and "sequential comparisons") are only extremes. There is a way of, again, only performing three comparisons, in which two of them can be done independently from one another, i.e. simultaneously; the third one, however, can be done only after the other two have been completed. It can be represented with the aid of a box in which two questions are put and which, as a result, has four possible exits, as in Fig. 5.

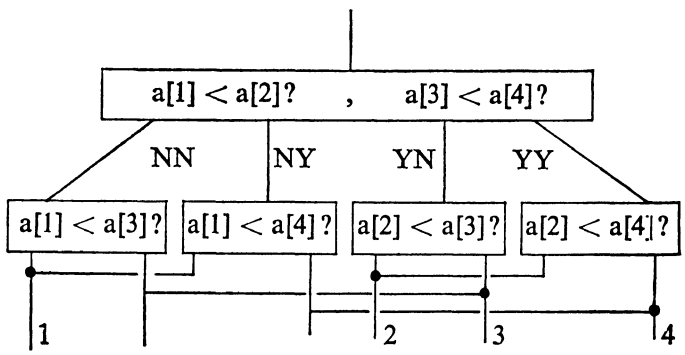


Fig. 5

The total time taken will be at least the sum of the comparison execution times. The process is of the first kind in the sense that the first two comparisons can be performed simultaneously, it is of sequential nature, as the third comparison can be selected from the second line only when the first two have both been completed.

We return to our purely sequential interpretation. Knowing that the diagram is meant for purely sequential interpretations, we can take advantage of this circumstance to make the description of the "rules of behaviour" more compact. The idea is that the two questions on the second line only one of which will be actually asked are highly similar: the questions on the same line differ only in the subscript value of the left operand of the comparison. And we may ask ourselves: "Can we map the questions on the same line of Fig. 3 on to a single question?"

This can be done, but it implies that the part that varies along a line—i.e. the subscript value in the left operand— must be regarded as a parameter, the task of which is to determine which of the questions mapped on each other is meant, when its turn to be executed has come. Obviously the value of this parameter must be defined by the past history of the process.

Such parameters, in which past history can be condensed for future use, are called "variables". To indicate that a new value has to be assigned to it we use the so-called assignment operator := (read: "becomes"), a kind of directed equality sign which defines the value of the left-hand side in terms of the value of the right-hand side.

We hope that the previous paragraph is sufficient for the reader to recognize also in the diagram of Fig. 6 a set of "rules of behaviour". Our variable is called `i`; and the reader may wonder why the first question, which is invariably `a[1] < a[2] ?` is not written that way, but with patience he will understand.

When we have followed the rules of Fig. 6 as intended from top till bottom, the final value of `i` will identify the maximum value, viz. `a[i]`.

The transition from the scheme of Fig. 3 to the one of Fig. 6 is a drastic change, for the latter's "rules of behaviour" can only be interpreted sequentially. And this is due to the introduction of the variable `i`: having only `a[1]`, `a[2]`, `a[3]`, and `a[4]` available as values to be compared, the question `a[i] < a[2] ?` is meaningless, unless it is known for which value of `i` this comparison has to be made.

Remark 5. It is somewhat unfortunate that the jargon of the trade calls the thing denoted by `i` a variable, because in normal mathematics the concept of a variable is a completely timeless concept. Time has nothing to do with the `x` in the relation

```
sin(2 * x) = 2 * sin(x) * cos(x)
```

if such a variable ever denotes a value it denotes "any value".

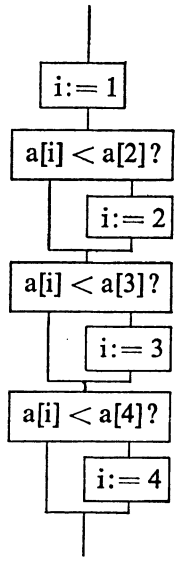


Fig. 6

Each time, however, that a variable in a sequential process is used—such as `i` in `a[i]`—it denotes a very specific value, viz. the last value assigned to it, and nothing else! As long as no new value is assigned to a variable, it denotes a constant value!

Remark 6. One may well ask what we are actually doing when we introduce a variable without specifying, for instance, a domain for it, i.e. a set of values which is guaranteed to comprise all its future actual values. We shall not pursue this question here.

Now we are going to subject our scheme to a next transformation. In Fig. 3 we have "wrapped up" the lines, now we are going to wrap up the scheme of Fig. 6 in the vertical direction, an operation to which we are invited by the repetitive nature of it and which can be performed at the price of a next variable, `j` say.

The change is a dramatic one, for the fact that the original problem was to identify the maximum value among *four* given values is no longer reflected in the "topology" of the rules of behaviour: in Fig. 7 we only find the number `4` mentioned once. By introducing another variable, say `n`, and replacing the `4` in Fig. 7 by `n` we have suddenly the rules of behaviour to identify the maximum occurring among the `n` elements `a[1], a[2], ..., a[n]`, and this practically only for the price that before application the variable `n` must

be given its proper value.

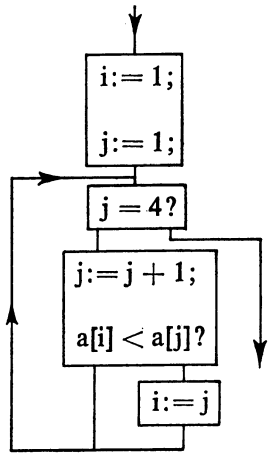


Fig. 7

The change is dramatic, for now we have not only given rules of behaviour which must be interpreted sequentially this was already the case with Fig. 6 but we have devised a single mechanism for identifying the maximum value among any number of given elements, whereas our original non-sequential machine could only be built for a previously well-defined number of elements. We have mapped our comparisons in time instead of in space, and if we wish to compare the two methods it is as if the sequential machine "extends itself" in terms of Fig. 3 as the need arises. It is our last transition which displays the sequential processes in their full glory.

The technical term for what we have called "rules of behaviour" is an algorithm or a program. (It is not customary to call it "a sequential program", although this name would be fully correct.) Equipment able to follow such rules, "to execute such a program" is called "a general-purpose sequential computer" or "computer" for short; what happens during such a program execution is called "a sequential process".

There is a commonly accepted technique of writing algorithms without the need of pictures such as we have used, viz. ALGOL 60 ("ALGOL" being short for Algorithmic Language). For a detailed discussion of ALGOL 60 I must refer the reader to the existing literature. We shall use it in future, whenever convenient for our purposes.

For the sake of illustration we shall describe the algorithm of Fig. 7 (but for n instead of 4) by a sequence of ALGOL statements:

```
      i:= 1; j:= 1;
back: if j <> n then
        begin j:= j + 1;
              if a[i] < a[j] then i:= j;
              goto back;
        end
```

The first two statements: `i:= 1; j:= 1` are—one hopes—self-explanatory. Then comes `back:`, a so-called label, used to identify this place in the program. Then comes `if j <> n then`, a so-called conditional clause. If the condition expressed by it is satisfied the following statement will be performed, otherwise it will be skipped. (Another example of it can be found two lines lower.) When the extent of the program which may have to be skipped presents itself primarily as a sequence of more than one statement, then one puts the so-called statement brackets `begin` and `end` around this sequence, thereby making it into a single statement as far as its surroundings are concerned. (This is entirely analogous to the effect of parentheses in algebraic formulae, such as `a * (b + c)` where the parenthesis pair indicates that the whole expression contained within it is to be taken as factor.) The last statement `goto back` means that the process should be continued at the point thus labelled; it does exactly the same thing for us as the upward-pointing line of Fig. 7.

2 LOOSELY CONNECTED PROCESSES

The subject matter of this chapter is the co-operation between loosely connected sequential processes, and this section will be devoted to a thorough discussion of a simple, but representative problem, in order to give the reader some feeling for the problems in this area.

In the previous section we have described the nature of a single sequential process, performing its sequence of actions autonomously, i.e. independent of its surroundings as soon as it has been started.

When two or more of such processes have to co-operate with each other they must be connected, i.e. they must be able to communicate with each other in order to exchange information. As we shall see below, the properties of these means of intercommunication play a vitaI role.

