Contents Chapter One

CHAPTER ONE

ON MULTIPROCESSING

1.1. INTRODUCTION

1.1.1. Origination and development

The first multiprocessing systems were designed and built nearly twenty years ago, in the late fifties **(Enslow '76, pg. 228)**. The initial systems originated from experiments with concurrency and alternative computer architectures, which were meant to explore the possibilities for increased computer performance.
One motivation for the introduction of some amount of concurrency in computer systems was to increase *throughput*, by improving upon the speed of operation. This objective led to experiments with concurrency in the form of: pipe-lining systems, associative memories, autonomous I/O channels, array computers etc. **(Jurca '77, pg. 3 - 15)**.
Another motivation, which led more specifically to experiments with timesharing systems and multiple computer systems, was to increase the *efficiency* of the then still expensive computer equipment. 'Efficiency' was then interpreted as mere 'system efficiency', that is a 'high occupancy of system resources'. This high occupancy could be realized by sharing the computer equipment among a number of users with non-conflicting demands.
The sharing of computer equipment had the additional advantage to the user of a greater availability of the computer and lower average turn-around times (at the cost of some increase in 'virtual execution times', due to the additional delays)[1].

In *on-line* applications there was yet a third objective to improve upon the '*alertness*' of the processor controls. To realize this, one introduced still more subtle types of timesharing systems controlled by real-time interrupts.

Many different types of multi-computer configurations have been investigated. England summarized them as follows:

> "At one end of the spectrum there are multiple computers, two, three, four discrete computers standing side by side and having no communication with one another whatsoever. In the middle of the spectrum are multi-computer systems, in which several discrete computers are interfaced together by some kind of interconnection whereby they can communicate towards a common objective. At the other end of the spectrum are multi-processor systems where a multiprocessor is a single computer with multiple computing elements".
> **(England '76a, pg. 13)**.

As England indicates, multi-computer systems may vary greatly in the amount of

(1) A high 'system efficiency' in terms of the occupancy of equipment is however not equivalent to high 'user efficiency' in terms of system availability and average turn-around times. These two objectives may well conflict.

integration of their computing elements. In section 1.1.3. (**Terminology**) we will consider what characterizes a 'true' multi-programming, multi-processor, and multi-processing system. For the moment intuitive notions will suffice.

Concurrent (or 'parallel') programming has always been considered as a highly complicated extension of sequential programming. As we shall see in section 1.1.2. (**Coordination problems**), even trivial concurrent programs can contain quite subtle errors which may be very hard to trace. The programming tools of the beginning sixties were grossly inadequate to solve the new problems. Only gradually did it become possible to develop an entirely new conceptual model of computer programming which allowed one to understand sequentialism as just a special case of concurrency. In 1972 Cerf described this 'reversal' of the problem statement as follows:

> "The trick is to think in parallel processing terms".
> **(Cerf '72, pg. 99, see also pg. 88).**

It is important to note that this new view-point was only possible after the introduction and evolvement of new programming tools and concepts. Of great significance in this respect was the introduction of such concepts as *abstract processes*, *software layering*, *information hiding*, *data abstraction*.
We will first briefly outline the development of these concepts.

New concepts
Traditionally, the concepts of a program, the execution of a program and the concept of a machine (on which the program is to be executed) are closely related. This close relation may be satisfactory to make the working of sequential programs understandable; it is however inadequate for a clear understanding of parallelism. Already in 1963 Conway suggested distinguishing more sharply between the execution of a (parallel) program and the machine on which it is executed.

> "Fundamental to the concepts presented here is the principle, not yet commonly accepted, that parallel paths in a program need not bear fixed relationships to the processors of a multiprocessor system executing that program".
> **(Conway '63, pg. 146).**

The link between a processor and a (parallel path in a) program should, in Conway's view, not be made by the programmer but by the operator, or operating system.
Dijkstra extended and perfected these ideas a few years later, when he introduced the concept of the *software layer*. The following remarks, quoted from two of Dijkstra's articles in which he elucidated the design philosophies of the THE multiprogramming system, are illustrative:

> "... the software of our multi-programming system can be regarded as structured in layers. We conceive an ordered sequence of machines: A(0), A(1), ..., A(n), where A(0) is the given hardware machine (-). The software of layer i is defined in terms of machine A(i), it is to be executed by machine A(i), the software of layer i uses machine A(i) to make machine A(i+1). (-) The software of level 0 takes care of processor allocation ...".
> **(Dijkstra '69, pg. 115 - 116).**

> "... above level 0 the number of processors actually shared is no longer relevant. At higher levels we find the activity of the different sequential processes, the actual processor (-) having disappeared from the picture".
> **(Dijkstra '68b, pg. 343).**

Dijkstra stressed that the concerns for 'machine efficiency' should not influence the structure of programs on the higher levels of abstraction. Hardware concerns are to be treated in a low software layer at a low level of abstraction. (The terms 'high' and 'low' should, however, not be interpreted as measures for the relative importance of the software layers.)

Another important notion is the concept of an abstract process (also used by Dijkstra, see above). It is difficult to trace the first author who has used the term 'process' in a systematic way. **Horning & Randell '73 (pg. 5)** mentioned **Saltzer '66, Turski '68** and **Dijkstra '68b** in this context, but many others must have influenced and contributed to the development of the idea. The problem was to give a new interpretation of the 'execution of a program', this time not in terms of machines or machine cycles. In a symposium paper, presented in 1967, Lampson tentatively described the concept as follows:

> "... the essential characteristic of a process is that it has, at least conceptually, a processor of its own to run on ...".
> **(Lampson '68, pg. 347).**

Wirth described it more precisely:

> "A distinction is made between *processors* (workers) and *processes* (jobs). Many processors can operate on a single process during consecutive periods of time, and vice versa - a single processor can operate on many processes, switching its attention from one to another at certain moments. (-) We never insist that processes *must* be executed by individual processors in real concurrency. We imply that the m parallel processes $P_1, \ldots, P_m$, where each P_i consists of a sequence of instructions $P_i = P_{i1}, P_{i2}, \ldots, P_{in_i}$ *may* be executed in any sequence by any number of processors (even only one!) as long as P_{ij} precedes P_{ik} for all i and all $j < k$."
> **(Wirth '69, pg. 490).**

Thus the inherent link between the traditional concepts of a process and of a processor has been removed. More difficult still is it to remove the similar link between the traditional concepts of a process and a program.
Lampson wrote:

> "A process should be sharply distinguished from a program, which is a sequence of instructions in memory. We can speak of either a process or a processor executing a program. The process is the logical, the processor the physical environment for this execution."
> **(Lampson '68, pg. 348).**

It is important to note that a process does not simply formalize the 'execution of a program'. A process can be said to be *running* even when the corresponding program is currently not being executed.
With the concept of an abstract process, to be distinguished from both a processor and a program, one can describe concurrency problems in quite a natural way on the appropriate level of abstraction. Only with these concepts has it become possible to consider the *suspension* or *delay* of processes, to talk about the *creation*, *blocking*, *releasing*, and *destruction* (removal) of processes, in precisely these terms. One can use these abstractions as primitive standard operations in a computer system, thus hiding specific information about their implementation from users. This 'information hiding' has two advantages. Firstly, the users are relieved from the task of thinking in terms of highly detailed operations on specific hardware. Secondly, the system is protected against erroneous (user) implementations of standard operations.

There are two major causes for the changes in the conceptual model of computer

programming, which have been outlined here. The first reason was the rapidly increasing potential of the computer systems, among others by the introduction of multiprocessing. The second reason was the equally rapid decrease in the cost of hardware equipment. The first development created the need for new higher level tools with which one could master the increased complexity of the new systems. The second development allowed one to form a new opinion on the concern for 'system efficiency' and 'user efficiency'.
Kirkley wrote in 1977:

> "Back in the 1950-s (-) the ratio of hardware to software costs was about four to one; by 1973 that ratio has reversed."
> **(Kirkley '77, pg. 63).**

The immediate result of these developments was that the traditional concerns for 'hardware efficiency' could be moved to the later stages in system design, while systematic abstraction and simplicity of software structures became the prime concerns of programmers (see also chapter 3, **On structured programming**).

What made multiprocessing so difficult at first was that each problem, notably the interaction problems, had to be understood and solved in terms of hardware and primitive operations on hardware. Characteristic in this respect is the way in which the first new programming tools for multiprocessing systems were formulated. First of all the *philosophy* behind the early proposals is remarkable.

- One suggested *extending* the existing sequential programming languages with two or three *ad hoc* primitive operations, in order to make them suited for multiprocessing. One tried, in short, to understand concurrency as a mere extension of sequentialism.

Remarkable further is the *terminology*:

- The working of the primitive operations was explained in terms of 'machines', and 'the synchronization of machines'.

We have noted that both the philosophy and the terminology have changed. We no longer speak of 'the synchronization of machines', but of 'the coordination of processes'. 'Interaction problems' are formalized as 'process coordination problems' etc. The discussion on multiprocessing problems is thus moved to a higher level of abstraction. In the overview of coordination tools, given in section 1.3.3., these changes in philosophy and terminology will become apparent. It has been rightly noted that today the advantage of the multi-processing systems no longer lies in the possibilities for the sharing of (hardware) resources (that is 'increased system efficiency', see page 1), but in the possibilities for the sharing of *information* (which can be related to 'user efficiency', see page 1). **(Infotech Update, June 1977, Vol. 2, no. 6).**

In the next section we will consider the advantages and disadvantages of the multi-processing systems in more detail.

1.1.2. Advantages and disadvantages of multiprocessing

As advantages of multiprocessing systems we have mentioned: increased throughput, increased efficiency, and the possibility for the sharing of information. Systems with more than one processing unit can have the additional advantages of a higher *flexibility* in their capacity (by adding or deleting processing units one can increase or decrease the system's capacity over a wide range (see also chapter 5)), a higher *availability* (the failure of one out of N processing units need not stop the whole system), and at least a potentially higher *reliability* **(Taylor '72; Infotech '76).**

A further advantage is that multiprocessor systems allow one to build other types of control in systems; other than the conventional monolithic, fully centralized ones. It is now possible to experiment with distributed processing (e.g. linear control chains and hierarchical control trees (**Bredt '70**)), with load sharing strategies, etc.

There are, however, also disadvantages.
One does have the advantage of the facilities for intended process interactions, but the counterpart is clearly that *un*intended interactions are also possible. Unintended interactions can cause indeterminism, the mutilation of data, undesirable blockings, and even deadlock.
Schneck '74 noted that an unwarranted application of multiprogramming can even lead to a severe *degradation* of system efficiency. In short, Schneck argued that the introduction of multiprogramming may create problems on the usage of the system's working storage. The size of the directly accessible working storage will in most cases be far too small to answer the run-time requirements of all executable programs simultaneously. These storage problems create the need for inefficient overlay structures and an abundance of I/O operations.
Brinch Hansen confirmed this some years later, when he wrote:

> "Dynamic store algorithms that move programs and data segments around during execution can be a serious source of inefficiency that is not under the programmer's control."
> **(Brinch Hansen '77, pg. 10).**

These problems seem, however, to be more related to the required size of the working storage, or the storage allocation techniques, in multiprocessing systems, than to multiprocessing as such.

Dijkstra warned most eloquently against the desire to build always larger, bigger, and faster machines:

> "... the distorting spell of speed still seems to make its victims (-) ... one observes the honest expectation that with faster machines with lots of concurrent processing, the life-size problems will come within reach. (-) But, honest as these expectations may be, are they justified?"
> **(Dijkstra '75, pg. 546/547).**

Several studies were published which show that the sharing of resources can under unfavorable circumstances lead to quite excessive delays, and thus to a very low system throughput. **(Baskett & Smith '76; Smith '77; cf. Raynor '74).**
Lorin reported that:

> "The delays caused by processor interference on memory reference have been estimated at as high as 20 percent in some studies."
> **(Lorin '72, pg. 267).**

All these observations are reasons to consider the (potential) advantages of multiprocessing per application and with some suspicion. It is evident that multiprocessing systems can be much more powerful than single computer systems, but one should note that this can be an advantage and a disadvantage. The more powerful a system is, or can be, the more powerful means we must have to control that system, to restrict it where necessary. The more powerful a system is, finally, the more complete our understanding of the system should be.

1.1.3. Terminology

Two or more sequential programs are said to be *concurrent* if their executions are potentially interleaved or overlapping in time. The word 'potentially' implies here that the actual pattern of interleavings (and/or overlappings) may well be unknown, or from a theoretical point of view *indeterministic*.
We will sometimes call concurrency by interleaving *virtual concurrency* to distinguish it from concurrency by overlapping, which is then called *actual concurrency*.
A *shared object* is a system resource which can be accessed by two or more concurrent processes. Shared objects can be in hardware (e.g. peripherals, card-readers, terminals) or in software (e.g. records, queues, data).
Objects which are not shared (on a certain level of abstraction) are said to be *private objects*.
That part of a computer program in which shared objects are being accessed is called a *critical section*.
Concurrent programs which share objects are called *interacting* or *interfering*.
In other cases the programs are called *disjoint* or *non-interfering*.
Let *range(A)* denote the memory locations where program A stores its results, and let *domain(A)* denote the memory locations where program A fetches its input.
The concurrent programs A and B are necessarily disjoint if:

- $range(A) \cap domain(B) = 0$ and
- $range(B) \cap domain(A) = 0$ and
- $range(A) \cap range(B) = 0$

(Coffman & Denning '73, pg. 38).
Note that:

- $domain(A) \cap domain(B) \neq 0$

merely implies that A and B share certain input variables. This does, however, not necessarily imply that A and B will interact (provided that reading is an indivisible act, or contains no destructive cycles).

The term *multiprogramming system* will be used to indicate any single-computer system on which two or more programs can be executed in timesharing mode.
The term *multi-computer system* or *multi-processor system* will be used to indicate a system on which two or more programs can be executed in real simultaneity (that is: actual concurrency instead of virtual concurrency).
If we do not want to distinguish between these two types of systems we will use the term *multiprocessing system*.
In so far as the concurrency aspects are concerned the multiprogramming systems are very well comparable to the multi-processor systems. The same types of coordination problems can occur, and at a sufficiently high level of abstraction the two types of systems are even indistinguishable.
Enslow '76 (pg. 228) defined a multi-processor system more in terms of hardware:

> A single system, with more than one processing-unit, with shared resources, with at least one common memory, which is controlled by one operating system, and in which all processing-units are equivalent in their capabilities (they must be able to access the same shared resources and to perform the same operations on these).

In our view, the essential characteristics of the multiprocessing systems are more on the software than on the hardware level. We define a multiprocessing system therefore as a computer system which allows for:

- the concurrent execution of programs,
- the interaction of concurrent programs, and most importantly
- the enforcement of specific 'coordination rules'.

In what manner the system allows for concurrency, interaction, or the enforcement of coordination is from our present point of view irrelevant.
A precise interpretation of the term 'coordination rule' will be elaborated in chapter 4 (**The section model**).

Terms like 'concurrent programming', 'parallel programming', 'multiprocessing' etc. are considered equivalent, just like the terms 'multi-processor system' and 'multi-computer system'. Computer systems with more than one processing-unit, which do not obey all requirements listed above for multiprocessing systems, will be called *multiple computer systems* or *multi-sequential systems*.

1.2. COORDINATION PROBLEMS

1.2.1. Introductory examples

Reading and writing in one memory location by two different processes at the same time will in nearly all existing computer systems be prohibited. Clearly, the effect of two simultaneous writing cycles in one specific location would be unpredictable. Similarly the effect of a read operation which is overlapped by a write operation on the same word in memory cannot be known. This type of indeterminism is, however, undesirable. A reliable multiprocessing system should therefore at least (have means to) guarantee that all write operations and all combinations of reading and writing in the same memory locations in shared memories, will be mutually excluded.
If a read operation contains an implicit writing cycle (e.g. destructive reading) read operations must also be mutually exclusive, for the same reasons.
Note that simultaneous reading and writing is not merely a theoretical problem: it is readily possible in for instance disk memories with more than one read/write head per disk.
In practice the multiprocessing systems provide for standard primitive read and write operations of which all combinations are mutually exclusive per entire memory module. The exclusion is then built in the hardware, that is, in the access mechanism of the memory modules. In multiprogramming systems the exclusion can be realized more straightforwardly by making the reading and writing operations indivisible by interrupt-masking techniques.