Furthermore, we have stipulated that the processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. In particular, we disallow any assumption about the relative speeds of the different processes. (Such an

assumption—say, "processes geared to the same clock"—could be regarded as implicit intercommunication.) This independence of speed ratios is in strict accordance with our appreciation of the single sequential process: its only essential feature is that its elementary steps are performed in sequence If we prefer to observe the performance with a chronometer in our hand we may do so, but the process itself remains remarkably unaffected by this observation.

The consistent refusal to make any assumptions about the speed ratios will at first sight appear to the reader as a mean trick to make things more difficult than they already are. I feel, however, fully justified in my refusal. First, we may have to cope with situations in which, indeed, very little is known about the speeds. For instance, part of the system may be a manually operated input station, another part of the system might be such that it can be stopped externally for any period of time, thus reducing its speed temporarily to zero. Secondly—and this is much more important—when we think that we can rely upon certain speed ratios we shall discover that we have been "penny wise and pound foolish". It is true that certain mechanisms can be made simpler under the assumption of speed-ratio restrictions. The verification, however, that such an assumption is always justified is, in general, extremely tricky and the task to make, in a reliable manner, a well-behaved structure out of many interlinked components is seriously aggravated when such "analogue interferences" have to be taken into account as well. (For one thing: it will make the proper working a rather unstable equilibrium, sensitive to any change in the different speeds, as may easily arise by replacement of a component by another—say, replacement of a line printer by a faster model—or reprogramming of a certain portion.)

2.1 A Simple Example

In considering two sequential processes, `process 1` and `process 2`, they can for our purposes be regarded as cyclic. In each cycle a so-called "critical section" occurs, critical in the sense that at any moment at most one of the two processes is allowed to be engaged in its critical section. In order to effectuate this mutual exclusion, the two processes have access to a number of common variables. We postulate that inspecting the present value of such a common variable and assigning a new value to such a common variable are to be regarded as indivisible, non-interfering actions, i.e. when the two processes assign a new value to the same common variable "simultaneously", then the assignments are to be regarded as done the one after the other, the

final value of the variable will be one of the two values assigned, but never a "mixture" of the two. Similarly, when one process inspects the value of a common variable "simultaneously" with the assignment to it by the other one, then the former process will find either the old or the new value, but never a mixture.

For our purposes ALGOL 60 as it stands is not suited, as ALGOL 60 has been designed to describe one single sequential process. We therefore propose the following extension to enable us to describe parallelism of execution. When a sequence of statements—separated by semicolons as usual in ALGOL 60—is surrounded by the special statement bracket pair `parbegin` and `parend` this is to be interpreted as parallel execution of the constituent statements. The whole construction—let us call it "a parallel compound"—can be regarded as a statement. Initiation of a parallel compound implies simultaneous initiation of all its constituent statements, its execution is completed after the completion of the execution of all its constituent statements. E.g.:

```
begin S1; parbegin S2; S3; S4 parend; S5 end
```

(in which `S1`, `S2`, `S3`, `S4`, and `S5` are used to indicate statements) means that after the completion of `S1`, the statements `S2`, `S3`, and `S4` will be executed in parallel, and only when they are all finished will the execution of statement `S5` be initiated.

With the above conventions we can describe our first solution:

```
begin integer turn; turn:= 1;
      parbegin
      process 1: begin L1: if turn = 2 then goto L1;
                           critical section 1;
                           turn:= 2;
                           remainder of cycle 1, goto L1
                 end;
      process 2: begin L2: if turn = 1 then goto L2;
                           critical section 2;
                           turn:= 1;
                           remainder of cycle 2; goto L2
                 end;
      parend
end
```

(Note for the inexperienced ALGOL 60 reader. After `begin` in the first line we find the so-called declaration `integer turn`, thereby sticking to the rule of ALGOL 60 that program text is not allowed to refer to variables

without having introduced them with the aid of a declaration. As this declaration occurs after the `begin` of the outermost statement bracket pair, it means that for the whole duration of the program a variable has been introduced that will only take on integer values and to which the program text can refer by means of the name `turn`.)

The two processes communicate with each other via the common integer `turn`, the value of which indicates which of the two processes is the first to perform (or rather: to finish) its critical section. From the program it is clear that after the first assignment the only possible values of the variable `turn` are 1 and 2. The condition for process 2 to enter its critical section is that it finds at some moment `turn <> 1`, i.e. `turn = 2`. But the only way in which the variable `turn` can get this value is by the assignment `turn:= 2` in process 1. As process 1 performs this assignment only at the completion of its critical section, process 2 can only initiate its critical section after the completion of critical section 1. And critical section 1 could indeed be initiated, because the initial condition `turn = 1` implied `turn <> 2`, so that the potential wait cycle, labelled `L1`, was initially inactive. After the assignment `turn:= 2` the roles of the two processes are interchanged. (N.B. It is assumed that the only references to the variable `turn` are the ones explicitly shown in the program.)

Our solution, though correct, is, however, unnecessarily restrictive: after the completion of critical section 1 the value of the variable `turn` becomes 2, and it must be = 1 again, before the next entrance into critical section 1.

As a result, the only admissible succession of critical sections is the strictly alternating one `1, 2, 1, 2, 1, 2, 1, ...`; in other words, the two processes are synchronized In order to stress explicitly that this is not the kind of solution we wanted, we impose the further condition: "If one of the processes is stopped well outside its critical section, this is not allowed to lead to potential blocking of the other process." This makes our previous solution unacceptable, and we have to look for another.

Our second effort works with two integers `c1` and `c2`, where `c1, c2 = 0/1` respectively will indicate that the corresponding process is inside/outside its critical section respectively. We may try the following construction:

```
begin integer c1, c2;
      c1:= 1; c2:= 1;
      parbegin
      process1: begin L1: if c2 = 0 then goto L1;
                          c1:= 0;
                          critical section 1;
                          c1:= 1;
                          remainder of cycle 1; goto L1
                end;
      process2: begin L2: if c1 = 0 then goto L2;
                          c2:= 0;
                          critical section 2;
                          c2:= 1;
                          remainder of cycle 2; goto L2
                end
      parend
end
```

The first assignments set both c's = 1, in accordance with the fact that the processes are started outside their critical sections. During the entire execution of critical section 1 the relation `c1 = 0` holds, and the first line of process 2 is effectively a wait: "Wait as long as process 1 is in its critical section." The trial solution gives indeed some protection against simultaneity of critical section execution, but is, alas, too simple, because it is wrong. Let first process 1 find that c2 = 1; let process 2 inspect c1 immediately afterwards, then it will (still) find c1 = 1. Both processes, each having found that the other is not in its critical section, will conclude that they can enter their own critical section safely!

We have been too optimistic, we must play a safer game. Let us invert, at the beginning of the parallel processes, the inspection of the `c` of the other and the setting of the own `c`. We then get the construction:

```
begin integer c1, c2;
      c1:= 1; c2:= 1;
      parbegin
      process 1: begin A1: c1:= 0;
                      L1: if c2 = 0 then goto L1;
                          critical section 1;
                          c1:= 1;
                          remainder of cycle 1; goto A1
                 end;
      process 2: begin A2: c2:= 0;
                      L2: if c1 = 0 then goto L2;
                          critical section 2;
                          c2:= 1;
                          remainder of cycle 2; goto A2
                 end
```

```
      parend
end
```

It is worthwhile to verify that this solution is at least completely safe.

Let us focus our attention on the moment that process 1 finds c2 = 1 and therefore decides to enter its critical section. At this moment we can conclude:

(1) that the relation `c1 = 0` already holds and will continue to hold until process 1 has completed the execution of its critical section;

(2) that, since `c2 = 1` holds, process 2 is well outside its critical section, which it cannot enter while `c1 = 0` holds, i.e. while process 1 is still engaged in its critical section.

Thus the mutual exclusion is indeed guaranteed.