Still, the basic exclusions on read and write operations discussed above cannot prevent the occurrence of a wide range of sometimes quite subtle interaction problems. We will discuss some main aspects of these problems with four examples.

a. *Concurrent counting*
Assume that we want two concurrent processes to maintain a count in a common variable. One may imagine the problem of counting the number of passing cars on a highway with two lanes. One uses two detection loops in the road, one in each lane. Each loop is connected to a different micro-processor. For each detected car the shared counting variable must be incremented by 1.
A rather naïve solution of the problem would be the following program (in Concurrent Pascal, see **Brinch Hansen '75, '77**):

```
var c: integer;
begin
      c := 0;
      cobegin
lane 1:     repeat
                  await next car;
                  c := c + 1
            forever;
lane 2:     repeat
                  await next car;
                  c := c + 1
            forever
      coend
end
```

Let us disregard the implicit synchronization relations between the flow of cars and the processes, for the moment.
We shall concentrate on the interactions of the two counting processes.
The crucial point here is that the operation 'c := c + 1' is not indivisible.

This operation will take two memory-cycles: one to fetch the old value of c, and one to return the new value. The intermediate result is stored in a private register. Assume that the subprocess *lane 1* uses the register R_1 and subprocess *lane 2* uses the register R_2. The following sequence of machine instructions is thinkable:

```
lane 1: R1 := c
lane 2: R2 := c                 (first cycle)
lane 1: R1 := R1 + 1
lane 2: R2 := R2 + 1            (updating)
lane 1: c  := R1
lane 2: c  := R2                (second cycle)
```

The result, after 2 increments of variable c, is that variable c is incremented just 1 time. The result can be still different for other sequences of machine instructions, which implies that the correctness of the result depends on the relative execution speeds of the two subprocesses.
To solve the problem outlined here, one needs a tool with which one can guarantee the indivisibility of the entire updating action 'c := c + 1' in each subprocess: one needs an *exclusion tool*.

b. *Producer-consumer problem*
The following problem is known as the 'producer-consumer' or the 'bounded buffer' problem.
Two processes share a finite buffer; one process (the producer) adds portions to the buffer, the other (the consumer) removes them. The following 'solution' is again rather naïve.

```
var buffer: 0 .. N;
begin buffer := 0;
      cobegin
producer:   repeat
                  while buffer = N do skip od;
                  add portion to buffer;
                  buffer := buffer + 1
            forever;
consumer:   repeat
                  while buffer = 0 do skip od;
                  remove portion from buffer;
                  buffer := buffer - 1
            forever
      coend
end
```

The order in which the operations are placed is essential. Note that if the ordering of operations *add portion to buffer* and 'buffer := buffer + 1' is reversed, the consumer may attempt to remove non-existing portions.
Still, the solution obviously suffers from the same problems as described in example a. The exclusion problems occur with respect to the variable 'buffer' and with respect to the access of the (physical) buffer itself.
Note also that if there would be more than just one producer or more than one consumer, the buffer may overflow, or consumers may again attempt to remove non-existing portions. In these cases we would need at least two shared variables, one for the signalling from consumers to producers and one for the signalling in the opposite direction. With a simple exclusion tool we should be able to solve the producer-consumer problems. Still, one may prefer another tool. One should note that the producer-consumer problem is not in the first place an exclusion problem. On a higher level of abstraction we can describe it as the problem of specifying a special type of ordering or *sequencing* relation

between the two types of processes involved. This sequencing problem can be solved via a systematic application of simple exclusion tools, but not really straightforwardly. It would be much better if we would have tools with which we can solve the sequencing problem directly at the (abstraction) level at which it occurs: we need a *sequencing tool*.

c. *Shared-resource problem*
We consider the problem of the exclusive usage of shared resources in a little more detail here. To solve that problem for a specific resource one might suggest the following scheme:

```
var free: boolean;
begin
      free := true;
      cobegin
process 1:  repeat
                  while  not-free do skip od;
                  free := false;
                  access shared resource;
                  free := true
            forever;
process 2   .... idem ....
      coend
end
```

The exclusion is obviously leaking. This can be shown most convincingly with the aid of the following diagrams (cf. **Denning '71, pg. 197**).
We abbreviate the line '**while not-free do skip od**' by the term 'testfree'.
The progress of *process 1* through its code is measured along the x-axis of the diagram. The progress of *process 2*, with which the first process competes for the usage of the resource, is measured along the y-axis.
If the first process completes the operation 'free := false' before the second process executes 'testfree', the exclusion is effective. Effective exclusions are indicated by drawn lines in the diagram. A drawn line can be interpreted such that 'no process can progress over a drawn line'. The combined progress of the two processes considered is translated in movements of a point in the two-dimensional plane of the diagram. The progress of the first process makes this point move to the right (that is: it increments the x-coordinate); the progress of the second process makes this point move upwards (it increments the y-coordinate).

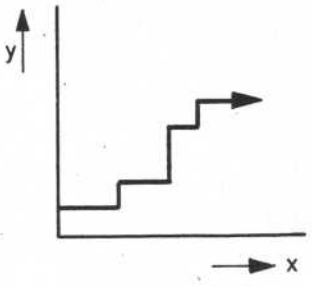

Figure 1.1.
Discrete Exclusion Diagram

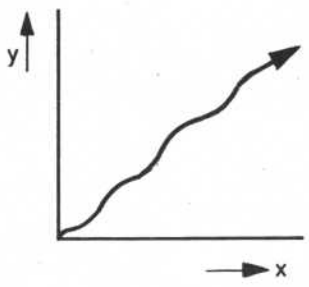

Figure 1.2.
Analogue Exclusion Diagram

Note that the function which is yielded by the translation of the combined progress of the two processes, in the x,y-plane is non-decreasing. In multi-programmed systems the function is *discrete*, as only one process can proceed at a time. In true multi-processor systems the function can be *analogue* (see figures 1.1 and 1.2).

The shaded area in the following diagram indicates those states in which both processes are in their critical sections at the same time. The exclusion is clearly only correct if this area cannot be entered, as indicated by the drawn lines (see figure 1.3).

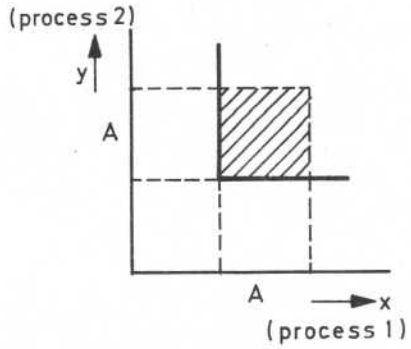

Figure 1.3.
Desired Exclusion

The symbol *A* stands for 'access shared resource'.
Let us now consider the effect of the scheme suggested earlier for the exclusion between the two competing processes (see figure 1.4).
The symbol *T* stands for 'testfree', the symbol *L* stands for 'free := false', and finally, the symbol *U* stands for 'free := true'.

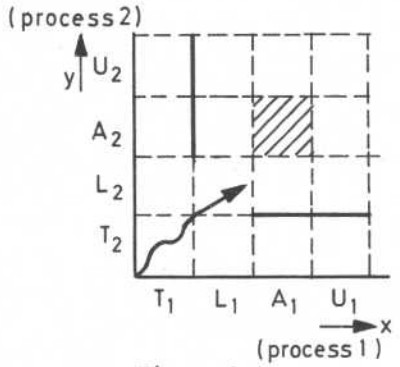

Figure 1.4.
Ineffective Exclusion

The defects of the solution are clearly visible in this diagram. One of the execution sequences which can lead into the illegal area is indicated. Reversing the order of the operations indicated by *T* and *L* is no answer (see figure 1.5).

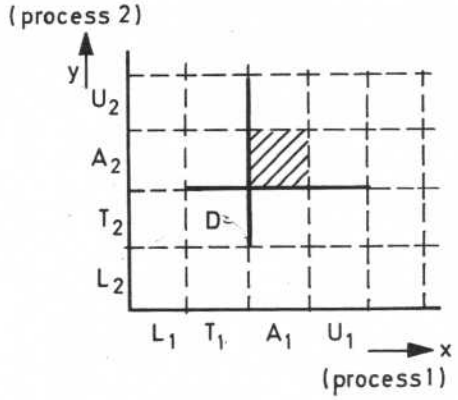

Figure 1.5. Deadlock

True enough, the illegal area cannot be reached, but this time the processes may get themselves entangled in a *deadlock* situation. If the area marked *D* is entered *process 1* will block *process 2* and vice versa. There is no way out of this mutual block.
The exclusion we would like to obtain cannot be realized with the available tools: indivisible read and write operations. We need special exclusion primitives for this.

d. *Cigarette smokers' problem*
As a last example we will consider a somewhat more complex sequencing problem. The 'cigarette smokers' problem' was introduced by **Patil '71** (in another context).
The problem is formulated as follows:

> "Three smokers are sitting at a table. One of them has tobacco, another has cigarette papers, and the third one has matches ... each one has a different ingredient to make and smoke a cigarette, but he may not give any ingredient to another.
> On the table in front of them, two of the three ingredients will be placed [**repeatedly, by a special 'supplier'; each time the supplied ingredients suffice only to make one cigarette**], ... the smoker who has the necessary third ingredient should pick up the ingredients from the table, make the cigarette and smoke it. Since a new set of ingredients will not be placed on the table until this action is completed, the other smokers who cannot make and smoke a cigarette with the ingredients on the table, must not interfere with the one who can."
> **(Patil '71).**

The following is a tentative solution of this problem. Note that it is never known beforehand which ingredients the supplier will place on the table.

```
var paper, matches, tobacco : boolean;
begin
      paper := matches := tobacco := false;
      cobegin
supplier:   repeat
                  while paper ∪ matches ∪ tobacco do skip od;
                  case random (1..3) in
                  1:    begin
                              paper := true; matches := true
                        end;
                  2:    begin
                              tobacco := true; matches := true
                        end;
                  3:    begin
                              tobacco := true; paper := true
                        end;
                  esac
            forever;
smoker T:   repeat
                  while  not (paper ∩ matches) do skip od;
                  make cigarette;
                  paper := matches := false;
                  smoke cigarette
            forever;
smoker P:   .... analogous ....
smoker M:   .... analogous ....
      coend
end
```

(Note that smoker T has a private source of tobacco, smoker P has a private source of paper, and smoker M has a private source of matches.)

At first sight there seems to be nothing wrong with this solution, but consider the following sequence of events:

- The supplier makes: tobacco := paper := true.
- Smoker T starts evaluating its iteration condition, and finds, as an intermediary result, that paper = true.
- Smoker M evaluates its iteration condition, finds it false, is therefore able to leave the idling loop, makes the cigarette and sets: paper := tobacco := false.
- The supplier consecutively finds that: paper = tobacco = matches = false, and makes: tobacco := matches := true.
- Smoker T now evaluates the second half of its iteration condition, and finds: matches = true, which makes the condition as a whole false, and allows T to escape the idling loop. Smoker T will attempt to make a cigarette, even though there is no paper on the table. Smoker T will interfere with smoker P.

This time the required sequencing relation between smokers and supplier is more complicated than it was in the producer-consumer problem.
On a low level of abstraction one can implement such a relation with rather simple exclusion tools (in this case, for instance, a tool which allows for an indivisible test & set on 2 boolean variables), but not very transparently.
We should rather have tools which allow us to describe this relation (and more in general, any type of dependence between two or more concurrent processes) in unambiguous and directly verifiable terms.

1.2.2. Problem statement

The four examples from the previous section elucidate the subtlety of interaction problems. Failures in a coordination scheme may be very hard to trace and solve indeed. One should therefore pose high demands to the verifiability of coordination structures.
To solve interaction problems one needs tools to construct exclusion and sequencing relations between the concurrent processes. One must be able to construct these relations on varying levels of complexity, but always in simple and unambiguous terms. The tools should further be free from side-effects (that is: they should not implicitly model unintended coordination relations).
Side-effects can occur in the form of deadlocks, starvations (also called 'live-locks'), unreachability of desirable system states, unintended exclusion- and/or ordering-effects etc.

> Starvation can occur when two or more processes can execute at such speeds (perhaps unintentionally) that a third process is blocked indefinitely. The starvation problem is best explained with the aid of the 'dining philosophers' problem' introduced in **Dijkstra '68a.** We discuss that problem in a later section (see section 1.3.2.3.2. **The semaphores**).

Denning '71 stressed that any coordinated system of processes should be *determinate* with respect to the shared memory cells. **(Denning '71, pg. 195).**
In Denning's view the outcome of a computation, as manifested in the contents of the shared memory cells, should depend only on the initial contents of memory cells, and not on the relative speeds of the processes.
It is perhaps a too strong requirement that the system should be determinate with respect to *all* memory cells shared. Instead we may define a subset of relevant memory cells for which the rule must hold.
Observe further that the working of a multiprocessing system is to a large extent non-deterministic. The user does not know in what order the concurrent processes are executed, which processing units will be used, or which resources, unless he is able to specify these. The interaction problem then is to ensure, in spite of the machine-level indeterminisms, that the result of the interactions is deterministic.
Müller, finally, worded it more freely as follows:

> "The *process control problem* is the problem of restricting the state sequences of a system of processes to those which lead to a correct computation."
> **(Müller '76, pg. 47).**

1.2.3. Arbitration and scheduling

We will now take a closer look at the arbitration and scheduling techniques that are necessary to implement the coordination structures refered to above.
We have seen that the interaction between concurrent processes must obey certain requirements to secure functional determinancy. We will call such requirements *coordination rules* from now on.
If the coordination rules are to be obeyed we must be able to restrict the set of possible execution sequences of the statements in the concurrent programs. That is, we must be able to enforce a partial ordering.
To be able to do this we must at least be able to *monitor* (follow) the progress of processes in the relevant parts of their computations.

Arbitration
When two or more concurrent processes arrive 'simultaneously' (as seen by the 'enforcer of the partial ordering', we will return to this later) at a point in their computation beyond which only one of them is allowed to proceed at that time, we must be able to *arbitrate*. We must be able to decide which of the

'competitors' will be enabled to proceed and which will be delayed.
In practice we may expect that on each multiprocessing system a *basic arbiter* is provided, either in hardware or in software, as a fundamental part of an operating system. This basic arbiter is invoked by the execution of a special instruction, usually called LOCK. The arbiter then guarantees that only one process at a time will be allowed to pass a LOCK instruction; each next process is delayed until the previous one has executed another special instruction called UNLOCK. The basic arbiter thus implicitly guarantees that at all times no more than one process is executing between the LOCK and UNLOCK 'brackets'.
The primary functions of the basic arbiter are therefore:

- to order, or arbitrate, and
- to exclude.

These two functions are inseparable. The enforcement of exclusions will always create an ordering problem, and alternatively a time- and speed-independent process-ordering necessitates the enforcement of exclusions. If we consider the main function of the 'basic arbiter' to order, then the required exclusion is implicit. In the sequel we will therefore sometimes refer to the 'implicit exclusion' in the basic arbiter.

We have already seen that in each multiprocessing system at least a (hardware level) exclusion mechanism must be available in the read and write operations per memory location. We may expect that this basic exclusion can be used to construct also higher level exclusions as required in the present case. This implies that we would need no additional hardware or software arbiters to resolve simultaneity conflicts. **Dekker** and **Dijkstra '65** have shown that it is indeed possible to construct algorithms which provide higher level exclusions in such a manner (see appendix A). The complexity of these algorithms, however, restricts their practical value.

Ideally the basic arbiter should obey the following four requirements:

a. The arbiter must take the decision which of a set of competing processes will be delayed, and which will be allowed to proceed, within finite time, irrespective of the relative speeds of these processes, provided that these speeds are nonzero on the average **(Dijkstra '65)**.
b. The halting of a process outside the region covered by the arbiter (that is, between LOCK and UNLOCK) may not cause blocking of other processes via the arbiter **(Dijkstra '65)**.
c. Each process which executes the LOCK instruction must be enabled to proceed beyond this instruction within N turns, when N is the maximum number of simultaneous requests to the arbiter, and a 'turn' is the acknowledgement of such a request. The arbiter may not enable any process to pass a LOCK instruction more than once while other processes are waiting. This can be called the *fairness requirement* **(Eisenberg & McGuire '72)**.
d. If two or more processes request arbitration simultaneously they must all have equal chances of gaining access as the first, the second, ... or as the last process of that batch. There should, for instance, be no static priorities to resolve simultaneity (or arbitration) conflicts. This can be called the *symmetry requirement* (see also appendix A).

The requirements are given in order of importance. Especially the fourth requirement will in most cases be a 'luxury' in arbitration functions.

Refering to our remarks on 'arbiter-algorithms', based on the available exclusions on read and write operations, we note that it is possible to construct such algorithms which indeed obey all four requirements (see appendix A). As noted, this is however not the most practical way to construct the basic arbitration functions. It is more practical to implement a LOCK instruction as a special primitive operation.

It is essential for the correct working of the primitive that the LOCK instruction be itself an indivisible operation. **Brinch Hansen '74 (pg. 241)** called it an 'amusing paradox' that one needs indivisible (mutually exclusive) operations to establish exclusions on higher levels. The word 'paradox' is indeed appropriate, that is, it is only an *apparent* contradiction. The following remarks can clarify this point:

On the 'bare hardware' of the multiprocessing machine(s) we have only coarse means to establish an exclusion between processes. In multiprogramming systems, for instance, we may establish indivisibility of operations by the setting of an interrupt mask. Such an indiscriminative mask may cause a delay for one or more other processes in the system, maximally for the duration of the interrupt mask. (The masking may affect those processes that succeed the considered process in the round-robin ready queue.) In fact it is only necessary to delay those processes that may attempt to execute conflicting operations at that time.

This strategy of 'blind exclusion' can be refined somewhat, by using *selective* interrupt-masking techniques. In that case only the reinitiation of potentially conflicting programs is masked. This is still a rather rigid technique: it is mainly hardware oriented, and difficult to extend into a method with which one can enforce more complex types of coordination relations. Clearly, many nontrivial errors may also result. If two (non-interfering) processes mask the reinitiation of the same third process in consecutive execution cycles, the first process removing the mask may **erroneously** remove the mask of the second process as well.

Another point is that even conflicting processes need only be synchronized at moments when they may actually conflict. Even selective masking is a rather coarse technique in this respect.

These 'blind' or 'nearly blind' exclusions, then, are only acceptable if the durations are extremely short (say 2 or 3 instructions).

For multi-processor systems an exclusion can be effectuated via the hardware of the memory-access mechanisms. In that case the exclusion will affect the progress of all processes that attempt to access the memory module in which a protected (sequence of) operation(s) is being executed. In the multi-programming case the 'blind exclusion' affects only the 'virtual execution times' of the delayed processes (the affected processes are suspended anyway); in the multi-processor case the 'actual execution times' are affected, as the delay concerns executing processes. In the latter case there will furthermore, most probably, be no adequate ordering discipline for waiting processes. We may conclude that also in the multi-processor case these low level exclusions are only acceptable if their durations are extremely short.

In general, the basic (low level) exclusion tools, as outlined above are inadequate to model directly the higher level coordination relations between concurrent processes. We can, however, use these basic tools to construct higher level coordination tools, which do not have so many disadvantages. A first step is then to use the low level exclusion mechanisms on the bare hardware, to construct a basic arbitration function in the form of protected LOCK and UNLOCK operations.

As a direct result of the restrictions discussed above, these primitive operations can only be of a very simple type. (Note that the duration of the execution of these operations must be brief.) They may for instance consist of the testing, setting, and resetting of special protection bits.

If the arbitration offered by the basic arbiter, thus constructed, would answer all requirements in a specific application, there would evidently be no need to build more complex coordination mechanisms on a higher level.
The basic arbiter, however, offers only one (though common) type of exclusion relation, and only one (though common) type of process scheduling mechanism (probably busy waiting with random ordering of requests).
In either of the following cases one will have to extend the basic arbitration function:

(1) The arbitration function is too simple for a specific application. For instance, it obeys the requirements (a) and (b) given above, but not requirements (c) and (d), while these are considered essential.
(2) The exclusion relations one wants to model are of another type than the standard simple type of relation implicity provided by the basic arbiter.
(3) One wants to implement another scheduling discipline than the one implicitly provided by the basic arbiter.

In each of these cases one can use the basic arbiter to construct higher level coordination mechanisms. One can for instance use the implicit exclusion of the basic arbiter, to insert a process identification in a queue. The queue can be ordered in FIFO (first in first out) or in accordance with more complex priority rules. The LOCK - UNLOCK brackets enclose the higher level enqueueing and dequeueing operations, and secure their correct working. On a higher level one can consider the new enqueueing and dequeueing operations as coordination primitives, of a very special type.
As a simple example one may consider the following algorithm, taken from **Knuth '66**. The variable t is initially zero. Array Q symbolizes the waiting queue. The variable i identifies the running process. Note that Knuth used the busy waiting strategy. The algorithm contains 3 distinct parts: (1) the insertion of a process identification in a queue, (2) a test and waiting loop, (3) the marking of the new 'head of the queue', if any.

```
        lock; t := Q(t) := i; unlock;
label:  lock;
        if i ≠ Q(0) then
              begin
                    unlock; goto label
              end;
        critical section;
        lock;
        if t = i then
              Q(0) := t := 0
        else
              Q(0) := Q(i);
        unlock;
```

The first process in the 'queue' can be enabled to access a still higher level arbiter function (the part called 'critical section' above). Note that a process should invoke the arbiter functions once more at the end of a critical section, to signal to the 'higher level arbiter' that the critical actions are completed.

In figure 1.6 a schematic representation of the different levels in the coordination mechanisms is given. On the lowest level we find the rather indiscriminative exclusion methods via interrupt-masking or memory-access protection (indicated with the symbol X). One level higher we find the basic arbiter functions LOCK (L) and UNLOCK (U). On the next level we find more abstract coordination primitives, like enter-queue (E) and leave-queue (O). On a still higher level one may construct primitives ('protected macro's or procedures) like 'produce portion' (P) and 'consume portion' (C).

Each new level is protected by, and constructed by means of, the mechanisms one level lower.

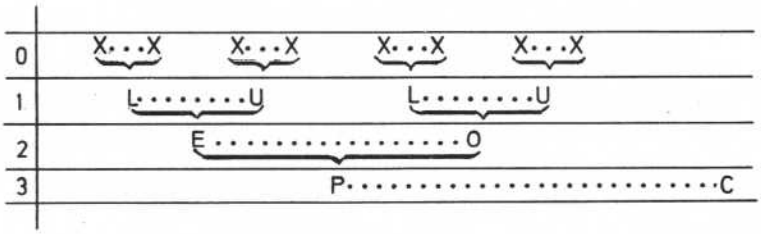

Figure 1.6.
Levelled Implementation of Coordination

Note that the order of the processes in a FIFO queue in the 'second-level arbiter' is influenced by the basic arbitration scheme at level 1. This order need not correspond completely to the actual order of arrival of the requests of the competing processes at the basic arbiter.

> Assume that the basic arbitration function does not obey the fairness requirement (which is plausible as there would otherwise be no need to construct an explicit FIFO queue on a higher level), but performs a random selection or a selection based on static priorities on all delayed processes each time a new process is allowed to pass a LOCK instruction. This leads to the conclusion that all processes that arrive at a LOCK instruction during the period of time in which some other process is executing between a LOCK and an UNLOCK instruction (that is: during the time that this lock is set) are considered to have arrived simultaneously. Their relative times of arrival are simply not taken into account. The 'grain of simultaneity', therefore, increases with the duration of the implicit exclusion in the basic arbiter. Indeed, all delayed processes which are delayed in the basic arbiter are considered to have arrived simultaneously. Their precise times of arrival are disregarded.

Traffic and service

We have seen above that the properties of the higher level arbiter(s) are influenced by the traffic load on the basic arbiter. Robinson worded this notion as follows:

> "... [**If the**] average time between successive concurrent calls is less than the average service time per call, the system cannot function. Even when the request rate is less than the service time, the efficiency of the system may be adversely affected by total exclusion ... This problem occurs often in operating systems. It must, therefore, be possible under certain circumstances, to overlap concurrent calls **[on a locking device]."**
> **(Robinson '75, pg. 2).**

Let us assume that there are N competing processes. Call λ the average number of requests for arbitration (i.e. for execution of the LOCK primitive) per time unit and per process. Call h the average duration of the implicit exclusion in the basic arbiter (i.e. the average execution time between successive LOCK and

UNLOCK instructions). As a measure of the traffic offered to the basic arbiter we can take $N.\lambda.h$. Call $\rho = N.\lambda.h$. Observe that ρ is dimensionless. It is measured in 'Erlang'. We are interested in the average number of delayed processes under these traffic conditions. The system then can be described as a Markovian birth-death process, with a finite number of sources. For one single server (the basic arbiter) and a negative exponential distribution of service times (the duration of the implicit exclusion in the basic arbiter) with mean h, we find that the average number of processes within or delayed on the basic arbiter equals:

$$\frac{\sum_{n=1}^{N} \left(\frac{n.N!}{(N-n)!} \cdot \rho^n \right)}{1 + \sum_{n=1}^{N} \left(\frac{N!}{(N-n)!} \cdot \rho^n \right)}$$

(Gross & Harris '74, pg. 119).

For given N and h, and λ approaching $(N.h)^{-1}$, the traffic density ρ approaches 1. For sufficiently large values of N we then find that the average number of processes which are within or delayed on the basic arbiter approaches N. This implies that for these values of λ and h, the basic arbiter has become a real bottle-neck. More important still is that any attempt to improve upon the fairness or the symmetry of the arbitration by the addition of a second level arbiter is bound to fail. The reason is that each process which is enabled by the basic arbiter to enter the higher level arbiter(s), for instance to enqueue a process identification in a FIFO queue, is selected on a *non*-FIFO basis from the set of delayed processes. There is no guarantee whatsoever that the ordering in the explicit FIFO queue of the higher level arbiter will correspond to the actual order of arrival of the process-requests in the basic arbiter. The relevant information is simply lost. Under these circumstances one cannot extend the basic arbiter into a higher level arbiter which will obey the fairness or symmetry requirement. Observe that this situation can arise also when the traffic density reaches temporal peak values.
If on the other hand the traffic density approaches zero, then clearly the average number of delayed processes will approach zero, and the basic arbiter will be capable of handling the larger part of the process-requests without destroying their order of arrival.

Call h_1 the mean of the service time in the basic arbiter, and
call h_2 the mean of the service time in the higher level arbiter.
The most interesting case is then of course:

$$h_1 \ll (N.\lambda)^{-1} \text{ and } h_2 \rightarrow (N.\lambda)^{-1}.$$

This time the extension of the basic arbiter into a higher level arbiter can restore the obeyance of the fairness requirement to a large extent[1]. The length of the FIFO queue in the higher level arbiter may still grow large (approaching N - 1), but the information on the arrival times need not be lost as the average number of delayed processes in the basic arbiter approaches zero. The extension of the basic arbiter is an improvement now. The same extension would, however, mean a detoriation of service when:

$$h_1 \rightarrow (N.\lambda)^{-1} \text{ and } h_2 \ll (N.\lambda)^{-1}.$$

Fortunately, the latter is not very likely to occur in practice.

— — — — —

(1) Observe, however, that the inclusion of a higher level arbiter may cause an increase of λ, especially for busy waiting strategies (see below).

Scheduling

To conclude this section we will discuss two aspects of scheduling: ordering and delaying. First some remarks about ordering. The simple arbitration functions provided in most multiprocessing systems order delayed processes mostly *at random* or with *fixed (static) priorities*. Above we have seen that one can, under certain conditions, extend these simple arbitration functions into functions which observe other types of ordering. We can order the processes explicitly in FIFO, or in accordance with some other set of priorities which are *user*-determined instead of *system*-determined.

A problem with priority ordering is that the low-priority processes may be surpassed indefinitely by the high-priority processes. It is, therefore, usually better to make the priorities a non-decreasing function of time.

Next, the delaying aspects of scheduling. What should one do with delayed processes? The simplest method is to engage such processes in an idling loop in which they repeatedly test whether they are enabled to proceed or not. This is called *busy waiting*. We have given some examples of such schemes earlier (sect. 1.2.1). The disadvantage of busy waiting is clearly that the processes engaged in an idling loop occupy system resources (at least a processing unit) which they do not really need at that moment.

We can, therefore, try to suspend delayed processes until they can proceed. This method is called *sleep waiting*. **Lampson described it as follows:**

> "The process can record somewhere the fact that it is waiting (-), and *block* itself. It will then execute no more instructions until (-) it has received a *wake-up* signal."
> **(Lampson '68, pg. 349).**

The major difficulty with sleep wait scheduling is that a sleeping process cannot by itself determine when it is time to wake-up. Someone or something else must generate the wake-up signal. The problem is then which process (or perhaps circuit, like a real-time alarm clock) should generate wake-up signals, and when.

Lampson continued:

> "An additional complication is introduced by the fact that it is not unusual for several processes to be blocked waiting for the same condition to occur. If this situation can arise, the process generating the wake-up must be prepared to send it not just to one process, but to an entire wake-up list of processes."

The latter problem can be circumvented if the strategy is followed that each newly awakened process will start by waking up precisely one other process (if any) which is waiting for the same condition, provided that this other process is non-conflicting **(cf. Kessels '77)**, see also subsection 1.3.2.3.3. **(The monitors)**.

> In the case of the basic arbiter, discussed earlier, the 'delay-condition' is merely the condition which indicates whether the lock is set or not. Note that these conditions are equal for all competing processes, but also mutually conflicting. On higher levels we can implement delay-conditions which are more complicated. Such complex delay-conditions may require whole sequences of events to become true or false, which in turn complicates the signalling problem even more.

There are essentially two approaches to the signalling problems.

(1) Each process which *may* have made a delay-condition (which may have caused running processes to be suspended) *false* will have to determine for each of the sleeping processes whether they can be resumed or not. If they can be resumed the process considered must generate the wake-up signal.

Clearly, in this case all relevant information on delay-conditions must be ac-

cessible to (that is: shared with) the signalling process, which may be impractical. The philosophy behind this first approach is that the process which causes a block will have to take the trouble to remove it completely (cf. the 'polluter pays'). This method does, however, impose stricter relations and dependences on the concurrent processes than necessary. Ideally the concurrent processes should in so far as possible be unaware of each others existence as would be possible with, for instance, the busy waiting strategy.

(2) The second approach is that each process which may have made a delay-condition false is only required to generate wake-up signals to all relevant sleeping processes, without determining in detail whether they can be resumed or not. The delayed process should then, after being awakened, re-evaluate its delay-condition, and if necessary block itself once more. This method is called *controlled busy wait* **(Brinch Hansen '73)**.

Observations similar to the above have induced some programmers to design a system with two signalling strategies **(Shrivastava '74)**. The user can then choose between the two approaches per application **(Shrivastava '74, pg. 42 - 43)**.

When we compare the busy waiting strategy to the sleep waiting strategy as such, we arrive at the following observations.
Let T_1 be the time needed to suspend a process for sleep-wait scheduling; and let T_2 be the time needed to restart that process. Let the average waiting time for each delayed process be W. Finally, let L be the (average) duration of one single busy-waiting cycle, for busy-wait scheduling[1].
We can now distinguish between two cases:

- $W < T_1$, and
- $W \geqslant T_1$.

In the first case, busy waiting will be more efficient than sleep waiting if:

- $W < T_1 + T_2 - \frac{1}{2}L$.

In the second case, busy waiting is more efficient if:

- $\frac{1}{2}L < T_2$.

Or in general: $W + \frac{1}{2}L < \max\ (T_1 + T_2,\ W + T_2)$.

We must conclude that sleep-wait scheduling is not in all cases more efficient than busy-wait scheduling. It may even be *inferior*.

Quite another problem, especially for the controlled busy-wait strategy, is to guarantee the obeyance of an ordering discipline.
After the occurrence of a signal that a delay-condition may have become false, (some) sleeping processes will have to be restarted to allow them to evaluate the relevance of that signal. If a FIFO ordering rule is to be obeyed, the restarted processes should clearly have a priority over running (newly arriving) processes in the re-evaluation of their delay-conditions. The sleeping processes are, however, in a rather unfavorable position compared to the running processes. They are deprived of their system resources (e.g. the processing unit) since they were blocked. To regain the necessary resources may take some time, and meanwhile the running processes can surpass them. The right to evaluate a delay-condition must, therefore, be passed explicitly from process to process. **(cf. Kessels '77)**.