But this solution, alas, must also be rejected: in its safety measures it has been too drastic, for it contains the danger of definite mutual blocking. When after the assignment `c1:= 0` but yet before the inspection of c2 (both by process 1) process 2 performs the assignment `c2:= 0`, then both processes have arrived at label `L1` or `L2` respectively and both relations `c1 = 0` and `c2 = 0` hold, with the result that both processes will wait for each other to eternity. Therefore this solution, too, must be rejected.

It was all right to set one's own `c` before inspecting the `c` of the other, but it was wrong to stick to one's own c-setting and just to wait. This is (somewhat) remedied in the following construction:

```
begin integer c1, c2;
      c1:= 1; c2: = 1;
      parbegin
      process 1: begin L1: c1:= 0;
                           if c2 = 0 then
                                 begin cl:= 1; goto L1 end;
                           critical section 1;
                           c1:= 1;
                           remainder of cycle 1; goto L1
                 end;
      process 2: begin L2: c2:= 0;
                           if c1 = 0 then
                                 begin c2:= 1; goto L2 end;
                           critical section 2;
                           c2:= 1;
                           remainder of cycle 2; goto L2
                 end
      parend
end
```

This construction is as safe as the previous one, and when the assignments `c1:= 0` and `c2:= 0` are performed "simultaneously" it will not necessarily lead to mutual blocking ad infinitum, because both processes will reset their own `c` back to 1 before restarting the entry rites, thereby enabling the other process to catch the opportunity. But our principles force us to reject this solution also, for the refusal to make any assumptions about the speed ratio implies that we have to cater for all speeds, and the last solution admits the speeds to be so carefully adjusted that the processes inspect the other's `c` only in those periods of time that its value is = 0. To make clear that we reject such solutions that only work with some luck, we state our next requirement: "If the two processes are about to enter their critical sections, it must be impossible to devise for them such finite speeds, that the decision which one of the two is the first to enter its critical section is postponed to eternity."

In passing we note that the solution just rejected is quite acceptable in everyday life, e.g. when two people are talking over the telephone and they are suddenly disconnected, as a rule both try to re-establish the connection. They both dial and if they get the signal "Number Engaged" they put down the receiver and, if not already called, they try "some" seconds later. Of course, this may coincide with the next effort of the other party, but as a rule the connection is re-established successfully after very few trials. In our mechanical circumstances, however, we cannot accept this pattern of behaviour: our parties might very well be identical!

Quite a collection of trial solutions have been shown to be incorrect, and at some moment people that had played with the problem started to doubt whether it could be solved at all. To the Dutch mathematician Th. J. Dekker the credit is due for the first correct solution. It is, in fact, mixture of our previous efforts: it uses the "safe sluice" of our last constructions, together with the integer `turn` of the first one, but only to resolve the indeterminacy when neither of the two immediately succeeds. The initial value of `turn` could have been 2 as well.

```
begin integer c1, c2, turn;
      c1:= 1; c2:= 1; turn:= 1;
      parbegin
      process 1: begin A1: c1:= 0;
                       L1: if c2 = 0 then
                              begin if turn = 1 then goto L1;
                                    c1:= 1;
                                 B1: if turn = 2 then goto B1;
                                    goto A1
                              end;
                           critical section 1;
                           turn:= 2; c1:= 1;
                           remainder of cycle 1; goto A1
                 end;
      process 2: begin A2: c2:= 0;
                       L2: if c1 = 0 then
                              begin if turn = 2 then goto L2;
                                    c2:= 1;
                                 B2: if turn = 1 then goto B2;
                                    goto A2
                              end;
                           critical section 2;
                           turn:= 1; c2:= 1;
                           remainder of cycle 2; goto A2
                 end
      parend
end
```

We shall now prove the correctness of this solution. Our first observation is that each process only operates on its own `c`. As a result, process 1 inspects `c2` only while `c1 = 0`, it will only enter its critical section provided it finds `c2 = 1`; for process 2 the analogous observation can be made.

In short, we recognize the safe sluice of our last constructions, and the solution is therefore safe in the sense that the two processes can never be in their critical sections simultaneously. The second part of the proof has to show that in case of doubt the decision which of the two will be the first to enter cannot be postponed until eternity. Now we should pay some attention to the integer `turn`: we note that assignment to this variable occurs only at the end or, if you wish, as part of critical sections, and therefore we can regard the variable `turn` as a constant during the decision process. Suppose that `turn = 1`. Then process 1 can only cycle via `L1`, that is with `c1 = 0` and only as long as it finds `c2 = 0`. But if `turn = 1`, then process 2 can only cycle via `B2`, but this state implies `c2 = 1`, so that process 1 cannot cycle and is bound to enter its critical section. For `turn` 2 the mirrored reasoning applies. As third and final part of the proof we observe that stopping, say,

process 1 in "remainder of cycle 1" will not restrict process 2: the relation `c1 = 1` will then hold, and process 2 can merrily enter its critical section, quite independently of the current value of `turn`. And this completes the proof of the correctness of Dekker's solution. Those readers that fail to appreciate its ingenuity are kindly asked to realize that for them I have prepared the ground by means of a carefully selected set of rejected constructions.

2.2 The Generalized Mutual Exclusion Problem

The problem of Section 2.1 has a natural generalization: given N cyclic processes, each with a critical section, can we construct them in such a way that at any moment at most one of them is engaged in its critical section? We assume the same means of intercommunication to be available, i.e. a set of commonly accessible variables. Furthermore, our solution has to satisfy the same requirements, viz. that stopping one process well outside its critical section may in no way restrict the freedom of the others, and that if more than one process is about to enter its critical section it must be impossible to devise for them such finite speeds that the decision which one of them is to be first to enter its critical section can be postponed to eternity.

In order to be able to describe the solution in ALGOL 60, we need the concept of the array. In Section 2.1 we had to introduce a `c` for each of the two processes and we did so by declaring

```
integer c1, c2
```

Instead of enumerating the quantities, we can declare—under the assumption that `N` has a well-defined positive value—

```
integer array c[1 : N]
```

which means, that at one stroke we have introduced N integers, accessible under the names

```
c[subscript]
```

where `subscript` might take the values 1, 2, ... N.

The next ALGOL 60 feature we introduce is the so-called "for clause", which we shall use in the following form:

```
for j:= 1 step 1 until N do statement S
```

and which enables us to express repetition of `statement S` quite conveniently. In principle, the for clause implies that `statement S` will be executed N times, with `j` in succession = 1, = 2,... = N. (We have added "in

principle", for via a goto statement as constituent part of statement S and leading out of it, the repetition can be ended earlier.)

Finally, we need the logical operator that in this monograph is denoted by **and**. We have met the conditional clause in the form:

```
if condition then statement
```

We shall now meet:

```
if condition 1 and condition 2 then statement
```

meaning that statement S will be executed only if condition 1 and condition 2 are both satisfied. (Once more we should like to stress that this monograph is not an ALGOL 60 programming manual: the above—loose!—explanations of parts of ALGOL 60 have been introduced only to make this monograph as self-contained as possible.)

With the notational aids just sketched we can describe our solution for fixed N as follows.

The overall structure is:

```
begin integer array b, c[0 : N];
      integer turn;
      for turn:= 0 step 1 until N do
            begin b[turn]:= 1; c[turn]:= 1 end;
      turn:= 0;
      parbegin
      process 1: begin ... end;
      process 2: begin ... end;
          .
          .
          .
      process N: begin ... end;
      parend
end
```

The first declaration introduces two arrays with N + 1 elements each, the next declaration introduces a single integer turn. In the following for clause this variable turn is used to take on the successive values 1, 2, 3,... N, so that the two arrays are initialized with all elements 1. Then turn is set = 0 (i.e. none of the processes, numbered from 1 onwards, is privileged). After this the N processes are started simultaneously.

The N processes are all similar. The structure of the ith process is as follows ($1 \leq i \leq N$):

```
process i: begin integer j;
             Ai: b[i]:= 0;
             Li: if turn <> i then
                 begin c[i]:= 1;
                       if b[turn] = 1 then turn:= i;
                       goto Li
                 end;
                 c[i]:= 0;
                 for j:= 1 step 1 until N do
                    begin if j <> i and c[j] = 0 then goto Li
                    end;
                 critical section i;
                 turn:= 0; c[i]:= 1; b[i]:= 1;
                 remainder of cycle i; goto Ai
           end
```

Remark. The description of the N individual processes starts with a declaration `integer j`. According to the rules of ALGOL 60 this means that each process introduces its own, private, integer `j` (a so-called "local quantity").

We leave the proof to the reader. It has to show again:

(1) that at any moment at most one of the processes is engaged in its critical section;

(2) that the decision which of the processes is the first to enter its critical section cannot be postponed to eternity;

(3) that stopping a process in its "remainder of cycle" has no effect upon the others.

Of these parts, the second one is the more difficult one. (*Hint*: As soon as one of the processes has performed the assignment `turn:= i`, no new processes can decide to assign their number to turn before a critical section has been completed. Mind that two processes can decide "simultaneously" to assign their `i`-value to `turn`!)

(Remark that can be skipped at first reading)

The program just described inspects the value of `b[turn]` where both the array `b` and the integer `turn` are in common store. We have stated that inspecting a single variable is an indivisible action and inspecting `b[turn]` can therefore only mean: inspect the value of `turn`, and if this happens to be = 5, well, then inspect `b[5]`. Or, in more explicit ALGOL:

```
process i: begin integer j, k;
                    .
                    .
                    .
                k:= turn; if b[k] = 1 then ...
```

implying that by the time that `b[k]` is inspected, `turn` may already have a value different from the current one of `k`.

Without the stated limitations in communicating with the common store, a possible interpretation of "the value of `b[turn]`" would have been "the value of the element of the array `b` as indicated by the current value of `turn`". In so-called uniprogramming i.e. a single sequential process operating on quantities local to it the two interpretations are equivalent. In multiprogramming, where other active processes may access and change the same common information, the two interpretations make a great difference! In particular, for the reader with extensive experience in uniprogramming this remark has been inserted as an indication of the subtleties of the games we are playing.

2.3 A Linguistic Interlude

In Section 2.2 we described the co-operation of N processes; in the overall structure we used a vertical sequence of dots between the brackets `parbegin` and `parend`. This is nothing but a loose formalism, suggesting to the human reader how to compose in our notation a set of N co-operating sequential processes, under the condition that the value of N has been fixed beforehand. It is a suggestion for the construction of 3, 4, or 5071 co-operating processes, it does not give a formal description of N such co-operating processes in which N occurs as a parameter, i.e. it is not a description valid for any value of N.

It is the purpose of this section to show that the concept of the so-called "recursive procedure" of ALGOL 60 caters for this. This concept will be sketched briefly.

We have seen how after `begin` declarations could occur in order to introduce and to name either single variables (by enumeration of their names) or whole ordered sets of variables (viz. in the array declaration). With the so-called "procedure declaration" we can define and name a certain action; such an action may then be invoked by using its name as a statement, thereby supplying the parameters to which the action should be applied.

As an illustration we consider the following ALGOL 60 program:

```
begin integer a, b;
      procedure square(u, v); integer u, v;
          begin u:= v * v end;
   L: square(a, 3); square(b, a); square(a, b)
end
```

In the first line the integers named a and b are declared. The next line declares the procedure named square" operating on two parameters, which are specified to be single integers (and not, say, complete arrays). This line is called "the procedure heading". The immediately following statement—the so-called "procedure body"—describes by definition the action named: in the third line—in which the bracket pair begin ... end is superfluous—it is told that the action of square is to assign to the first parameter the square of the value of the second one. Then, labelled L, comes the first statement. Before its execution the values of both a and b are undefined, after its execution a = 9. After the execution of the next statement the value of b is therefore = 81, after the execution of the last statement the value of a is = 6561, the value of b is still = 81.

In the previous example the procedure mechanism was essentially introduced as a means for abbreviation, a means for avoiding to have to write down the "body" three times, although we could have done so quite easily:

```
begin integer a, b;
   L: a:= 3 * 3; b:= a * a; a:= b * b
end
```

When the body is much more complicated than in this example a program along the latter lines tends to be much lengthier indeed.

This technique of "substituting for the call the appropriate version of the body" is, however, no longer possible as soon as the procedure is a so-called recursive one, i.e. may call itself. It is then that the procedure really extends the expressive power of the programming language.

A simple example might illustrate the recursive procedure. The greatest common divisor of two given natural numbers is:

(1) if they have the same value equal to this value;

(2) if they have different values equal to the greatest common divisor of the smaller of the two and their difference.

In other words, if the greatest common divisor is not trivial (first case) the problem is replaced by finding the greatest common divisor of two numbers with a smaller maximum value.

(In the following program the insertion `value v, w;` can be skipped by the reader as being irrelevant for our present purposes; it indicates that for the parameters listed the body is only interested in the numerical value of the actual parameter, as supplied by the call.)

```
begin integer a;
      procedure GCD(u, v, w); value v, w; integer u, v, w;
          if v = w then u:= v
                   else
               begin if v < w then GCD(u, v, w - v)
                              else GCD(u, v - w, w)
               end;
      GCD(a, 12, 33)
end
```

(In this example the more elaborate form of the conditional statement is used, viz.:

```
if condition then statement 1 else statement 2,
```

meaning that if `condition` is satisfied, `statement 1` will be executed and `statement 2` will be skipped, and that if `condition` is not satisfied `statement 1` will be skipped and `statement 2` will be executed.)

The reader is invited to follow the pattern of calls of GCD and to see how the variable `a` becomes = 3; he is also invited to convince himself of the fact that the (dynamic) pattern of calls depends on the parameters supplied and that the substitution technique—replace call by body—as applied in the previous example would lead to difficulties here.

We shall now write a program to perform a matrix * vector multiplication in which:

(1) the order in which the M scalar * scalar products are to be calculated is indeed prescribed (the rows of the matrix will be scanned from left to right);

(2) the N rows of the matrix can be processed in parallel.

(Where we do not wish to impose the restriction of purely integer values, we have used the declarator `real` instead of the declarator `integer`; furthermore, we have introduced an array with two subscripts in what we hope is an obvious manner.)

It is assumed that, upon entry of this block of program, the integers `M` and `N` have positive values.

```
begin real array matrix[1 : N, 1 : M];
      real array vector[1 : M];
      real array product[1 : N];
      procedure rowmult(k); value k; integer k;
          begin if k > 0 then
                parbegin
                   begin real s; integer j;
                         s:= 0;
                         for j:= 1 step 1 until M do
                            s:= s + matrix[k, j] * vector[j];
                         product[k]:= s
                   end;
                   rowmult(k - 1)
                parend
           end
      .
      .
      .
      rowmult(N);
      .
      .
      .
end
```

3 THE MUTUAL EXCLUSION PROBLEM REVISITED

We return to the problem of mutual exclusion in time of critical sections, as introduced in Section 2.1 and generalized in Section 2.2. This section deals with a more efficient technique for solving this problem; only after having done so we have adequate means for the description of examples, with which I hope to convince the reader of the rather fundamental importance of the mutual exclusion problem, in other words, I must appeal to the patience of the wondering reader (suffering, as I am, from the sequential nature of human communication!).

3.1 The Need for a More Realistic Solution

The solution given in Section 2.2 is interesting in as far as it shows that the restricted means of communication provided are, from a theoretical point of view, sufficient to solve the problem. From other points of view, which are just as dear to my heart, it is hopelessly inadequate.

To start with, it gives rise to a rather cumbersome description of the individual processes, in which it is anything but transparent that the overall behaviour is in accordance with the (conceptually so simple) requirement of the mutual exclusion. In other words, in some way or another this solution

is a tremendous mystification. Let us try to isolate in which respect this solution represents indeed a mystification, for this investigation could give the clue to improvement.

Let us consider the period of time during which one of the processes is in its critical section. We all know, that during that period no other processes can enter their critical section and that, if they want to do so, they have to wait until the current critical section execution has been completed. For the remainder of that period hardly any activity is required from them: they have to wait anyhow, and as far as we are concerned "they could go to sleep".