— — — — —

(1) Observe that L, T_1 and T_2 should include the delay-times caused by the required exclusions. Observe also that these delay-times will be larger for busy-wait scheduling than for sleep-wait scheduling.

The re-awakened processes should also be given priority in the regaining of their system resources, as they may unnecessarily be blocking other competitors while waiting for the resources. (See further **Brinch Hansen '73; Coffman & Denning '73.**)

1.3. TOOLS

1.3.1. The specification of multiprocessing structures in higher level languages

We now turn to one of the more practical problems related to multiprocessing: the problem of extending the existing programming languages such that they allow for an efficient and transparent specification of concurrency.

We have phrased the problem rather loosely as the problem of 'extending existing languages', and this is indeed what one has initially attempted to do.
One should, however, note that this is not the best approach to the problem as it is phrased in the title of this section. One will never be able to exploit the full capabilities of a multiprocessing system by working with sequential programming languages extended with a few *ad hoc* primitives for concurrency. Ideally, one should develop new languages which can be used for the specification of sets of concurrent processes, and which *include* the sequential programming languages as special cases. This means working from the other end: from concurrent to sequential programming and not vice versa.

In this section we will review some of the extensions that have been suggested for the sequential programming languages. We will briefly outline some recent attempts to approach the problem 'from the other side', as described above. Some alternative ideas will be elaborated in the second part of this thesis.

The first tool one needs is a means to construct parallel structures in a program. One of the first suggestions for a new keyword in the (sequential) programming languages was from **Richards '60**, who suggested the term BRT (short for BRANCH TRANSFER). **Dreyfus '61** preferred the term SIMU. Better known is the proposal of **Conway '63** to use the pair FORK - JOIN, to be used respectively at the start and the end of a closed parallel path. Conway in fact introduced 3 different types of FORK statements and 2 types of JOIN's.
Since 1963 a flow of alternative notations has been suggested **(Bear '73, pg. 35)**. We list some of the alternatives below.

DOTOGETHER/HOLD **(Opler '65)**,
AND (to be placed between concurrent statements, instead of ";", **(Wirth '66)**,
SPLIT/ASSEMBLE **(Lehman '66)**,
FORK/QUIT **(Dennis & VanHorn '66)**,
PARBEGIN/PAREND **(Dijkstra '68a)**,
START/HALT **(Wirth '69)**,
COBEGIN/COEND **(Brinch Hansen '73,** inspired by **Dijkstra '68a)**.

Still other constructs were suggested for the parallel execution of do-loops, for instance, COFOR **(Lamport '74)**.
The problem is clearly that the language constructs should be efficient, simple, flexible and amenable to formal verifications, all in one.

The key-word pair COBEGIN/COEND answers these requirements best, and has gained the widest support of the alternatives mentioned.
We consider this proposal in a little more detail below.

The notation:

COBEGIN S_1; ... ; S_n COEND;

indicates that the statements (blocks, programs) S_1, ... , S_n *can* be executed concurrently. The statement-block itself terminates only when *all* statements

S_1, ... , S_n have terminated. (We will call this AND-termination.)
Jumps out of a concurrent block are not permitted, as this would violate the termination rule. Similarly, also jumps into a concurrent block should be prevented.

Concurrent blocks may not overlap, just like ALGOL blocks. This restriction forces the programmer to structure his programs, which can contribute to their clarity and verifiability (see chapter 3). A compiler can readily verify that every COBEGIN statement is correctly matched by a COEND. This prevents a whole class of faults, which could otherwise lead to very subtle run-time errors. There is no way to detect such errors with, for instance, the FORK/JOIN pair.

A disadvantage is that although the COBEGIN/COEND construct is simple, it is not very flexible. In fact, we can specify only one simple type of parallel structure with it: concurrent execution with AND-termination.
The need for other types of structures may be small in today's programming practice, but that may well be because most programmers still think and program mainly in terms of *sequential* programming concepts.
The restriction to one specific simple type of concurrency as such, on the other hand, fits in well with the structured programming philosophy.

The more recent proposals (e.g. **Brinch Hansen '77; Holt '78**) are far less restrictive. These proposals are based on new higher level languages for concurrent programming (Concurrent PASCAL, CSPL). They allow one to declare, initiate, and terminate concurrent processes dynamically. The enforcement of the required multiprocessing structures (the restriction of the 'full concurrency') is treated as a process coordination task, which is delegated to special coordination tools (mostly 'monitors').
We will consider the available coordination tools in detail in the next section.

1.3.2. The specification of coordination structures in higher level languages

A large number of proposals for new language constructs, that can be used for the explicit synchronization of concurrent processes, have been presented.
In most cases it has been suggested including a small set of standard indivisible operations in the existing programming languages. These standard operations are called synchronization or coordination primitives. These primitives are standarized higher level arbitration functions.
The primitives were at first defined on a rather low level of abstraction, usually in terms of synchronizations of machines on the hardware level. More recently primitives have been defined in more abstract terms.
Below we discuss some criteria for the adequateness of coordination primitives. We then present a tentative taxonomy of these tools. This section is concluded with an overview, and an evaluation of the existing proposals.

1.3.2.1. *Evaluation criteria for coordination primitives*

We can divide the criteria in two classes, first the programming tool criteria, second the implementation criteria. The criteria of the first type depend mostly on the *form* of the primitives, and the *level* at which they were defined.
Depending on the specific application of a primitive it may be decided which of the criteria mentioned under each heading will be given emphasis and which can be ignored.
The programming tool requirements are: *simplicity*, *flexibility* and *verifiability*. The implementation requirements are: *efficiency* and *robustness*.
We discuss each of these items below.

Programming tool requirements

1. *Simplicity*
   We require that the coordination primitives can be applied in a simple and straightforward way. There must be a clear relation between the coordination structure yielded by the primitives and the problem being solved. The coordination structures must be transparent, and the effect of each single operation must be clear. One must be able to describe coordination relations on varying levels of abstraction. It must be possible to distinguish exclusion effects from ordering or sequencing effects and to describe each type of relation in the appropriate way.

2. *Flexibility*
   The coordination primitives must permit (simple) solutions to a wide variety of coordination problems. One should in principle be able to construct any type of relation with these tools, in clear and unambiguous terms. The structure of each solution must be uniform. The coordination tools should further be applicable as design and analysis tools.

3. *Verifiability*
   It is very important that the coordination structures that can be built with the primitives will be amenable to some formal type of analysis. It must be verifiable whether system deadlocks are possible, whether processes may run the risk of being starved in the coordination, whether the correct type of exclusion is realized in specific program parts, whether higher level priority rules (cf. the readers & writers' problems) are correctly obeyed, etc. Ideally these properties should be amenable to an automated verification.

Implementation requirements

4. *Efficiency*
   The enforcement of the coordination rules may not lead to excessive delays. This poses restrictions on the type of arbitration and scheduling strategies that can be implemented. Further, the duration and the frequency of the exclusions on the execution of the primitive operations must be restricted to a minimum.

One may argue that also the avoidance of system deadlocks and process starvations *within* primitive operations is part of the efficiency requirement. We will consider this under the heading *robustness*, however.

5. *Robustness*
   Under the criterion 'robustness' we can group such items as:
   - *Integrity*, implying that it must be impossible to circumvent the enforcement of the coordination rules in user processes; similarly it must be impossible to disturb the correct working of the primitives by incorrect use of code.
   - *Determinancy*, implying that the effect of the execution of a primitive operation is reproducible, that is, the effect must be independent of time and the relative speeds of the processes involved in the coordination. The effect of the execution of a primitive operation may only depend on the state of the (relevant part of the) data base at the moment of execution. All interacting primitive operations should, therefore, be mutually exclusive.
   - *No deadlocks - no starvations*. No combination of processes may run the risk of getting entangled in a deadlock situation while executing the primitive operations. It may not be possible that either any single process, or any combination of processes, is starved while executing such operations. A correct exclusion between interfering primitive operations can prevent the occurrence of deadlock and starvation at this level.
   - *Fairness*. If it can occur frequently that two or more (cyclical) processes of equal priority are delayed in the same waiting queue, it may be worth-

while to secure that no process can gain access to its critical section more than once while other processes are waiting. This fairness requirement secures that no process can repeatedly 'overtake' another process in a waiting queue. (See also appendix A.)
- *Symmetry*. The fairness requirement still does not pose restrictions on the order in which simultaneous process requests (for the access to a critical section) must be acknowledged. For heavy traffic load and long waiting times it may be worth-while to provide for a random ordering of the acknowledgements of the simultaneous process requests (in so far as possible), to prevent that some processes benefit repeatedly from fixed priorities.
- *Scheduling facilities*. It should be possible, at least in general purpose multiprocessing systems, to choose between busy-waiting and sleep-waiting scheduling strategies, on the user level. It may further be desirable to have more extensive facilities for the explicit ordering of delayed processes, according to a number of standard ordering disciplines.

Some of the requirements clearly conflict. For instance, *efficiency* and *simplicity* conflicts with *flexibility*, *symmetry*, *fairness* and *scheduling facilities*. Other requirements enhance one another, like *verifiability* and *simplicity*, or *determinancy* and *no-deadlocks/no-starvations*.
Every coordination primitive must compromise between the conflicting requirements. One primitive cannot be expected to meet all criteria listed.
With respect to the programming tool criteria, the stress used to be on simplicity and flexibility, but more recently there is a shift in emphasis towards verifiability. From the implementation requirements the stress used to be on efficiency (and clearly the prerequisite determinancy and deadlock/starvation requirements), but this also is gradually diminishing. One has come to realize that it is worth-while to sacrifice some efficiency in favor of integrity and verifiability.

1.3.2.2. *Taxonomy of coordination tools*
The tools to be discussed in this section will be divided in three broad classes, according to the amount in which they make use of state variables.

(1) Those tools which are not based on the explicit use of state variables (synchronization variables). We call these the *locking primitives* (section 1.3.2.3.1).
(2) Those tools which make a limited use of state variables via a small set of permissible primitive operations. We call these the *semaphores* (section 1.3.2.3.2).
(3) Those tools which allow for an extensive (and extendable) systematic use of state variables. We call these the *monitors* (section 1.3.2.3.3).

We use the explicit use of state variables as our yardstick because state variables play an important role in correctness analysis (cf. **Howard '76; Owicki '76**). We will return to this point later (chapter 2).

A consequence of this classification is that the context in which we will discuss the known coordination tools will differ slightly from the usual. For instance, we discuss Brinch Hansen's concept of the 'critical region' in the context of the locking primitives, as in our taxonomy the region belongs to the same class as the primitives LOCK and UNLOCK. Similarly the primitive TEST&SET is treated in the context of the semaphores, and the conditional critical region in the context of the monitors.
The chosen taxonomy thus allows us to discover unexpected resemblences between the known tools, which can be quite revealing.

Still another class of tools for the construction of coordination structures will not be included in the present discussion. These tools are more closely

linked to specific correctness analysis techniques (formal specifications, path expressions, the actor model) and they will be discussed in that context.

1.3.2.3. *Overview and discussion of coordination tools*

(1) **The locking primitives**

We have already seen that we need essentially two types of interaction control: for exclusion relations and for sequencing relations. For purposes of analysis we need adequate means for the specification of coordination structures, and for actual programming we need adequate programming constructs. The locking primitives, which were among the first suggestions for new programming primitives, deal only with the exclusion aspects.

We have already discussed the pair LOCK - UNLOCK. A range of other, more or less equivalent notations has been proposed. In most cases the primitives can be extended with a parameter field in which the shared variables on which the lock is required can be identified. We mention the following tools:

OBTAIN - RELEASE		**(Anderson '65)**
PRO	(from PROTECT)	**(Lampson '68)**
WAIT	- NOTIFY	**(Spier & Organick '71)**
DEAF	- AWARE	**(Wodon '72)**
BLOCK	- WAKEUP	**(Saltzer '66; Lampson '68)**
REGION *parameter* DO	- END	**(Brinch Hansen '72)**
WITH *parameter* DO	- END	**(Hoare '72a).**

These primitives are defined such that they will guarantee the exclusive execution of a process between the first and the second part of the tool (e.g. between OBTAIN and RELEASE). The only exception is the primitive PRO which guarantees the exclusive operation for 'some fixed interval of time', say for the succeeding 10 instructions. The exclusion as such relates to those processes that (attempt to) execute pieces of code enclosed in the same locking 'brackets'. If parameters are declared, the exclusion relates only to those competing processes that have declared the same parameters in the opening bracket.

An example of a straightforward application of these tools is given below. We take the 'shared resource problem' discussed earlier (section 1.2.1). The resource will be associated with a string of characters (its name), which is explicitly declared as a shared object.

```
shared r: string
begin
        r := read;
        cobegin
process 1:      repeat
                        lock(r);
                        use shared resource r;
                        unlock(r)
                forever;
process 2:      idem ....;
        coend
end
```

In Brinch Hansen's notation the statement LOCK(r) should be replaced by 'REGION r', and the statement UNLOCK(r) by 'END'.

The last two primitives are generally indicated as *critical regions*. These critical regions are really the most important of the tools mentioned. They are the structured equivalents to the others. The difference in notation between the two is not essential (REGION can be exchanged for WITH, and vice versa). According to the definitions, every REGION (..) statement should be matched by one

specific END-statement. Critical regions may never overlap each other. They can be nested though. Rules like these were not explicitly defined for the other tools mentioned.
Ruling out overlapping locks still does not prevent all types of problems. Let us, as an example, consider two independent locks, indicated respectively by A and B. Consider two concurrent processes which contain proper nestings of these two locks, but in different orders (cf. **Lister & Maynard '76, pg. 383**).

```
process 1: .....                process 2: .....
           lock(A);                        lock(B);
           lock(B);                        lock(A);
           .....                           .....
           unlock(B);                      unlock(A);
           unlock(A);                      unlock(B);
           .....                           .....
```

The processes will run into a deadlock if they both succeed in setting their first lock and then wait for the second lock to be released.
We can prevent this type of deadlock by requiring that locks (regions) may only be nested in one specific order, for instance the order in which the corresponding shared variables were declared. The obeyance of this rule can be verified by the compiler.
For ease of wake-up signalling with sleep-wait scheduling Brinch Hansen has further stated as a rule that a critical region may only be related to one shared object at a time. The objects are to be treated as entities, though they can of course consist of complex data structures.

It can readily be verified at compile time that shared variables (if they are explicitly declared as such) are only accessed in the context of a corresponding critical region. It is similarly easy to verify that all exclusions will eventually be released, and that they will be released in the same order as they were set.
An important advantage of the critical regions is further that the structures that can be built with them are, because of their simplicity, to some extent amenable to correctness proving techniques (e.g. the invariant methods, see chapter 2).
The disadvantages of the simple tools discussed here lie mainly in their lack of flexibility.
Let us first consider the locking primitives without parameter field. In this case one is severely restricted in the type of exclusion relations that can be modelled. Note that one cannot even model two independent exclusions in the system. Using some terms from set theory we can clarify the restrictions.
The exclusion relations one can model with the locking devices without parameter field are restricted to the set of *equivalence relations* over the set of concurrent processes. These relations are *symmetric*, *reflexive* and *transitive*. The latter terms should be interpreted in this context as follows:

Symmetry

Process A excludes process B implies that process B excludes also process A. Note that there is a difference between exclusion and mutual exclusion. There are many situations in which exclusion relations are not, or not completely, symmetric. A simple example was discussed in **Lipton '75** (also discussed in section 1.3.2.3.2).
If we consider classes of processes (see part 2) symmetry implies that if process A from class I excludes all processes from class II, then by necessity all processes from class II should also exclude all processes from class I.

Reflexiveness

If process A excludes process B, then process A excludes also itself. This seems a rather unpractical type of relation, but it is in fact quite common. Observe that it does not mean that A will block itself by setting a lock for B. It means that A cannot execute the code in which it locks B more than once without blocking itself. By 'closing a door' in B's process, A must simultaneously 'close a door behind its own back'. The situation is more clear when we consider classes of processes. When process A from class I sets a lock for a process B from class II, then A will by necessity lock also all other processes in class I on that code.