Our solution does not reflect this at all: we keep the processes busy setting and inspecting common variables all the time, as if no price has to be paid for this activity. But if our implementation—i.e. the ways in which or the means by which these processes are carried out—is such that "sleeping" is a less-expensive activity than this busy way of waiting, then we are fully justified (now also from an economic point of view) to call our solution misleading.

In present-day computers there are at least two ways in which this active way of waiting can be very expensive. Let me sketch them briefly. These computers have two distinct parts, usually called "the processor" and "the store". The processor is the active part, in which the arithmetic and logical operations are performed, it is "active and small"; in the store, which is "passive and large", there resides at any moment the information which is not being processed at that very moment but only kept there for future reference. In the total computational process information is transported from store to processor as soon as it has to play an active role, the information in store can be changed by transportation in the inverse direction.

Such a computer is a very flexible tool for the implementation of sequential processes. Even a computer with only one single processor can be used to implement a number of concurrent sequential processes. From a macroscopic point of view it will seem as though all these processes are being carried out simultaneously, a closer inspection will reveal, however, that at any "microscopic" moment the processor serves only one single program at a time, and the overall picture only results because at well-chosen moments the processor will switch from one process to another. In such an implementation the different processes share the same processor, and activity (i.e. a non-zero speed) of any single process will imply zero speed for the others; it is then undesirable that precious processor time is consumed by processes which cannot go on anyhow.

Apart from processor sharing, the store sharing could make the unnecessary activity of a waiting process undesirable. Let us assume that inspection of or assignment to a "common variable" implies the access to an information unit a so-called "word" in a ferrite-core store. Access to a word in a core store takes a non-zero time, and for technical reasons only one word can be accessed at a time. When more than one active process may wish access to words of the same core store the usual arrangement is that in the case of imminent coincidence the storage access requests from the different active processes are granted according to a built-in priority rule: the lower priority process is automatically held up. (The literature refers to this situation when it describes "a communication channel stealing a memory cycle from the processor".) The result is that frequent inspection of common variables may slow down any processes which share the same core storage for their local quantities.

3.2 The Synchronizing Primitices

The origin of the complications, which lead to such intricate solutions as the one described in Section 2.2, is the fact that the indivisible accesses to common variables are always "one-way information traffic": an individual process can either assign a new value or inspect a current value. Such an inspection itself, however, leaves no trace for the other processes, and the consequence is that, when a process wants to react to the current value of a common variable, that variable's value may have been changed by the other processes between the moment of its inspection and the following effectuation of the reaction to it. In other words: the previous set of communication facilities must be regarded as inadequate for the problem at hand, and we should look for more appropriate alternatives.

Such an alternative is provided by introducing:

(a) among the common variables special-purpose integers, which we shall call "semaphores";

(b) among the repertoire of actions, from which the individual processes have to be constructed, two new primitives, which we call the "P-operation" and the "V-operation" respectively.

The latter operations always operate on a semaphore and represent the only way in which the concurrent processes may access the semaphores.

The semaphores are essentially non-negative integers; when used only to solve the mutual exclusion problem the range of their values will even be restricted to 0 and 1. It is the merit of the Dutch physicist and computer designer C. S. Scholten to have shown a considerable field of applicability for semaphores that can also take on larger values. When there is a need for distinction we shall talk about "binary semaphores" and "general semaphores" respectively. The definition of the P- and V-operation that I shall give now holds regardless of this distinction.

Definition. The V-operation is an operation with one argument, which must be the identification of a semaphore. (If `S1` and `S2` denote semaphores we can write `V(S1)` and `V(S2)`.) Its function is to increase the value of its argument semaphore by 1; this increase is to be regarded as an indivisible operation.

Note that this last sentence makes `V(S1)` inequivalent to `S1:= S1 + 1`. For suppose that two processes A and B both contain the statement `V(S1)` and that both should like to perform this statement at a moment when, say, `S1 = 6`. Excluding interference with `S1` from other processes, A and B will perform their V-operations in an unspecified order—at least: outside our control—and after the completion of the second V-operation the final value of `S1` will be = 8. If `S1` had not been a semaphore but just an ordinary common integer, and if processes A and B had contained the statement `S1:= S1 +` instead of the V-operation on `S1`, then the following could happen. Process A evaluates `S1 + 1` and computes 7; before effecting, however, the assignment of this new value, process B has reached the same stage and also evaluates `S1 + 1`, computing 7. Thereafter both processes assign the value 7 to `S1`, and one of the desired incrementations has been lost. The requirement of the "indivisible operation" is meant to exclude this occurrence when the V-operation is used.

Definition. The P-operation is an operation with one argument, which must be the identification of a semaphore. (If `S1` and `S2` denote semaphores we can write `P(S1)` and `P(S2)`.) Its function is to decrease the value of its argument semaphore by 1 as soon as the resulting value would be non-negative. The completion of the P-operation—i.e. the decision that this is the appropriate moment to effectuate the decrease and the subsequent decrease itself—is to be regarded as an indivisible operation.

It is the P-operation which represents the potential delay, viz. when a process initiates a P-operation on a semaphore, that at that moment is = 0, in that case this P-operation cannot be completed until another process has

performed a V-operation on the same semaphore and has given it the value 1. At that moment more than one process may have initiated a P-operation on that very same semaphore. The clause that completion of P-operation is an indivisible action means that when the semaphore has got the value 1 only one of the initiated P-operations on it is allowed to be completed. Which one, again, is left unspecified, i.e. at least outside our control.

At this stage we shall take the implementability of the P- and V-operations for granted.

3.3 The Synchronizing Primitives Applied to the Mutual Exclusion Problem

The construction of the N processes, each with a critical section, the executions of which must exclude one another in time (see Section 2.2) is now trivial. It can be done with the aid of a single binary semaphore, say `free`. The value of `free` equals the number of processes allowed to enter their critical section now, or;

`free = 1` means: none of the processes is engaged in its critical section
`free = 0` means: one of the processes is engaged in its critical section.

The overall structure of the solution becomes:

```
begin integer free; free:= 1;
  parbegin
  process 1: begin ... end;
  process 2: begin ... end;
     .
     .
     .
  process N: begin ... end;
  parend
end
```

with the ith process of the form:

```
process i: begin
             Li: P(free); critical section i; V(free);
                 remainder of cycle i; goto Li
           end
```

4 THE GENERAL SEMAPHORE

4.1 Typical Uses of the General Semaphore

We consider two processes, which are called the "producer" and the "consumer" respectively. The producer is a cyclic process, and each time it goes

through its cycle it produces a certain portion of information that has to be processed by the consumer. The consumer is also a cyclic process, and each time it goes through its cycle it can process the next portion of information, as produced by the producer. A simple example is given by a computing process, producing as "portions of information" punched-card images to be punched out by a card punch, which plays the role of the consumer.

The producer-consumer relation implies a one-way communication channel between the two processes, along which the portions of information can be transmitted. We assume the two processes to be connected for this purpose via a buffer with unbounded capacity, i.e. the portions produced need not be consumed immediately, but they may queue in the buffer. The fact that no upper bound has been given for the capacity of the buffer makes this example slightly unrealistic, but this should not trouble us too much now.

(The reason for the name "buffer" becomes understandable when we investigate the consequences of its absence, viz. when the producer can only offer its next portion after the previous portion has been actually consumed. In the computer-card punch example, we may assume that the card punch can punch cards at a constant speed, say 4 cards per second. Let us assume that this output speed is well matched with the production speed, i.e. that the computer can perform the card image production process with the same average speed. If the connection between computing process and card punch is unbuffered, then the couple will only work continuously at full speed when the card-production process produces a card every quarter of a second. If, however, the nature of the computing process is such that after one or two seconds vigorous computing it produces 4 to 8 card images in a single burst, then unbuffered connection will result in a period of time during which the punch will be idle (for lack of information), followed by a period in which the computing process has to be idle, because it cannot get rid of the next card image before the preceding one has been actually punched. Such irregularities in production speed, however, can be smoothed out by a buffer of sufficient size and that is why such a queuing device is called "a buffer".)

In this section we shall not deal with the various techniques of implementing a buffer. It must be able to contain successive portions of information, it must therefore be a suitable storage medium, accessible to both processes. Furthermore, it must not only contain the portions themselves, it must also represent their linear ordering. (In the literature two well-known techniques are known as "cyclic buffering" and "chaining" respectively.) When the producer has prepared its next portion to be added to the buffer we shall denote

this action simply by `add portion to buffer`, without going into further details; similarly, the `take portion from buffer` describes the consumer's behaviour, where the oldest portion still in the buffer is understood. (Another name of a buffer is a "First-In-First-Out-Memory".)

Omitting in the outermost block all declarations for the buffer, we can now construct the two processes with the aid of a single general semaphore, called `number of queuing portions`.

```
begin integer number of queuing portions;
      number of queuing portions:= 0;
      parbegin
      producer: begin
             again 1: produce the next portion;
                      add portion to buffer;
                      V(number of queuing portions);
                      goto again 1
                end;
      consumer: begin
             again 2: P(number of queuing portions);
                      take portion from buffer;
                      process portion taken;
                      goto again 2
                end
      parend
end
```

The first line of the producer represents the coding of the process which forms the next portion of information; it has a meaning quite independent of the buffer for which this portion is intended; when it has been executed the next portion has been successfully completed, the completion of its construction can no longer be dependent on other (unmentioned) conditions. The second line of coding represents the actions which define the finished portion as the next one in the buffer; after its execution the new portion has been added completely to the buffer, apart from the fact that the consumer does not know it yet. The V-operation finally confirms its presence, i.e. signals it to the consumer. Note that it is absolutely essential that the V-operation is preceded by the complete addition of the portion. About the structure of the consumer analogous remarks can be made.

Particularly in the case of buffer implementation by means of chaining the operations `add portion to buffer` and `take portion from buffer`—operating as they are on the same clerical status information of the buffer—may interfere with each other in a most undesirable fashion, unless we see to it, that they exclude each other in time. This can be catered for by a binary semaphore, called `buffer manipulation`, the values of which mean:

= 0: either adding to or taking from the buffer is taking place
= 1: neither adding to nor taking from the buffer is taking place.

The program is as follows:

```
begin integer number of queuing portions,
              buffer manipulation;
      number of queuing portions:= 0;
      buffer manipulation:= 1;
      parbegin
      producer: begin
             again 1: produce next portion;
                      P(buffer manipulation);
                      add portion to buffer;
                      V(buffer manipulation);
                      V(number of queuing portions);
                      goto again 1
                end;
      consumer: begin
             again 2: P(number of queuing portions);
                      P(buffer manipulation);
                      take portion from buffer;
                      V(buffer manipulation);
                      process portion taken;
                      goto again 2
                end
      parend
end
```

The reader is requested to convince himself that:

(a) the order of the two V-operations in the producer is immaterial;

(b) the order of the two P-operations in the consumer is essential.

Remark. The presence of the binary semaphore `buffer manipulation` has another consequence. We have given the program for one producer and one consumer, but now the extension to more producers and/or more consumers is straightforward: the same semaphore sees to it that two or more additions of new portions will never get mixed up, and the same applies to two or more takings of a portion by different consumers. The reader is requested to verify that the order of the two V-operations in the producer is still immaterial.

4.2 The Superfluity of the General Semaphore

In this section we shall show the superfluity of the general semaphore and we shall do so by rewriting the last program of the previous section, using binary

semaphores only. (Intentionally I have written "we shall show" and not "we shall prove". We do not have at our disposal the mathematical apparatus that would be needed to give such a proof, and I do not feel inclined to develop such mathematical apparatus now. Nevertheless, I hope that my show will be convincing!) We shall first give a solution and postpone the discussion till afterwards.

```
begin integer numqueupor, buffer manipulation,
              consumer delay;
      numqueupor:= 0; buffer manipulation:= 1;
      consumer delay:= 0;
      parbegin
      producer: begin
             again 1: produce next portion;
                      P(buffer manipulation);
                      add portion to buffer;
                      numqueupor:= numqueupor + 1;
                      if numqueupor = 1 then
                            V(consumer delay);
                      V(buffer manipulation);
                      goto again 1
                end;
      consumer: begin integer oldnumqueupor;
                wait: P(consumer delay);
               go on: P(buffer manipulation);
                      take portion from buffer;
                      numqueupor:= numqueupor - 1;
                      oldnumqueupor:= numqueupor;
                      V(buffer manipulation);
                      process portion taken;
                      if oldnumqueupor = 0 then goto wait
                                           else goto go on
                end
      parend
end
```

Relevant in the dynamic behaviour of this program are the periods of time during which the buffer is empty. (As long as the buffer is not empty, the consumer can go on happily at its maximum speed.) Such a period can only be initiated by the consumer (by taking the last portion present from the buffer), it can only be terminated by the producer (by adding a portion to an empty buffer). These two events can be detected unambiguously, thanks to the binary semaphore `buffer manipulation`, that guarantees the mutual exclusion necessary for this detection. Each such period is accompanied by a P- and a V-operation on the new binary semaphore `consumer delay`. Finally, we draw attention to the local variable `oldnumqueupor` of the consumer: its value is set during the taking of the portion and fixes whether

it was the last portion then present. (The more expert ALGOL readers will be aware that we only need to store a single bit of information, viz. whether the decrease of `numqueupor` resulted in a value = 0; we could have used a local variable of type Boolean for this purpose.) When the consumer decides to go to `wait`, i.e. finds `oldnumqueupor = 0`, at that moment `numqueupor` itself could already be greater than zero again!

In the previous program the relevant occurrence was the period with empty buffer. One can remark that emptiness is, in itself, rather irrelevant: it only matters, when the consumer should like to take a next portion, which is still absent. We shall program this version as well. In its dynamic behaviour we may expect less P- and V-operations on `consumer delay`: they will not occur when the buffer has been empty for a short while, but is filled again in time to make delay of the consumer unnecessary. Again we shall first give the program and then its discussion.

```
begin integer numqueupor, buffer manipulation,
      consumer delay;
      numqueupor:= 0; buffer manipulation:= 1;
      consumer delay:= 0;
      parbegin
      producer: begin
             again 1: produce next portion;
                      P(buffer manipulation);
                      add portion to buffer;
                      numqueupor:= numqueupor + 1;
                      if numqueupor = 0 then
                         begin V(buffer manipulation);
                               V(consumer delay) end
                                     else
                         V(buffer manipulation);
                      goto again 1
                end;
      consumer: begin
             again 2: P(buffer manipulation);
                      numqueupor:= numqueupor - 1;
                      if numqueupor = -1 then
                         begin V(buffer manipulation);
                               P(consumer delay);
                               P(buffer manipulation) end;
                      take portion from buffer;
                      V(buffer manipulation),
                      process portion taken;
                      goto again 2
                end
      parend
end
```

Again, the semaphore `buffer manipulation` caters for the mutual exclu- sion of critical sections. The last six lines of the producer could have been formulated as follows:

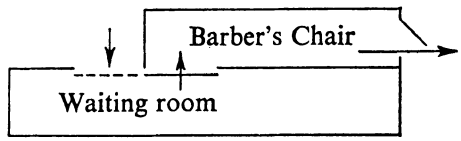
```
if numqueupor = 0 then V(consumer delay);
V(buffer manipulation); goto again 1
```

In not doing so I have followed a personal taste, viz. to avoid P- and V- operations within critical sections; a personal taste to which the reader should not pay too much attention.

The range of possible values of `numqueupor` has been extended with the value `-1`, meaning (outside critical section execution) "the buffer is not only empty, but its emptiness has already been detected by the consumer, which has decided to wait". This fact can be detected by the producer when, after the addition of one, `numqueupor = 0` holds.

Note how, in the case of `numqueupor = -1`, the critical section of the consumer is dynamically broken into two parts: this is most essential, for otherwise the producer would never get the opportunity to add the portion that is already so much wanted by the consumer.