Transitivity

If process A excludes process B, and independently B excludes process C, then by necessity A will also exclude process C.

The locking primitives that do contain a parameter field can of course model independent exclusion relations. But again, all such relations are by necessity always *symmetric*. More complex relations, or even simple sequencing relations can only be modelled by devious ways.

(2) **The semaphore primitives**

The first tool which differs from the locking primitives in that it makes a limited use of state variables is the indivisible test&set operation. The indivisible test&set can be considered as the predecessor and as a sort of non-structured equivalent to the actual semaphores.

The working of the test&set operation is well known: the value of the shared variable specified is evaluated, and if it equals some fixed value, the variable is set to another fixed value and the first subsequent operation in the code will be skipped; in all other cases the value of the shared variable is not changed and the subsequent operation is executed normally. The operation which can be skipped, depending on the value of the shared variable, is usually a goto-jump backwards.

As an example consider the following program with a boolean test&set operation. If the shared (boolean) variable 'm' is true, it is set to false and the goto-jump is skipped.

```
variable m: boolean;
begin
        m := true;
        cobegin
process 1:      begin
                L:      test&set(m); goto L;
                        critical section;
                        m := true
                end;
process 2:      .... idem ....
        coend
end
```

With the test&set operation we can establish symmetric mutual exclusion relations. The operation 'goto L' can be replaced for an abstract operation 'sleep', which will block the running process. The resetting of 'm' should then be extended with code for wake-up signalling (note that also this code should be protected from unintended interactions).

The essential difference between the test&set and the locking primitives discussed earlier is that it uses an explicit state variable. The value of the

variable 'm' in the example represents the state of the 'lock'. In the critical regions, discussed above, this information about the state of the lock has intentionally been hid from the user. One would therefore be inclined to call these 'higher level tools' compared to the test&set. In this taxonomy we, however, concentrate on the amount in which a tool allows us to make a structured use of state variables. Without the use of state variables it will not be possible to model a wide range of coordination relations, which was one of the requirements for the coordination tools. In that context the test&set operations can be placed better than the regions. Apart from that it is obvious that the test&set as such is not more flexible, simple, or verifiable than the critical region.
It is not difficult to give examples of situations in which the facility of the test&set operation to model states is far too restricted.

We mention two tools which are in fact equivalents of the test&set operation: the indivisible EXCHANGE **(Brinch Hansen '73, pg. 292)**, and the indivisible CORRELATE **(England '76b, pg. 209)**.
The EXCHANGE operation will interchange the values of two variables, of which one is a shared and one is a private variable. Example:

```
L: exchange(private, shared);
   if private = false then goto L;
   critical section;
   exchange(shared, private)
```

The initial value of all private variables should be 'false'. The initial value of the shared variable is 'true'. Only the process which succeeds in exchanging its private variable value 'false' for the shared variable value 'true' has the right to proceed into its critical section.
The value 'true' of the shared variable can be interpreted as a 'key' to the critical sections, of which there is only one in the system. The first process executing the EXCHANGE operation removes the key and proceeds. All other processes executing EXCHANGE operations afterwards will find that the key is missing and will wait.
The second EXCHANGE operation (after the critical section) will return the key to its place. It is possible to make the shared variable into a 'key' more literally, e.g. by defining it as a pointer which identifies a shared data field (cf. the 'queue-semaphores', defined by **Lauesen '75, pg. 380)**.
The working of the CORRELATE operation is as follows: A complete word of state-bits is read. Each bit from that word is consecutively examined, until the first 1 is encountered. The first 1 found will be set to zero, and the corresponding bit-number is placed in a register.
This type of CORRELATE operation can be used conveniently for the selection of a free resource from a pool of shared resources (that is on an implementation level).
We now turn to the large class of 'true' semaphores. We will discuss:

(a) simple semaphores,
(b) parallel semaphores,
(c) number semaphores,
(d) test semaphores,
(e) boolean test semaphores,
(f) array semaphores,
(g) dependence operations, and
(h) extended-dependence operations.

(a) *Simple semaphores*
The semaphore was introduced by Dijkstra in 1965. It is interesting to note here that the semaphore was designed as a tool for the explicit *sequencing* of

processes, and not as a mere exclusion tool. In 1969 Dijkstra recalled:

> "I had designed, together with C.S. Scholten a set of sequencing primitives for the mutual synchronization of a number of independent processors and I knew in the mean time a systematic way to use these primitives for the regulation of the harmonious co-operation between a number of sequential machines (virtual or not)."
> **(Dijkstra '69, pg. 116)** (underlinings added - GH).

Another interesting point to note is that the semaphores were at first applied on a rather low level of abstraction, judging from Dijkstra's use of terms like 'processor' and 'machine' (see underlinings) instead of 'process' and 'computation'. We will return to this point later (see also the earlier remarks on the change in the conceptual model of computer programming which multiprocessing has gradually caused.)
In 1968 Dijkstra described the semaphore concept as follows:

> "They are special purpose integer variables allocated in the universe in which the processes are embedded; they are initialized (with the value 0 or 1) before the parallel processes themselves are started. After this initialization the parallel processes will access the semaphores only via two very specific operations, the so-called synchronizing primitives. For historical reasons they are called the P operation and the V operation."
> **(Dijkstra '68b, pg. 345).**

There still appears to be no consensus on the actual meaning of the symbols P and V. The P-symbol is usually related to the Dutch word "*p*asseren" (pass by). The most fantastic explanation for the origin of this symbol stems from **McKeag, Wilson & Huxtable '76 (pg. 148)**. In their view the P represents the Dutch compound word "*p*rolagen", a combination of "proberen" (attempt) and "verlagen" (decrement).
For the V-symbol there are at least three different explanations in use. One has related it to the Dutch words "*v*erlaten" (leave), "*v*rijgeven" (release), and "*v*erhogen" (increment).
The terms "passeren" and "vrijgeven", however, seem to have been authorized by Dijkstra himself.

There are also two different definitions for the working of the conventional semaphore in use today. The difference may seem rather small on the implementation level, but it can be of great importance on the programming level. We will outline the differences below. First, we discuss the original definitions, as given by Dijkstra when he introduced the concept of the semaphore.
According to this definition the effect of the execution of a P-operation is:

- first a decrement by 1 of the specified state variable (named 'the semaphore' for short);
- then an evaluation of the *resulting* value of that semaphore, and depending on that value the resumption or the suspension of the executing process.

> "If the resulting value of the semaphore concerned is nonnegative, **[the]** process (-) can continue with the execution of its next statement; if however the resulting value is negative, **[the]** process (-) is stopped and booked on a waiting list associated with the semaphore concerned."
> **(Dijkstra '68b, pg. 345).**

The order of execution of the two operations mentioned (decrement and evaluate) is important. In the second definition this order will be reversed.

The working of the V-operation is as follows. The value of the semaphore is incremented by 1, unconditionally:

"If the resulting value of the semaphore concerned is positive, the V-operation in question has no further effect; if however the resulting value (-) is nonpositive, one of the processes booked on its waiting list is removed (-) [**and enabled to proceed**] ..."
(Dijkstra '68b, pg. 345).

Observe that the nonpositive value of the semaphore indicates that there are indeed processes waiting to proceed. The coordination requirements and the delay-conditions of all these processes is assumed to be equal. Each of these processes can, therefore, take the place of the running process, which facilitates the signalling problem considerably (see also section 1.1.2.3).

The second definition of the P-operation, for which we shall use the symbol P' in order to avoid confusion, is as follows:

- First the value of the semaphore is evaluated. If this value is positive, then the process is allowed to proceed with the second operation; if the value is nonpositive the process is suspended, and booked on a waiting list as before.
- The second operation is to decrement the value of the semaphore by one, unconditionally. (Observe that the two operations are indivisible when the semaphore is found to be positive.)

The working of the corresponding V'-operation is then as follows:

- First the length of the waiting list associated with the semaphore is evaluated. If the length is zero, the semaphore is incremented by 1 and the V'-operation is terminated. If the length is nonzero, then one of the waiting processes is removed and enabled to proceed. The resumed process is restarted directly after its P'-operation, without altering the value of the semaphore.

Observe that the value of the semaphore itself is no longer a measure for the number of waiting processes.
Another, in this case equivalent, method to implement the V'-operation, as suggested by **Presser '75**, is to increment the semaphore variable first, and then evaluate the length of the corresponding waiting list. If this length is nonzero, one of the processes can be removed and restarted in the P'-operation, directly after the first operation. To avoid ambiguity the exclusion on the execution of a primitive operation must then be passed explicitly from the signalling to the signalled process. This can be achieved by preventing the signalling process from releasing the lock which secures the exclusion between primitive operations, when the V'-operation is terminated in case it has re-awakened a process from the waiting list. The lock is then maintained in favor of that re-awakened process.

With the first definition of the semaphore operations, the semaphore variable represented the following specific state:

"If the semaphore value is nonpositive its absolute value equals the number of processes booked on its waiting list."
(Dijkstra '68b, pg. 345).

With the second definition this correspondence is no longer valid. In this case the semaphore variable represents on a higher level of abstraction 'the privilege to execute protected operations'. The privilege can be acquired only via the P'-operation. The amount in which one makes use of such a privilege can even be varied now (see below under 'number semaphores'). In that respect the second type of semaphore is much more flexible than the first one.
The length of the waiting list, which is explicit in the first type of semaphore, is hid from the users in the second type. In its place a more abstract

entity, which we have described as a 'privilege', is made explicit.

The order in which delayed processes are restarted can be determined in the wake-up strategy which is implemented. No single discipline is strictly prescribed in the definitions of the semaphore. In Dijkstra's words, this ordering discipline is 'logically immaterial' to the concept.

We have already mentioned that the semaphore was originally designed as a synchronizing or sequencing tool. It was soon realized that it could be used as a pure exclusion tool as well.

> "During system conception it transpired that we used the semaphores in two completely different ways (-). On the one hand, we have the semaphores used for mutual exclusion, on the other hand, the private semaphores."
> **(Dijkstra '68b, pg. 345).**

What Dijkstra called 'the private semaphores' is a general ordering method which we shall consider later.
First we present two simple examples of the application of semaphores, first as an exclusion tool, then as a sequencing tool.
We consider the shared resource problem, and the producer-consumer problem.

(1) *Shared resource problem*

```
semaphore mutex;
begin
        mutex := 1;
        cobegin
process 1:      repeat
                        P(mutex);
                        critical section;
                        V(mutex)
                forever;
process 2:      .... idem ....
        coend
end
```

The semaphore is explicitly declared and initialized to 1. Note that a semaphore can equally well be initialized to a value larger than 1, which can be helpful in the modelling of less simple types of coordination rules. If the semaphore is only used on the values 0 and 1, it is called a *binary* or *boolean* semaphore, otherwise it is called a *general* semaphore.
In the second example we use such general semaphores.

(2) *Producer-consumer problem*

```
semaphore full, empty;
begin
        full := 0; empty := n;
        cobegin
producer:       repeat
                        P(empty);
                        produce portion;
                        V(full)
                forever;
```

```
consumer:      repeat
                     P(full);
                     consume portion;
                     V(empty)
               forever
         coend
end
```

The buffer capacity is speficied in constant 'n'.

We now return to Dijkstra's concept of 'the private semaphores'. This method connects the semaphore with the third class of synchronization tools, in which an extensive use of state variables is foreseen.

One associates with every process a (number of) private semaphore(s) on which no other process is allowed to perform P-operations.

> "Whenever a process reaches a stage where the permission for dynamic progress depends on current values of state variables, it follows the pattern:
> P(mutex);
> 'inspection and modification of state variables including a conditional V(private semaphore)';
> V(mutex);
> P(private semaphore)."
> **(Dijkstra '68b, pg. 346).**

The conditional operation V(private semaphore) is only performed if the inspection of the state variables has learned that the process can safely proceed.

> "- otherwise, this V operation is skipped, leaving to the other processes the obligation to perform this V operation at a suitable moment."

> "Whenever a process reaches a stage where as a result of its progress possibly one (or more) of the blocked processes should now get permission to continue, it follows the pattern:
> P(mutex);
> 'modification and inspection of state variables including zero or more V operations on private semaphores of other processes';
> V(mutex)."
> **(Dijkstra '68b, pg. 346).**

An alternative method would be to delay a process until a general boolean condition B of shared (state) variables is true (cf. **Haberman '67, pg. 91**). Note that the evaluation of B must be indivisible, which can be guaranteed with a single semaphore 'mutex'.

```
L: P(mutex);
   if B then
         begin
               critical section;
               V(mutex)
         end
   else
         begin
               V(mutex);
               goto L
         end
   fi;
```

Note however that if two or more processes compete for access to their critical sections while B is false, these processes will be engaged in a very inefficient waiting cycle if the P operation is implemented with sleep waiting. In reference to the symbols used in section 1.2.3: h_2 is relatively small (the duration of the evaluation of the value of B) while h_1 is much larger, especially for sleep waiting (the duration of the execution of an unsuccessful P operation). The probability that indeed $h_1 \rightarrow (N.\lambda)^{-1}$ and $h_2 \ll (N.\lambda)^{-1}$ is quite large, so that the inclusion of such tools as P and V operations (seen as an extension of the more primitive functions LOCK(mutex) and UNLOCK(mutex)) may well result in an unnecessary detoriation of system efficiency.

Haberman '72 could prove an interesting property of semaphores, mainly by explicating still more state variables which are implicitly related to the use of semaphore operations. Haberman proved that the following relation is always obeyed. The relation is called the *semaphore invariant*.

$$n_p(s) = \min \left\{ n_w(s),\ C(s) + n_s(s) \right\},$$

where: $C(s)$ is the initial value assigned to semaphore s,
$n_w(s)$ is the total number of times P(s) has been initialized,
$n_p(s)$ is the total number of times operation P(s) has been passed.
Note that $n_p(s) \leq n_w(s)$, and that $n_w(s) - n_p(s)$ represents the number of delayed processes.
$n_s(s)$ is the total number of times operation V(s) has been executed.

The effect of the execution of the P and V operations, in terms of these implicit state variables, can then be summarized as follows:

```
P(s):   n_w(s) := n_w(s) + 1;
        while n_w(s) > n_s(s) + C(s) wait;
        n_p(s) := n_p(s) + 1.
V(s):   n_s(s) := n_s(s) + 1.
```

The semaphore invariant can be used in correctness arguments.

To facilitate the application of semaphores to more general interaction problems, a large number of extensions of the concept have been suggested. We will briefly examine some of these extensions.

(b) *Parallel semaphores*
Parallel semaphores (see for instance **Presser '75**), or PV-multiple semaphores **(Lipton '73)** allow one to evaluate and update more than one state variable within one indivisible action. This facility can sometimes be very helpful. Patil's cigarette smokers' problem, for instance, can be solved directly with these parallel semaphores, and not (without conditional statements) with the simple semaphores as defined by Dijkstra **(Patil '71)**.
Each of Patil's smokers has to evaluate 2 conditions in one indivisible action, for instance: 'Is there paper?' and 'Is there tobacco?' for the smoker with matches.
The signalling problems with parallel semaphores are somewhat larger than with simpler semaphores, as here the extension to the more complicated delay conditions which was hinted at in section 1.2.3 (*Scheduling*) is made. There is no longer a trivial connection between a single semaphore and a waiting list.
A waiting list can be associated with a composite condition which occurs in the parallel P operation, or it can be associated with a single arbitrary semaphore from such a composition, each with the appropriate signalling strategies.
By necessity, one will have to apply a 'controlled busy-wait' strategy.
Furthermore, the second type of semaphore operation (P'/V') will be the most

plausible one to apply in this case.

We give the application of parallel semaphores to another standard interaction problem from the literature: the problem of the dining philosophers.
The problem was introduced by **Dijkstra '68a**. The original problem concerned 5 philosophers. It is often used to elucidate the occurrence of circular deadlocks and starvations. Here we will concentrate on the starvation aspect, which allows us to limit the problem to only 3 philosophers.
The problem can be described as follows:

> Three philosophers are seated at a table with three plates and four forks (see figure 1.7).