(The program just described is known as "The Sleeping Barber". There is a barbershop with a separate waiting room. The waiting room has an entry and next to it an exit to the room with the barber's chair, entry and exit sharing the same sliding door, which always closes one of them; furthermore, the entry is so small that only one customer can enter it at a time, thus fixing their order of entry. The mutual exclusions are thus guaranteed.



When the barber has finished a haircut he opens the door to the waiting room and inspects it. If the waiting room is not empty he invites the next customer, otherwise he goes to sleep in one of the chairs in the waiting room. The complementary behaviour of the customers is as follows: when they find zero or more customers in the waiting room they just wait their turn, when they find, however, the Sleeping Barber—`numqueupor = -1`—they wake him up.)

The two programs given present a strong indication that the general semaphore is, indeed, superfluous. Nevertheless, we shall not try to abolish

the general semaphore: the one-sided synchronization restriction expressible by it is very common, and comparison of the solutions with and without the general semaphore shows convincingly that it should be regarded as an adequate tool.

4.3 The Bounded Buffer

I shall give a last simple example to illustrate the use of the general semaphore. In Section 4.1 we have studied a producer and a consumer coupled via a buffer with unbounded capacity. This is a typically one-sided restriction: the producer can be arbitrarily far ahead of the consumer; on the other hand, the consumer can never be ahead of the producer. The relation becomes symmetric when the two are coupled via a buffer of finite size, say of N portions. We give the program without discussion; we ask the reader to convince himself of the complete symmetry. ("The consumer produces and the producer consumes empty positions in the buffer.") The value N, as well as the buffer, is supposed to be defined in the surrounding universe into which the following program should be embedded.

```
begin integer number of queuing portions,
              number of empty positions,
              buffer manipulation;
      number of queuing portions:= 0;
      number of empty positions:= N;
      buffer manipulation:= 1;
      parbegin
      producer: begin
             again 1: produce next portion;
                      P(number of empty positions);
                      P(buffer manipulation);
                      add portion to buffer;
                      V(buffer manipulation);
                      V(number of queuing portions);
                      goto again 1
                end;
      consumer: begin
             again 2: P(number of queuing portions);
                      P(buffer manipulation);
                      take portion from buffer;
                      V(buffer manipulation);
                      V(number of empty positions);
                      process portion taken;
                      goto again 2
                end
      parend
end
```

5 CO-OPERATION 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 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 co-operating sequential processes have to be constructed and the overall behaviour of these processes combined has to satisfy more elaborate requirements—the community, formed by them, has, as a whole, to be well behaved in some sense—we can only expect to be able to achieve this 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, such as the general semaphore, to do the job. In general, we shall need such 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". Two points should be stressed.

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 then take the decisions. If we cannot find a guiding principle (e.g. efficiency considerations), then we must have the courage to impose some rule for the sake 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 of doubting) 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 we can hardly expect any serious help from field tests. 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 be 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 will, however, 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 accommodate 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 would only occur when the buffer was full to the brim, now it can also happen because free space in the buffer, although present, is insucient for the portion concerned.

Furthermore, when we have more than one producer and one of them is waiting, then the other ones may go on 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 accommodate its portion, no other producers can therefore add their portions until further notice: they cannot do so if the new portion is larger (for then it will not fit either), they are not allowed to if the new portion is smaller, for then they have a lower priority and must leave the buffer for the earlier request.

Suppose a moment at which there is a completely filled buffer and three producers, waiting to offer portions of 1, 2, and 3 units respectively. When

a consumer now comsumes a five-unit portion the priority rule implies that the producers with the 2-unit portion and the 3-unit portion will get the opportunity to go on and not the one offering the 1-unit portion. It is *not* meant to imply that in that case 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 producer 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.)

Vacated buffer space must be signalled back to the producers, but the possible consequences of vacating 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, 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 could even be lower had we had 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 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 = 0 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. This part of the program hopefully presents no difficulties to the reader who 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 immediately or not. It can add immediately 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 immediately, 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]` > 0, `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 re-enters 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 waiting case 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 fulfil 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:

(a) 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 more than one producer. Furthermore, this decrease is in accordance with the producer behaviour.

(b) `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.

(c) 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 left (`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 co-operating 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 of the total construction, admittedly somewhat simplified yet hopefully sufficiently complicated to pose to us a real problem, yet sufficiently simple so as not to drown the basic pattern of our solution in a host of inessential details.

We have N identical processes (numbered from 1 through N), and essentially they can each 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 since 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, `A1(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 occurrence: it relieves the operator from the duty 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` with the rule that a next conversation can be initiated only when the previous one has been completed.

We shall now complicate the requirements in three respects.

First, 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 `A4(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), A3 - - - A4(i)`

(d) `Q1(i), A3 - - - 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)` will arrive, and we cannot delegate its interpretation 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 the operator too might come to this conclusion at the same time. As soon as the channel becomes available, we wish that the operator can use it and that, if he so desires, 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 the operator

(a) 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 machine's reaction, whether his intervention 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 even penetrate if the children are not listening (talking, for instance).

This priority rule may cause the communication channel to be reserved for an announced A4—or A5 message. By the time 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, on second thoughts, 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 messages are applicable; when a message has been rejected as non-applicable the interpreter should return to such a state that the operator can then 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 acceptable at all times, so our message interpreter will have to have 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 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.

The above five points are helpful, and in view of our previous experiences they seem a set of reasonable principles. One facet of this subject has been to 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 any such solution is found 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, interpretable meaning, when `mutex = 1` holds, i.e. when none of the processes is inside a critical section, in which these values 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 have been constructed, but we can only construct the programs after 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, then starts to enumerate the different states and finally tries to build the programs. Then two 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 did not bother to minimize the number of states introduced.

In my experience it is easier to conceive first the states (these 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 = 1; 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 variables 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 start with the element `procvar[i]`, describing the state of process `i`.

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

This we call "the home 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 reasonable 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 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 indefinitely 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*). Moreover, `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] = 4`

"`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 too much at once, 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 $\neq 0$, 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 occurrence 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 Q1-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 $\neq 0$. 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 ceases to occupy the communication facility decides upon its future usage. This occurs 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 message 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 purpose only. In ALGOL 60 such a comment is 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 records
                      itself as dormant 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 communication
                facility; 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
                    neither another 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 interpretation 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[il:= 4; V(procsem[i]);
                              goto perhaps comvar to zero end;
                        comment Otherwise process not waiting
                        for postponed 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 at 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 section we have first presented 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 live recording"— of the birth of this solution. When I started to write this section the problem posed was as new to me as it was to 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 to 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 satisfied 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) will no longer increase with the number of possible answers to the question Q1.)

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 the 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 ith 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(mutex) end
                else
begin procvar[n]:= 1; procsem[n]:= 0;
      V(mutex); 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".

The latter are called "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 semaphores `mutex` caters for the mutual exclusion. Besides these, we can distinguish regions in which relevant actions occur, such as:

In the ith Process

Region 1: sending an M-message
Region 2: sending a Q1(i)-question
Region 3: reacting to operanswer[i] (This region is somewhat openended).

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. These sequencing patterns 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 to be inspected at least within the critical section. In this example, however, we shall observe the rule.)

The remaining program improvements are less fundamental.

Coding will be smoothed if we represent the fact of requested operator priority not by additional values of `comvar` but by 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. they have 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`	home 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`	home 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 so 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[loop]:= 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 nth 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 pvcopy[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(process 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 in which 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

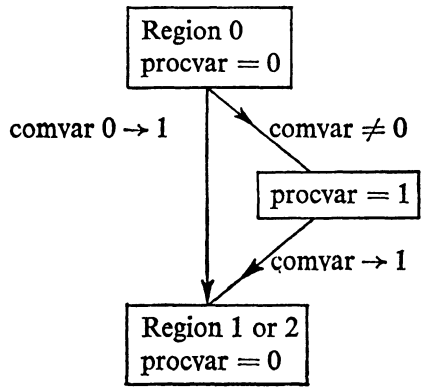
which the conversation has been initiated. (Otherwise we have to exclude simultaneous, contradictory messages `A4(i)` and `A5(i)` via the two different consoles. The solution without this restriction is left to the really fascinated reader.)

5.2.2 Proving the Correctness

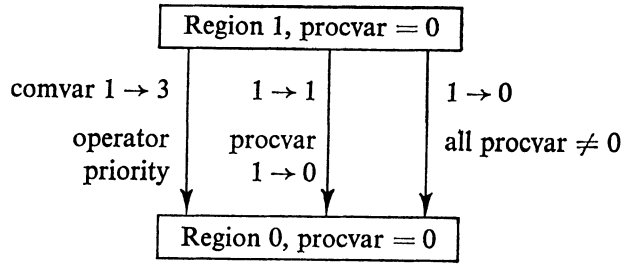
In this section title I have used the word "proving" in an informal way. I have not defined what formal conditions must be satisfied by a "legal proof", and I do not intend to do so. When I can find a way to discuss the program of Section 5.2.1, by which I can convince myself of—and hopefully anybody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

In the following "state picture" we make a diagram of all the states in which a process may find itself "for any considerable length of time", i.e. outside sections critical to mutex. The arrows 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.

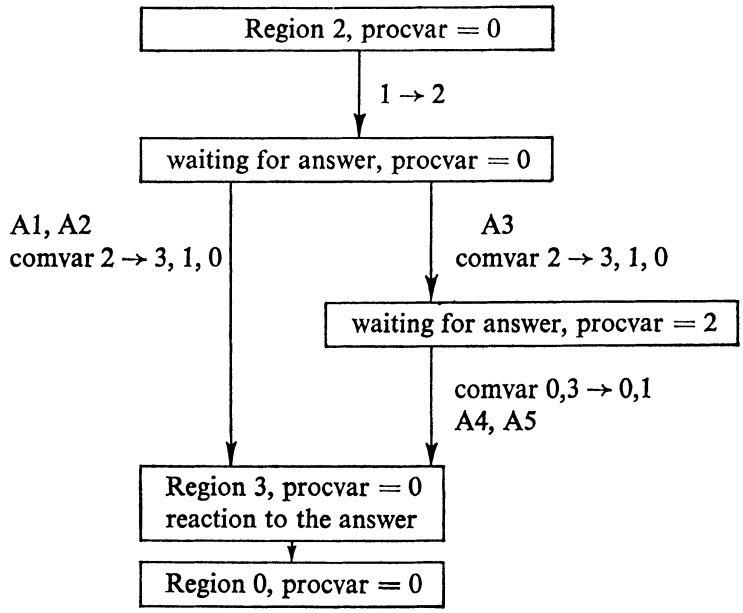
Calling the neutral region of a process before entry into a Region 1 or Region 2, `Region 0`, we can give the state picture



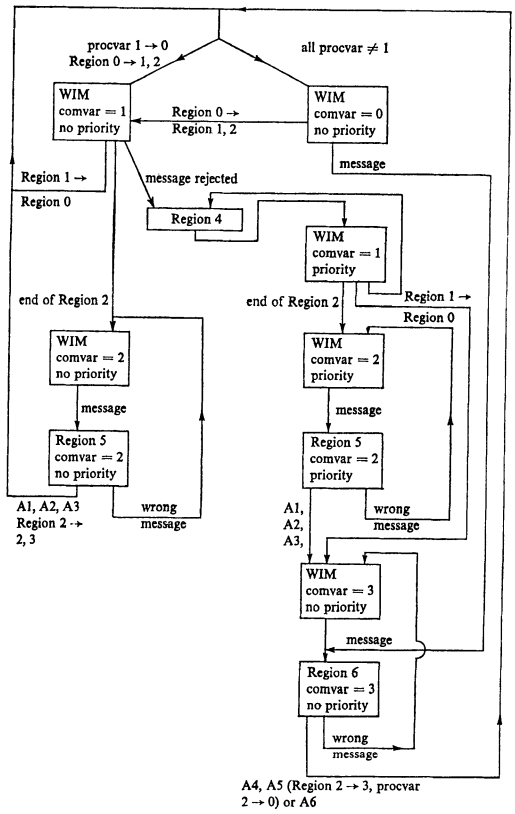
Leaving Region 1 can be pictured as:



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 abbreviation for "Waiting for Incoming Message".



These diagrams, of course, tell us nothing new, but they may be a powerful aid to program inspection.

We verify first that `comvar = 0` represents indeed the home position of the communication facility, i.e. its availability either for entrance into Region 1 or Region 2 (by one of the processes) or for 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 when it comes 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:= true.`

These assignments are performed only under the condition `comvar <> 0`; furthermore, the assignment `comvar:= 0`—only occurring in the procedure `select new comvar value`—is only performed provided "non-operator priority and all procvar $\neq$ 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`.

Since all ways of ceasing to occupy 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 if necessary;

(b) that such a delay is guaranteed to end at the earliest opportunity.

The structure of the message interpreter shows clearly that:

(a) it can execute Region 5 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 $\neq$ 2.

The only assignment `comvar:= 2` occurs at the end of Region 2. As a result, each Region 2 can be followed only 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 sets `procvar[i]:= 2` for this process, thus allowing the answer A4 or A5 for this process exactly *once* in Region 6. Again `V(procsem[i])` is performed only 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 analysis of 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 of correction.

By the above analysis we hope to have created sufficient confidence in the correctness of our construction. The analysis followed 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 co-operation between various processes when they have to share the same facilities. We have selected this problem for various reasons. First, it arises by a straightforward extension of the sound principle that no two persons should use a single compartment of a revolving door simultaneously. Secondly, its solution, which I regard as non-trivial and which will be given in Section 6.1, gives us a nice example of more subtle co-operation 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 facility-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 those processes of which is known in advance:

(1) that their demand on storage space will not exceed a certain limit, and

(2) that each computational process will end, provided that storage space requested by the process will be put at its disposal. 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. By "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 nor one out of a special group.

We now request that a process, once initiated, will—sooner or later—get the opportunity to complete its action and reject any organization in which it may happen that a process may have to be killed half-way through 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 processes request their next page, and if 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 continue only 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 unnecessarily 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 by florin.

4. A customer when asking for an increase in his current loan undertakes to accept without complaint 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 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 subject 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 enter into 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.

In order 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]` $\leq$ `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

$0 \leq$ `loan[i]` $\leq$ `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

$0 \leq$ `cash` $\leq$ `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 inspecting 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[il:= 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[i]`. As soon as `finish doubtful[i]:= false` can be executed the algorithm can decide directly on safety of the situation, for then clearly this attempted 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 also the florins are processes. (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 return a borrowed florin immediately. After the customer has indicated that he has no further use for this florin the florin may not be instantaneously available for subsequent 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 to cash: it will signal its return into the banker's cash to the customer from which it came via a customer semaphore florin returned. A P-operation on this semaphore should guard the customer against an inadvertent 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 to be granted now. (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 labelled 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 array finish doubtful[1 : 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;
              L0: for i:= 1 step 1 until N do
                  begin if finish doubtful[i]
                        and claim[i] <= free money then
                        begin if i <> j then
                              begin
                                finish doubtful[i]:= false;
```

```
                                free money:=
                                  free money + loan[i];
                                goto L0
                              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]);
       comment 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;
       P(mutex);
       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 take place only 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 co-operation it has given rise to a simplification that proved to be indispensable. The underlying cause of this increase in simplicity is 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 "multi-programming"—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 co-operate with each other, and our discussions on this co-operation apply equally well to "concurrent programming" as to "multi-programming" or any mixture of the two. What in concurrent programming is spread out in space (e.q. equipment) is in multi-programming 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 multi-processors, multi-programming, and the like—the complete symmetry between a normal sequential computer, on the one hand, and its peripheral 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 controlling them is by attaching meanings to the values of identified quantities. For instance, in the program section

```
    i:= 10;
L0: x:= sqrt(x); i:= i - 1;
    if i > 0 then goto L0
```

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". 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 multi-programming 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 any 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 effiiciency of the implementation will be considerable. If this chapter gives any reader a clearer indication of what kind of hierarchical ordering can be expected to be relevant I have reached one of my 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?