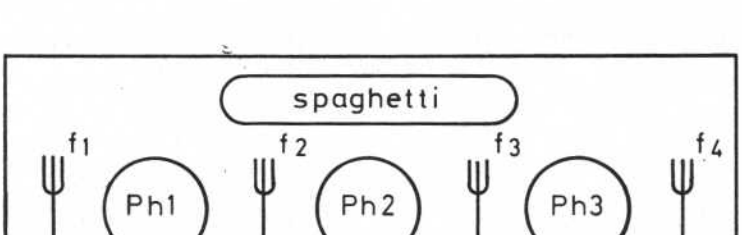

Figure 1.7.
The Dining Philosophers' Problem

> The philosophers spend their time alternately thinking and eating. In the middle of the table there is a plate with spaghetti. It is assumed that the plate is never empty. To eat the spaghetti, which is of a very difficult kind, each philosopher needs both forks lying beside his plate. Clearly, two philosophers seated at adjacent places cannot eat at the same time. They are in conflict over the fork they share.

A solution to this problem can be constructed with parallel semaphores as follows. We give the program for the i-th philosopher (i = 1, 2, 3). All semaphores are to initialized to 1.

```
philosopher i: repeat
                   P'(f_i; f_i+1);
                   eat;
                   V'(f_i; f_i+1);
                   think
               forever;
```

With each fork we associate a single semaphore variable, of which the state indicates whether the corresponding fork is in use or free.

The solution is incorrect as it does not prevent the starvation of the middle philosopher. Note that the following sequence of events is possible:

(1) Philosopher 1 starts eating, using forks 1 and 2.
(2) Philosopher 2 executes the P'-operation and is blocked, as fork 2 is missing. (He can be enqueued in a waiting list associated with fork 2).
(3) Philosopher 3 starts eating, using forks 3 and 4.
(4) Philosopher 1 returns the forks, and awakens philosopher 2. The latter

checks his forks and finds the third fork missing, and is blocked once more. (This time he can be enqueued in a waiting list associated with fork 3).

(5) Philosopher 1 finishes thinking and starts eating, again using forks 1 and 2.

(6) Philosopher 3 returns his forks, and awakens philosopher 2, who again checks his forks and finds that the second fork is missing, and is blocked. (He is again enqueued in the list associated with fork 2).

This sequence can be repeated indefinitely from point 3 to 6, with the result that the second philosopher dies from starvation. The solution is speed dependent, which makes it inadequate.

One of the problems with the semaphore operations discussed up to now is that one cannot distinguish the *evaluation* of a delay-condition from the *setting* or *changing* of such a condition. (When we consider the state of the semaphore variable as the delay-condition in question.)
Lipton '75 gave an example of a problem which normal semaphore operations cannot solve adequately. Some reflection shows that the cause is essentially this implicit link between testing and setting in the semaphore primitives.
Lipton phrased the following problem:

> There is a *supervisor* process, executing a.o. the program F, and a *user* executing a.o. the program G. The requirements in this problem are that:
>
> (1) if the supervisor executes program F before the user executes G, the user must be blocked at G, but
> (2) the supervisor may not be delayed by this synchronization.

The best solution, according to Lipton, with standard semaphore operations is to have the supervisor start its program F with the operation P(s), and to have the user start its program G with the sequence P(s); V(s).
The second problem requirement is not completely fulfilled, as the supervisor may be delayed in P(s) if the user lingers between its P(s) and V(s) operation.
Lamport '76 commented on these observations that the problem posed is no real synchronization problem, but a scheduling problem. In his view the semaphore operations abstract intentionally from scheduling details, which explains their inability to solve this problem. Lamport's distinction between synchronization and scheduling seems rather arbitrary though. Actually 'scheduling problems' like these are just a specific kind of sequencing or ordering problems which should be readily solvable with sufficiently flexible coordination tools.

The following tools are less restrictive. For some of these it implies a less efficient implementation. There is a fundamental trade-off here between flexibility and ease of use (or software efficiency), and ease of implementation (or hardware efficiency). In our view the emphasis should be on software efficiency.

(c) *Number semaphores*
Number semaphores **(Presser '75, Vantilborgh *et al.* '72)** or 'PV-Chunk' semaphores **(Lipton '73)** allow for the decreasing and increasing of semaphore variable values by any positive number specified. If the result of the P operation is nonnegative the process can again continue, else it is delayed.
An implementation in accordance with the definitions of the P'/V' operations and the controlled busy-waiting strategy is again the most plausible one (sometimes even a necessity).
We give an example of the use of such number semaphores to one of the versions of the readers' and writers' problem **(Courtois *et al.* '71)**.

> R readers and W writers share a data base. Only one writer may be writing at a time. Reading and writing may not occur simultaneously. Any number of readers may, however, be executing at the same time.

The solution we give will grant a priority to reader processes.
Each reader program, and each writer program will look as follows:

```
reader: repeat                          writer: repeat
            P'(mutex, 1);                           P'(mutex, R);
            read;                                   write;
            V'(mutex, 1)                            V'(mutex, R)
        forever                                 forever
```

The initial value of the semaphore 'mutex' should be R.
The programs will be self-explanatory.

The combination of the number and the parallel semaphore concept yields the 'parallel-number semaphores'. As an artificial example we can think of the problem of R readers which read in one data base (A) and write in another (B), and W writers which reversely write in the first data base (A) and read in the second (B).
A rather naïve solution to this problem would be as follows:

```
reader: repeat                          writer: repeat
            P'(A, 1; B, W);                         P'(A, R; B, 1);
            read (A), write (B);                    write (A), read (B);
            V'(A, 1; B, W)                          V'(A, R; B, 1)
        forever                                 forever
```

The initial value of semaphore 'A' should be R, and of 'B' it should be W.

Closer inspection of the problem shows that the above solution must be equivalent to the much simpler algorithm for readers *and* writers[1] :

```
repeat
      P(mutex);
      critical section;
      V(mutex)
forever
```

The initial value of 'mutex' can be taken equal to 1.

This instructive example elucidates that it is sometimes far from trivial to find the 'simplest' solution to a given coordination problem. We are easily misled by the power of our tools. The latter can of course be used as an argument against too powerful or too flexible coordination tools.

(d) *Test semaphores*
The semaphore operations can be extended still further by specifying in each P operation 2 distinct numbers. The effect of the operation P(s, n, δ) is then a decrement of semaphore s by n, and consecutively a test on the value (s-δ) (cf. **Presser '75**).

Test semaphores can again be combined with parallel semaphores, to yield the parallel-test semaphore. With such parallel-test semaphores we can construct a simple solution to another version of the readers' and writers' problem in which writers have a priority over readers.

– – – – –

(1) Observe that only one 'reader' or one 'writer' may be executing at a time, which is a result of the coupling of the read and write operations in both programs.

This time we use two semaphore variables; one additional semaphore 'WW' is used to count the number of waiting writers.

```
reader:                                  writer:
        P'(mutex, 1, 1; WW, 0, 0);               P'(WW, 1, -n);
                                                 P'(mutex, R, R);
        read;                                    write;
        V'(mutex, 1)                             V'(mutex, R; WW, R)
```

Semaphore WW is to be initialized to 0; semaphore mutex should be initialized to R; n can be any positive number larger than W-2. It serves only to guarantee that the writers will not be blocked in the first P operation, which is used to signal to accessing readers that writers are waiting. Note that in the worst case the W-th writer must be able to pass the first P operation when WW has already reached the value W-1.

(e) *Boolean test semaphores*
Another type of semaphore operation allows one to test on the value of arbitrary boolean conditions in the P operation. The condition to be evaluated must then be specified in the P statement: e.g. P(B), where B is a boolean variable **(Presser '75)**.

(f) *Array semaphores*
A smaller extension of the concept is to allow for semaphores to be declared in arrays. A specific semaphore variable can be identified by the array-index. **Parnas '75** showed that the cigarette smokers' problem can be solved with these array-semaphores, without using additional conditional statements (which is impossible with the conventional semaphores).

Additional semaphore operations
A rather straightforward extension of the semaphore concept is to define some additional primitive operations. A first possibility is to define an indivisible SET operation.
The effect of the execution of primitive **set**(s, n) then is that semaphore variable s is unconditionally set to the value n, after which some standard signalling functions are to be performed (depending on the implementation).

The SET primitive is especially helpful in combination with another new type of semaphore P operation: the EOR-PASSAGE **(Cerf '72, pg. 88)**[1] which is in fact the logical counterpart of the conventional parallel P'-operation.
The latter operation can only be passed if *all* semaphore variables specified can be decremented. We can indicate this type of operation as an AND-PASSAGE.
The EOR-PASSAGE can be passed if *at least one* of the semaphore variables specified can be decremented. The choice of which of these semaphores will actually be decremented is arbitrary. In the AND-PASSAGE we separate the semaphores by semicolons: $P(s_1; s_2; s_3)$. To make the distinction we will separate the semaphores in EOR-PASSAGES by colons: $P(s_1: s_2: s_3)$.
AND and EOR-PASSAGES can further be combined in so-called COMPOSITE-PASSAGES. For example: $P(s_1; s_2; (s_3: s_4))$ then indicates a barrier that can only be passed if: $s_1 > 0 \wedge s_2 > 0 \wedge (s_3 > 0 \vee s_4 > 0)$.
Composite passages can further be combined with number semaphores and/or with test semaphores.

— — — — —

(1) The term EOR is short for 'exclusive or'. Cerf used the term 'PX function'.

(g) *Dependence operations*
With dependence operations (d-operations for short) **(Wodon '72, '75; Belpaire *et al.* '73; Lipton '73)** the implicit link between the decrement and the test operations in the other semaphore-primitive proposals is removed.
In the simplest case there are three indivisible operations defined on semaphore variables:

(1) semaphore PASSAGE
Notation: s:
s is the name of a semaphore variable. The operation can be passed only if s is nonnegative. If s is negative the process attempting to execute the passage will be blocked. Execution of this operation has no effect on the value of the semaphore variable.

(2) semaphore CLOSING
Notation: **down** s
The execution of this operation is unconditional. The effect is that the value of s is decremented by 1.

(3) semaphore OPENING
Notation: **up** s
The execution of this operation is also unconditional. The effect is that the value of s is incremented by 1.

Extensions of this concept to parallel, and to number d-semaphore operations suggest themselves. Wodon and Belpaire & Wilmotte defined the following composition rule, which formalizes the extension to a special type of parallel operation.
The compositions are treated as entities, which implies that they are indivisible operations. In its general form the composition will yield a structure of the following type:

$$s_1: s_2: \ldots. s_n: \textbf{down } t_1, t_2, \ldots., t_m \textbf{ up } u_1, u_2, \ldots., u_k$$

with n, m, k nonnegative integers.
This operation can be passed if: $\sum_{i=1}^{n} s_i$
is nonnegative. Execution of the operation has the effect that all semaphore variables listed after the 'down', that is all semaphores $t_1, \ldots., t_m$, are decremented by 1, and all semaphores listed after the 'up', that is all semaphores $u_1, \ldots., u_k$, are incremented by 1.
Note that the test precedes the decrement/increment effect.

The conventional P(s) operation can be expressed in d-operations as follows:

s: down s

Similarly the conventional V(s) operation can be represented by:

up s

Reversely we can only express the working of the dependence primitives with number-test semaphores.

s: corresponds to P(s, 0, 1)
down s corresponds to P(s, 1, $-\infty$)
up s corresponds to V(s, 1)

The composite operations as defined above, however, do not correspond directly to the parallel semaphore operations we have discussed earlier.

The following program solves the first version of the readers' and writers'

problem. All semaphores are initialized to zero.

```
reader:                    writer:
    w:down r;                  r:w:down w;
      read;                        write;
      up r                         up w
```

Observe that the composite passages, as defined above, represent neither AND nor EOR-PASSAGES. The composition is not *logical* but *arithmetical*. This is quite remarkable as in process coordination problems we are mainly concerned with the specification of boolean delay-conditions, and logical combinations of such conditions. None of the tools from the semaphore-class allow us to represent these directly. To fill this gap in the available tools we will define more appropriate operations, while still basing ourselves on the distinctions between passages, openings, and closings.
We will base our further studies (part 2 and 3) on these extended d-semaphores:

(h) *Extended dependence operations*
We define two types of passages: an AND-PASSAGE and an EOR-PASSAGE.

(1) AND-PASSAGE
Notation: $s_1 \cap s_2 \cap .... \cap s_n$:
The AND-PASSAGE can be passed if the following condition is satisfied:
minimum $(s_1, s_2, ..., s_n) > 0$
The execution of an AND-PASSAGE has no effect on the value of the semaphores.

(2) EOR-PASSAGE
Notation: $s_1 \cup s_2 \cup .... \cup s_n$:
The EOR-PASSAGE can be passed if the following condition is satisfied:
maximum $(s_1, s_2, ..., s_n) > 0$
The execution of an EOR-PASSAGE also has no effect on the value of the semaphores.

AND-PASSAGES and EOR-PASSAGES can be combined in logical COMPOSITE-PASSAGES. For example, we can make the compositions:

$$(s_1 \cap s_2) \cup (s_3 \cap s_4) :$$
$$(s_1 \cap s_2 \cap (s_3 \cup s_4)) : \text{etc.}$$

The working of such operations is self-explanatory.
The normal semaphore DOWN and UP operations can be defined.
In addition we can define the indivisible semaphore SET operation: **set**(s, n).

As an example of the application of these operations we give the solution of a non-trivial coordination problem. The problem is yet a *third* version of the readers' and writers' problem. *This time neither readers nor writers may be able to monopolize the data base.* In the solution given, reader processes will be granted priority if the last process which left the data base was a writer, and vice versa. The reader will find that it is very hard if not impossible to find a comparable solution with conventional semaphore operations[1].

```
reader:                                   writer:
            down WR;                                  down WW;
AW ∩ (WW ∪ RP):down AR;          AW ∩ AR ∩ (WR ∪ WP):down AW;
            read;                                     write;
            set(RP,0; WP,1);                          set(RP,1; WP,0);
            up WR,AR                                  up WW,AW
```

— — — — —

(1) We can, however, find another solution, with non-extended d-operations, with the aid of path expressions (see section 2.3.3).

Semaphores WR, WW, and AR, AW are initially 0. Semaphore RP is initially either 0 or 1, and semaphore WP is initially (1 - RP).
Note that the coordination relations in this problem are not symmetric, reflexive or transitive.
The exclusion on the execution of primitive operations need not be such that all operations exclude one another. Consider for instance the following three operations:

(1) **down** WW
(2) AW ∩ (WW ∪ RP): **down** AR
(3) AW ∩ AR ∩ (WR ∪ WP): **down** AW

The execution of operation (1) should exclude the execution of (2).
The execution of operation (2) should exclude the execution of (3). But, the execution of operation (1) need not exclude the execution of (3).

The extended dependence operations are very helpful for the direct translation of delay-conditions and coordination rules in primitive operations.
We will return to this point in parts 2 and 3 of this dissertation.

(3) **Monitors and monitor-like tools**
In the previous section we have seen how simple semaphores can be used to synchronize concurrent processes with the aid of a set of state variables (the concept of the 'private semaphores', see pg. **36**). In the third class of coordination tools we will discuss language constructs that allow one to perform the operations on state variables in a more structured manner.
We will first discuss the concept of the 'conditional critical region', then we will turn to the concept of the true 'monitor' in Hoare's sense **(Hoare '74)**.

Conditional critical regions
The state variables are used to formulate delay-conditions. One needs safe means to *update* state variables, to *evaluate* delay-conditions, to *enforce* delays by blocks, and to *remove* such blocks. Clearly, a 'safe' access to a (set of) shared variable(s) is an exclusive access. A straightforward method to work with delay-conditions then is to use one of the simple lock-primitives in combination with primitives which allow for the evaluation of conditions, blocking and de-blocking of processes.
The protection required on the access to the state variables is of a very simple type: straight mutual exclusion.
It is, therefore, plausible to apply the safely structured 'critical region', discussed earlier, here.
We will discuss two sets of additional primitives. They were suggested as extensions to the 'critical region' concept. The critical region extended with these primitives is called the 'conditional critical region'.

(a) WAIT(e) and CAUSE(e) **(Brinch Hansen '73)**
The process executing the primitive WAIT(e) will be delayed unconditionally. An identification of the process will be entered in a (perhaps only conceptual) queue, which is identified by the parameter e. The corresponding primitive operation CAUSE(e) will de-queue all process-identifications listed in the (conceptual) queue identified by parameter e, and awake all corresponding processes. The parameter e is called an 'event variable'. Observe that the execution of a CAUSE(e) operation cannot be 'remembered', as for instance the execution of a V(s) operation with simple semaphores. This may cause the serious coordination problem of the outcome of computations depending on the relative speeds of concurrent processes.
Brinch Hansen, therefore, concluded that:

> "In particular, it will be an intolerable burden to verify that the speed assumptions are uninfluenced by modifications or extensions of a large system. We must therefore conclude that event variables of

the previous type are impractical for system design."
(Brinch Hansen '74, pg. 234).

Still, the problem outlined above can for the larger part be circumvented if we embed the event primitives in conditional clauses of the following type:

```
if B then WAIT(e);
if C then CAUSE(e);
```

The conditions B and C can then be composed of state variables.
An example of such an application, taken from **Brinch Hansen '73 (pg. 120)**, for a resource scheduling problem, is given below:

```
type P = 1 .. number of processes;
     R = 1 .. number of resources;
var U: shared record
             available: sequence of R;
              requests: queue of P;
                  turn: array P of event U;
end;

procedure reserve (process : P; var resource : R);
region U do
begin
       while empty (available) do
       begin
             enter (process, requests);
             wait (turn (process))
       end
       get (resource, available)
end;

procedure release (resource : R);
var process : P;
region U do
begin
       put (resource, available);
       if not empty (requests) then
       begin
             remove (process, requests);
             cause (turn (process))
       end
end
```

The state variables are concentrated in a single shared data structure, identified by U. These variables, and therefore also the WAIT and CAUSE operations, may only be used within a critical region on U.

The state variables are here related to the length of the queue 'requests' and the length of the sequence 'available'. These state variables are up-dated via the standard procedures 'enter(p, q), remove(p, q)' and 'put(r, s), get(r, s)' respectively. Each process has an individual event queue associated with it. This allows for explicit (pin-point) wake-up signalling strategies.

We now turn to the second pair of primitives for the blocking and de-blocking of processes in critical regions.

(b) AWAIT C and SIGNAL
The execution of the primitive operation 'AWAIT C' causes the evaluation of the specified condition C. The condition C is a general boolean expression which can refer to shared variables, but only to those shared variables that are speficied in the heading of the innermost critical region in the nesting hierarchy. **Kessels '77** suggested a still more restrictive approach in which all delay-conditions used in AWAIT statements should be declared explicitly as 'shared conditions'. This implies that the condition C cannot contain other than shared variables. The latter facilitates the implementation of an efficient, pin-point signalling strategy, as was outlined before.
(Kessels' original proposal concerned the use of the AWAIT statement in actual monitor-procedures, which will be discussed below. The principle of his suggestion, however, applies equally well to the use of AWAIT primitives in conditional critical regions.)

If the condition specified in the AWAIT statement is found to be false, the process executing this statement is delayed. The execution of the primitive SIGNAL will cause the re-evaluation of all delay-conditions, either by the signalling process (as suggested by Kessels and others), or by the delayed processes themselves in accordance with the 'controlled busy waiting' strategy (as suggested by Brinch Hansen, Hoare and others). The SIGNAL primitive is often an implicit part of the statement which releases the exclusion on a conditional critical region (that is the key-word END). With this second pair of primitives, the solution of the scheduling problem which was solved with event variables on page **45**, can be found by replacing the statements:

'wait (turn (process))' with 'await (turn (process))',
'cause (turn (process))' with 'turn(process) := true',

and changing the declaration of 'turn' in shared record U, into:

'turn : **array** P **of** boolean'
(see Brinch Hansen '73, pg. 117).

The signalling function, first performed by CAUSE(e), is now implicit in the final END of each critical region.
The new program can be used to exemplify some problems related to wake-up signalling in general:

- If no precautions are taken, the awakened processes will have to compete with newly arrived processes for the access to their critical regions.
- With the implicit SIGNAL function as defined here, it is not possible to awake one specific process from a set of delayed processes.

The second problem can be circumvented with the use of event variables.

We have already noted that arbitrary nestings of (conditional) critical regions may cause deadlock problems. As a precautionary measure one can require that nestings can only be made in one given order. Brinch Hansen required further that the AWAIT clauses may only refer to shared variables that were declared in the innermost region on the nesting hierarchy. The releasing of that restriction (as suggested by **Müller '76, pg. 58**) would confront us with the following problem.
Consider the construct:

```
region 1 do
begin
      ....
      region 2 do
      begin
            .... await C ....
      end
end
```

If condition C refers to variables that were declared in REGION 1, then processes may be deadlocked via the delaying-clause AWAIT C. Note that if during the time that a process is delayed in critical REGION 2 on condition C, the exclusion on REGION 2 is released, but the exclusion on REGION 1 is not, no other process can access the variables which make C false, and C will remain false indefinitely. To allow for the releasing of the restriction mentioned above, we would, therefore, need a more complicated implementation of the conditional critical region.

'True' monitors

From the concept of a conditional critical region it is only a small step to the concept of a monitor. All accesses to shared data are again concentrated in a set of mutually exclusive procedures (and functions).
This time the construct is defined as a special data structure.

Hoare '74 introduced the monitor concept as follows:

> "A monitor is a collection of data, procedures and functions, shared by concurrent processes."

Access to the data can only take place via the specified procedures and functions. The obeyance of this condition can be verified at compile time.
All procedures and functions within a monitor are mutually exclusive. This restriction is sometimes stronger than strictly necessary, in most applications, but it is a simple and safe one. Brinch Hansen explained:

> "... one generally needs several monitors - one for each shared variable. (One can, of course, combine all shared variables into a single shared data structure. But this becomes an unnecessary bottleneck due to the requirement of mutual exclusion of all operations on it)."
> **(Brinch Hansen '73, pg. 121).**

Remark:

It is interesting to note, also in this context, the gradual change in thinking on the task of a monitor. This shift is related to the shift in conceptual models of computer programming which was outlined in the introduction to this chapter.
Brinch Hansen defined the monitor's task as follows:

> "The purpose of a monitor is to control the scheduling of resources among individual processes according to a certain policy."
> **(Brinch Hansen '73, pg. 121).**

Hoare described the task of the monitor in similar terms **(Hoare '74, pg. 549)**. More recently, the monitor was described as:

> "... a language feature that can be used to specify interprocess communication."
> **(Kessels '77).**

Brinch Hansen '77 (pg. 22 for instance) used similar terms.
These definitions can of course be criticized. On the one hand they are too broad, as clearly every coordination primitive can be defined in this way; on the other hand they are too restrictive, as process coordination involves far more than 'resource scheduling' or 'interprocess communication' alone. What distinguishes the monitor concept from the other tools discussed is that it allows for an extensive, and safely structured, use of state variables.

A simple way to understand the working of a monitor is to consider it as a conceptual set of mutually exclusive processes, which can be invoked by user-processes. The user-processes are the *active* processes, the monitor processes are the *passive* processes (they can only respond to requests from the user-processes; the terminology is from **Brinch Hansen '77**).
The actual implementation of monitor procedures can of course be quite different.
Brinch Hansen for instance remarked:

> "I do not regard the monitor as an independent process, but rather as a software extension of the hardware structure that makes the computer more attractive for multiprogramming."

Hoare's proposal for the declaration and specification of a monitor is based on the cIass concept from SIMULA '67 (see also **Hoare '72b**).
The class concept allows one to define different monitors, identical in structure and behavior, as instances of the same class of monitors. This facility can be used when one wants to coordinate the accesses to several independent resources.

The notation suggested by **Hoare '74**, is as follows (words between parentheses are optional):

```
{class} class- or monitorname: monitor
begin
        declarations of data which is local to the monitor as a whole;
        procedure procedurename (formal parameters);
        begin
                declarations of data which is local to this procedure;
                procedurebody
        end;
        declarations of other procedures local to the monitor;
        initialization of data local to the monitor as a whole
end
```

Instances of a monitor-class are declared as:

monitor1, monitor2 : classname ;

To invoke a monitor procedure, one can simply write:

monitorname.procedurename(actual parameters);

Within a monitor procedure one may only access data which is local to the procedure, or local to the monitor as a whole.

For the modellingand manipulation of delay-conditions Hoare introduced the pair of primitives CONDVAR.WAIT and CONDVAR.SIGNAL. The first part of the primitive, CONDVAR, identifies a waiting queue. It is the programmers' task to link these waiting queues to boolean conditions, in the same manner as with the event variables (discussed earlier).
Each waiting queue must be declared explicitly, as follows:

condvar : **condition**;

The key-word CONDITION is somewhat misleading though. Hoare explained:

> "... a condition 'variable' is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition; but this queue is invisible both to waiters and to signallers."

So far, the condition variables resemble the event variables closely. The differences are mostly in the signalling strategy. Hoare defined:

> "If there are no waiting programs, the signal has no effect."
>
> "If more than one program is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting program."

Brinch Hansen defined the signalling operation of the event variables, CAUSE(e), such that *all* waiting processes would be reactivated.
Hoare continued:

> "... we decree that a signal operation be followed immediately by the resumption of a waiting program, without the possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor-entry and seize the resource instead."
>
> "... no other program can intervene between the signal and the continuation of exactly one waiting program."
> **(Hoare '74).**

The monitor exclusion, that is 'the right' to execute a monitor procedure, is thus passed explicitly from signalling process to reactivated process.

> "Only when no one further wants the privilege is **[the exclusion]** finally released."

Remark:

Hoare even went so far as requiring that a signalling process gives up its right to proceed with the execution of a monitor-procedure in favor of the reactivated process.

> "... the signalling process must wait until the resumed process permits it to proceed."

This lays a rather heavy burden on the processes performing the signalling functions. In fact, the waiting processes which are able to proceed are given a higher priority than the processes which are already executing in the monitor. To give the executing process the highest priority would be equally plausible (see for instance **Lister & Maynard '76**).
In either case it is most efficacious to concentrate all signalling functions in the monitor-releasing statement.

As a simple example of the application of a monitor we consider the solution of the 'shared resource problem', taken from **Hoare '74**.
We associate a monitor structure with the shared resource, named 'singleresource'. We need only one waiting queue, which is declared as a condition variable, named 'nonbusy'. The boolean condition 'busy' is used to indicate the state of the shared resource. It is the programmers' task to link the evaluation of this state variable to the execution of the condition variable primitives, via standard conditional statements. There are two monitor procedures, named 'acquire' and 'release'. State variable 'busy' is initialized to 'false' in the last line of the monitor declaration.

```
singleresource : monitor
begin
        busy    : boolean;
        nonbusy : condition;
        procedure acquire;
        begin
                if busy then nonbusy.wait;
                busy := true
        end;
        procedure release;
        begin
                busy := false;
                nonbusy.signal
        end;
        busy := false
end singleresource
```

A more complex monitor structure results for the solution of the third version of the readers/writers' problem[1]. The program is again taken from **Hoare '74**. We associate a class of monitors with the shared data base. For each shared record (or each shared memory module) one can declare another instance of this class. The name of the class is 'reader-and-writer'.
There are two waiting queues, declared as condition variables, named 'oktoread' and 'oktowrite'. The boolean condition 'busy' is again used to indicate the state of the record/module in question. There are four monitor-procedures, to start and terminate read and write actions.
One integer variable 'readercount' is declared as a local variable in the monitor as a whole. The state variable 'busy' indicates whether a writer-process is executing or not, the state variable 'readercount' indicates how many reader processes are executing.
The standard boolean function CONDVAR.QUEUE indicates whether there are processes waiting in the queue identified by CONDVAR.
The initial value of variable 'readercount' is zero; the initial value of variable 'busy' is false.

```
class reader-and-writer : monitor
begin readcount : integer;
      busy      : boolean;
      oktoread, oktowrite : condition;
      procedure startread;
      begin
            if busy OR oktowrite.queue then oktoread.wait;
            readercount := readercount + 1;
            oktoread.signal;
            comment once one reader can start, they all can;
      end;
      procedure startwrite;
      begin
            if readercount ≠ 0 OR busy then oktowrite.wait;
            busy := true
      end;
      procedure endread;
      begin
            readercount := readercount - 1;
            if readercount = 0 then oktowrite.signal
      end;
```

(1) Neither readers nor writers may be able to monopolize the shared data base.

```
        procedure endwrite;
        begin
              busy := false;
              if oktoread.queue then oktoread.signal else oktowrite.signal
        end;
        readercount := 0;
        busy := false
end readers-and-writers;
```

The major advantage of the 'monitor-approach' is that it allows for a structured manipulation of sets of state variables. The protection is simple and safe, namely a full exclusion on all monitor accesses.
Brinch Hansen and Holt have recently suggested the inclusion of the monitor structure in the definitions of high level languages for concurrent programming, like Concurrent Pascal, and CSPL. **(Brinch Hansen '77; Holt '78).**

An important criticism of the monitor concept is that it does not allow us to formalize coordination relations between concurrent processes directly. It does not allow one to make coordination structures more 'visible'.
As an example one may consider the procedures 'acquire' and 'release' in the monitor solution to the shared resource problem, discussed above. The higher level relations between the two (respectively producing and consuming) processes involved is not formalized and not visible in the monitor structure.
Similarly, in the solution of the third version of the readers/writers' problem the higher level relations between reader processes and writer processes are not clearly recognizable from the monitor structure. This can make problem solving and problem analysis more difficult than necessary.
Comparing the monitor solution to the solution of the same problem with extended dependence operations, discussed earlier, we feel that the latter is much more transparent and easier to understand.
The programmer (of the monitor) cannot abstract from the fact that a false delay-condition implies the blocking of a process: the programmer must make the link via relatively cumbersome conditional expressions. Similarly, the programmer cannot abstract from the fact that an access of the state variables may have made a delay-condition false, and blocked processes may be able to proceed. Clearly, all these links must be laid on some level of abstraction, but in our view this should not be the level at which the problem is being solved, and the solution being described. The (extended) dependence operations are superior to the monitor in this respect.

1.3.2.4. *Evaluation of coordination tools*

With the set of criteria discussed in section 1.3.2.1 we can evaluate the three classes of coordination tools considered.
We are immediately confronted with the problem that nearly all relevant criteria are rather subjective. There is no 'objective' measure for items like 'simplicity', 'flexibility', or 'verifiability'.
We must restrict ourselves to some general obervations.

(1) **Simplicity and flexibility**

Simplicity and flexibility are strongly related. It would be wrong to consider these items in isolation.
The lower level tools, like the locking primitives, are qua form more 'simple' than the higher level tools, like monitors. Still, the lower level tools can only solve the more comprehensive problems in devious ways (readers/writers' problems, cigarette smokers' problem). The simplicity of the solutions that can be made with a coordination tool should, therefore, be taken into account.
True enough, all tools discussed can in principle be used to solve any type of

coordination problem. With each tool we can, for instance, establish a full exclusion on specific code, and within that code we can protect the most comprehensive types of state variable manipulations. Still, not each tool is equally adequate for solving specific problems.
A too low level tool yields nontransparent and ill-proportioned solutions to the more complex problems. Similarly, a too high level tool yields lop-sided structures for the more trivial coordination problems.

Very few tools yield coordination structures which are conceptually clear representations of the problem being solved. We have given some examples of this in section 1.3.2.3. Only the extended dependence operations, and on a higher level (in user-processes) the monitor structures, approach this ideal in some respects.

With monitors it is possible to describe a coordination structure in a levelled manner, but it is not readily possible to describe and analyze the coordination relations (the dependence between the processes involved in the coordination) on varying levels of abstraction. Problem solving by stepwise refinement is not simplified in this manner.
With monitors it is not directly possible to distinguish exclusion effects from sequencing effects, nor to use such specific effects in a standard orderly way. The monitors derive their relative simplicity mainly from their structuredness. It should, however, be noted that a well-structured primitive as such does not always imply a clear structure of the solutions built with that primitive. **Bernstein '73 (pg. 81)** remarked aptly (in another context):

> "System descriptions with highly structured primitives take on the structure of the model, rather than that of the system itself."

When the structure of the coordination primitive tends to supersede the structure of the problems being solved, the transparency of the solutions is easily lost.

On the user-level the monitor constructs allow one to build simple and safe structures (via procedure-calls) for a wide variety of problems.
Furthermore, the monitor is without doubt the most carefully motivated and best elaborated proposal of the existing coordination tools. The inclusion of the monitor structures in the language Concurrent Pascal, and the use of this language is exemplified most convincingly in **Brinch Hansen '77.**

(2) **Verifiability**
There are only a few methods available (today) for the formal verification of properties of coordination problem solutions. One would like to prove, for instance, absence of deadlock and starvations, correct exclusions, obeyance of priority rules etc.
The methods that are available are mostly based on the disciplined reasoning with state variables (we consider such methods in chapter 2). A structured use of state variables simplifies such verifications. The less structured or less restrictive coordination primitives are at a disadvantage with respect to such formal verifications. With tools like the (conditional) critical region and monitors it is possible to detect a large class of programming errors at compile time as syntactical errors. Some verification methods have been explicitly designed for specific higher level tools.
Owicki '76 discussed a method for conditional critical regions. **Howard '76** considered verification methods for monitor-structures. **Brinch Hansen '73** discussed a method for the simple critical regions. In part 2 we will consider still another method, which can be used to derive verified solutions with extended dependence operations. The other tools discussed (locking primitives, semaphores) are in general not, or at least far less, amenable to verification.

(3) **Efficiency**
The efficiency of a coordination problem depends, among other things, on the problems to which it is applied. When one uses a high level tool to solve only trivial coordination problems, the solutions are most probably inefficient. Reversely, to use low level tools for complex problems will be even more inefficient. In the latter case the solutions will not only be inefficient, but also hard to find and hard to analyze or verify. Efficiency will further depend on the skill and experience of the problem solver.
Efficiency depends also on the type of implementation chosen (the number and type of 'facilities' included). Sleep-wait scheduling can be inefficient in some applications (see section 1.2.3).
The higher level primitives, which allow for the use of composite delay-conditions, will require a more extensive implementation. This, however, does not imply that such tools are in general less efficient. Tools like the extended dependence operations allow one to construct clear and straightforward solutions to most types of coordination problems. Just like in structured programming, the gain in the simplicity and directness of the programs may well exceed the relative loss in simplicity of implementation (cf. hardware efficiency versus software efficiency).
We may, therefore, expect that, under normal conditions, the overall differences in efficiency of the different tools will be rather small. It must, however, be noted that there are no comparative studies which can support this hypothesis.

(4) **Robustness**
As noted, securing the integrity of the coordination structures is easier with the more structured tools, like (conditional) critical regions and monitors. The obeyance of the other requirements listed under 'robustness' (determinancy, no deadlock/no starvation, fairness, symmetry, scheduling facilities) can in principle be secured for any primitive.
Determinancy is usually secured by the implementation of a full exclusion between all primitive operations. The implementation (on a lower level) of that exclusion must guarantee the obeyance of the other requirements, in so far as they are considered relevant.

Summarizing the results of this evaluation, we note that the (extended) dependence operations and the monitors have answered the requirements best. It seems plausible to use the (extended) dependence operations especially in design and analysis phases of a coordination structure, and to use monitors in actual implementations (the 'translation' should be straightforward).

1.4. CONCLUSIONS

- The introduction of multiprocessing has led to the development of a new conceptual model of computer programming. One has been developing methods to solve programming problems in a levelled manner. Concerns for hardware efficiency have been overruled by concerns for verifiability and simplicity of structures. The techniques available for the coordination of concurrent processes have evolved from simple, hardware oriented tools for the synchronization of physical machines, into high level, abstract programming primitives for the composition of coordination structures in conceptual processes.

- One will not be able to exploit the full potential of the multiprocessing systems by working with the conventional sequential programming languages extended with a few ad hoc synchronization primitives. Instead, one should develop new languages which can be used for the specification of sets of concurrent processes, and which include sequential programming languages as special cases.

- In general purpose multiprocessing systems it will not be sufficient to provide only one type of coordination primitive. The user must have the option to choose from a whole range of tools, differing in power and implementation (for instance, the user should at least have the choice between an implementation with busy-wait scheduling and sleep-wait scheduling). The user can then decide per application which tool suits his purposes best.

- We can distinguish between at least two principal types of coordination relations between concurrent processes: exclusion relations and ordering relations. Each of these relations can be defined on varying levels of complexity, and in different combinations with the other type of relations. From our coordination tools we may expect that they allow us to describe, to analyze and to verify each type of coordination relation in simple and transparent ways. There must always be a clear relation between the problem being solved and the coordination structure which solves that problem.

- Hardly any of the available coordination tools answer the requirements completely. The (extended) dependence operations and the monitors are relatively best. The extended dependence operations allow us to describe coordination problems in a direct way. The monitors allow us to construct simple and safe coordination patterns on the user-level. The first type of primitives can, therefore, be used best for system design and analysis; the second type of primitives can be used for the construction of coordination structures in general purpose systems.

1.5. REFERENCES

Anderson, J.P. (1965), *Program structures for parallel processing*, Comm. ACM, Vol. 8, No. 12, Dec. 1965, 786-788.

Baskett, F. & Smith, A.J. (1976), *Interference in multiprocessor systems with interleaved memory*, Comm. ACM, Vol. 19, No. 6, June 1976, 327-334.

Baer, J.L. (1973), *A survey of some theoretical aspects of multiprocessing*, Computing Surveys, Vol. 3, No. 1, March 1973, 31-80.

Belpaire, G. & Wilmotte, J.P. (1973), *A semantic approach to the theory of parallel processes*, Proc. Int. Computing Symp. 1973, Davos (Sw.), ACM, 159-164.

Bernstein, P.A. (1973), *Description problems in the modelling of asynchronous computer systems*, Techn. Report No. 48, Dept. Computer Science, University of Toronto, 1973, 116 pgs.

Bredt, T.H. (1970), *The mutual exclusion problem*, Report SU-STAN-CS-70-173, Stanford University, California, August 1970, 71 pgs.

Brinch Hansen, P. (1972), *A comparison of two synchronizing concepts*, Acta Informatica I, 1972, 190-199.

Brinch Hansen, P. (1972), A reply to: *"Comments on 'A comparison of two synchronizing concepts'..."*, Acta Informatica II, 1972, 189.

Brinch Hansen, P. (1973), *Operating system principles*, Prentice Hall, Englewood Cliffs, N.J., 1973.

Brinch Hansen, P. (1974), *Concurrent programming concepts*, Computing Surveys, Vol. 5, No. 4, Dec. 1974, 223-245.

Brinch Hansen, P. (1975), *The purpose of Concurrent Pascal*, Sigplan Notices, Vol. 10, No. 6, 1975, 305-309.

Brinch Hansen, P. (1977), *The architecture of concurrent programs*, Prentice Hall, Englewood Cliffs, N.J., July 1977, 317 pgs.

Cerf, V. (1972), *Multiprocessors, semaphores, and a graph model of computation*, PhD Thesis, University of California, L.A., Computer Science Dept., 1972, 317 pgs.

Coffman, E.G. & Denning, P.J. (1973), *Operating systems theory*, Prentice Hall, Englewood Cliffs, N.J., 1973.

Conway, M. (1963), *A multiprocessor system design*, Proc. AFIPS, 1963, Fall Joint Computer Conference, Vol. 24, Spartan Books, N.Y., 139-146.

Courtois, P.J., Heymans, T. & Parnas, D.L. (1971), *Concurrent control with readers and writers*, Comm. ACM, Vol. 14, No. 10, Oct. 1971, 667-668.

Denning, P.J. (1971), *Third generation computer systems*, Computing Surveys, Vol. 3, No. 4, Dec. 1971, 175-216.

Dennis, J.B. & Van Horn, E.C. (1966), *Programming semantics for multiprogrammed computations*, Comm. ACM, Vol. 9, No. 3, March 1966, 143-155.

Dreyfus, P. (1961), *Programming on a concurrent digital computer*, Notes of the University of Michigan, 1961, Engineering Summer Conf. 'Theory of Computing Machine Design'.

Dijkstra, E.W. (1965), *Solution of a problem in concurrent programming control*, Comm. ACM, Vol. 8, No. 9, Sept. 1965, 569.

Dijkstra, E.W. (1968a), *Cooperating sequential processes*, In: Programming Languages, F. Genuys (ed.), Academic Press New York, 1968. (Originally published as a report EWD 123, Mathematical Dept., TH Eindhoven, Netherlands, Sept. 1965.)

Dijkstra, E.W. (1968b), *The structure of the THE multiprogramming system*, Comm. ACM, Vol. 11, No. 5, May 1968, 341-346.

Dijkstra, E.W. (1969), *Complexity controlled by hierarchical ordering of function and variability*, In: Software Engineering, 1969, P. Naur & B. Randell (eds.), Scient. Affairs Div. NATO, Brussels, 114-116.

Eisenberg, M.A. & McGuire, M.R. (1972), *Further comments on Dijkstra's concurrent programming control problem*, Comm. ACM, Vol. 15, No. 11, Nov. 1972, 999.

England, D.M. (1976a), Quoted in: 'Multiprocessor Systems', 1976, p. 13, See: **Infotech.**

England, D.M. (1976b), *Design issues in multiprocessor software*, In: 'Multiprocessor Systems', 1976, 205-218, See: **Infotech.**

Enslow jnr., P.H. (1976), *Multiprocessors and other parallel systems: an introduction and overview*, In: 'Multiprocessor Systems', 1976, 219-262, See: **Infotech.**

Greif, I. (1975), *Semantics of communicating parallel processes*, PhD thesis, MIT, Cambridge, Mass., 1975, 161 pgs.

Gross, D. & Harris, C.M. (1974), *Fundamentals of queueing theory*, J. Wiley & Sons, London, 1974, 556 pgs.

Haberman, A.N. (1967), *On the harmonious cooperation of abstract machines*, PhD Thesis, TH Eindhoven, The Netherlands, 1967.

Haberman, A.N. (1972), *Synchronization of communicating processes*, Comm. ACM, Vol. 15, No. 3, March 1972, 171-176.

Hoare, C.A.R. (1972a), *Towards a theory of parallel programming*, Operating Systems Techniques, New York, Academic Press, New York, 1972.

Hoare, C.A.R. (1972b), *Notes on data structuring*, In: Structured Programming, by Dahl, O.J., Dijkstra, E.W. & Hoare, C.A.R., Academic Press, London, 1972.

Hoare, C.A.R. (1974), *Monitors, an operating system structuring concept*, Comm. ACM, Vol. 17, No. 10, Oct. 1974, 549-557. (Errata in Vol. 18, No. 2, Febr. 1975.)

Holt, R.C. *et al.* (1978), *Structured concurrent programming with operating systems applications*, Addison-Wesley, Reading, Mass., 1978.

Horning, J.J. & Randell, B. (1973), *Process structuring*, Computing Surveys, Vol. 5, No. 1, 1973, 5-31.

Howard, J.H. (1976), *Proving monitors*, Comm. ACM, Vol. 19, No. 5, May 1976, 273-279.

Infotech (1976), *Multiprocessor Systems*, Infotech State of the Art Report, 1976, 555 pgs, Infotech Int. Ltd., Berkshire, U.K.

Jurca, I. (1977), *A multiprocessor system with multitasking facilities*, PhD Thesis, TH Delft, The Netherlands, 1977, 162 pgs.

Kessels, J.L.W. (1977), *An alternative to event queues for synchronization in monitors*, Comm. ACM, Vol. 20, No. 7, July 1977, 500-503.

Kirkley, J.L. (ed.) (1977), Datamation, May 1977, p. 63.

Kosten, L. (1973), *Stochastic theory of service systems*, Pergamon Press, Oxford, 1973, 168 pgs.

Lamport, L. (1974), *The parallel execution of do-loops*, Comm. ACM, Vol. 17, No. 2, Febr. 1974, 83-93.

Lampson, B.W. (1968), *A scheduling philosophy for multiprocessing systems*, Comm. ACM, Vol. 11, No. 5, May 1968, 347-360.

Lauesen, S. (1975), *A large semaphore based operating system*, Comm. ACM, Vol. 18, No. 7, July 1975, 377-389.

Lehman, M. (1966), *A survey of problems and preliminary results concerning parallel processing and parallel processors*, Proc. of the IEEE, Vol. 54, No. 12, Dec. 1966, 1889-1901.

Lipton, R.J. (1973), *On synchronization primitives*, Thesis, Carnegie Mellon University, Pittsburgh, Pa., June 1973.

Lister, A.M. & Maynard, K.J. (1976), *An implementation of monitors*, Softw. Pract. & Experience, Vol. 6, No. 3, 1976, 377-385.

Lorin, H. (1972), *Parallelism in hardware and software: real and apparent concurrency*, Prentice Hall, 1972.

McKeag, R.M., Wilson, R. & Huxtable, D.H.R. (1976), *Studies in operating systems*, APIC Studies in data processing, No. 13, 1976, C.A.R. Hoare (ed.), Academic Press, London.

Müller, K.G. (1976), *On the feasibility of concurrent garbage collection*, PhD Thesis, TH Delft, The Netherlands, 1976, 193 pgs.

Opler, A. (1965), *Procedure oriented language statements to facilitate parallel processing*, Comm. ACM, Vol. 8, No. 5, May 1965, 306-307.

Owicki, S. & Gries, D. (1976), *Verifying properties of parallel programs: an axiomatic approach*, Comm. ACM, Vol. 19, No. 5, May 1976, 279-285.

Parnas, D.L. (1975), *On a solution to the cigarette smokers' problem (without conditional statements)*, Comm. ACM, Vol. 18, No. 3, March 1975, 181-183.

Patil, S.S. (1971), *Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes*, Project MAC, Computational Structures Group, Memo 57, Febr. 1971, MIT, Mass.

Presser, L. (1975), *Multiprogramming coordination*, Computing Surveys, Vol. 7, No. 1, March 1975, 21-44.

Raynor, R.J. (1974), *Minimization of supervisor conflict for multiprocessor computer systems*, Georgia Institute of Techn., PhD Thesis, 1974, Computer Science, 130 pgs.

Richards, P. (1960), *Parallel programming*, Technical Operations Inc., Report No. TO-B 60-27, Aug. 1960, pg. 4.

Robinson, L. (1975), *Specification and proof in problems of concurrency*, Proc. Meeting on 20 years of Computer Science, Piza, June 1975, 69-83.

Saltzer, J.H. (1966), *Traffic control in a multiplexed computer system*, PhD Thesis, Report MAC-TR-30, Project MAC, MIT, Cambridge, Mass., July 1966.

Schneck, P.B. (1974), *The myth of multiprogramming*, Softw. Pract. & Experience, Vol. 4, No. 1, 1974, 59-62.

Shrivastava, S.K. (1974), *Synchronization of concurrent processes*, PhD Thesis, University of Cambridge, U.K., 1974, 155 pgs.

Smith, A.J. (1977), *Multiprocessor memory organization and memory interference*, Comm. ACM, Vol. 20, No. 10, Oct. 1977, 754-761.

Spier, M.J. & Organick, E.I. (1971), *The MULTICS inter-process communications facility*, Proc. 2nd. Symp. on Operating System Princ., ACM, New York, 1971, 83-91.

Taylor, J.R. (1972), *Multiprocessor systems for reliability. A comparative study*, Report AERE-R-7102, UKAEA, Harwell, Berkshire, U.K., May 1972, 86 pgs.

Turski, W.M. (1968), *Soda - a dual activity operating system*, Computer J., Vol. 11, No. 2, 1968, 148-156.

Vantilborgh, H. & Lamsweerde, A. van (1972), *On an extension of Dijkstra's semaphore primitives*, Information Processing Letters, Vol. 1, 1972, North Holland Publ. Company, 181-186.

Wirth, N. (1966), A note on: *'Program structures for parallel processing'*, Comm. ACM, Vol. 9, No. 5, May 1966, 320-321. Refering to an article in: Comm. ACM, Vol. 8, No. 12, Dec. 1965, 786-788.

Wirth, N. (1969), *On multiprogramming, machine coding and computer organization*, Comm. ACM, Vol. 12, No. 9, Sept. 1969, 489-498.

Wodon, P.L. (1972), *Still another tool for synchronizing cooperating processes*, Dept. Computer Science, Rep. Carnegie Mellon Univ., Pittsburgh, Pa., Aug. 1972.

Wodon, P.L. (1975), *Concurrent processes and synchronizing operators*, Unpublished report, Febr. 1975.