1982

Synchronization of processes

Alfs T. Berztiss
University of Wollongong

THE UNIVERSITY OF WOLLONGONG

DEPARTMENT OF COMPUTING SCIENCE

DEPARTMENTAL NOTES AND PREPRINTS

SYNCHRONIZATION OF PROCESSES

Alfs T. BERZTISS

Department of Computer Science
University of Pittsburgh
PITTSBURGH, Pa. 15260
U.S.A.

Preprint No 82-11 **May 13, 1982**

P.O. Box 1144, WOLLONGONG, N.S.W. AUSTRALIA
telephone (042)-282-981
telex AA29022

SYNCHRONIZATION OF PROCESSES

1. INTRODUCTION

The study of the synchronization of processes is a very interesting field. It brings together concepts that have originated in the design of operating systems, and of high level programming languages. Also it is becoming clear that the design of algorithms for parallel execution is intimately connected with synchronization problems. Some specialized synchronization problems have arisen in the design of data base systems. Indeed, distributed data bases provide an example of distributed processing that has immense practical significance. To summarize, synchronization of processes is a universal activity whose importance is being felt throughout computer science.

The time has therefore come for the synchronization of processes to be studied as a topic in its own right. In this course I am taking such a broad viewpoint, and am trying to integrate some aspects of operating systems, languages, and parallel algorithms. However, this being a first attempt, the integration is not as thorough as I would have wished. Also, in the short time at my disposal, I am not able to discuss several very important topics, such as reliability.

2. PROCESSES, RESOURCES, PROCESSORS

To begin with we need to establish a terminology that will be used in what follows. The basic term process has been defined in a great number of ways. This is understandable because being so basic it cannot be defined. We can only give examples. They will show that the same activity can be regarded as one process at one time, and as a set of processes at another. Examples of processes:

(i) Merging of two files to produce a third.

(ii) Traversal of a file (accessing of the elements of the file in sequence).

(iii) Generation of an output file by accepting as input a sequence of records.

(iv) Output of a record.

(v) Multiplication of two matrices A and B to produce a product matrix C.

(vi) Taking the inner product of a row of a matrix A and a column of a matrix B. (This can be regarded as generating an element of the product matrix C.)

(vii) Generation of a row of C by a sequence of operations based on the traversal of one row of A and row-wise traversal of all of B.

(viii) Traversal of a row in matrix A.

(ix) Row-wise traversal of all of matrix B.

(x) Processing of the job stream submitted to a computer center.

(xi) An operating system.

(xii) An I/O system.

(xiii) Conversion by a printer of an output stream sent to it into lines printed on paper.

(xiv) A data base system in operation.

(xv) An update in a data base made necessary by the transfer of an employee from branch X of a company to branch Y of this company.

A process needs resources. For example, process (ii) needs access to the storage areas where matrices A and B are to be found, to a storage area in which it builds up the row of matrix C, and it needs a processor that does the actual work.

Resources may be dedicated or shared. For example, in process (xiii), the printer (with its control unit) is the processor, and it serves this one process alone. On the

other hand, the situation with process (xv) can be very complicated. It could well be that the employee data base is distributed over the various localities where the company has branches. Here then the process would need as resources the sections of the distributed data base found a localities X and Y, processors (both at X and Y) for carrying out the update, and a communications link between X and Y.

Consider a process as a sequence of state transitions of a machine. Moreover, the machine may be accepting an input, and generating an output. In terms of this model we can classify processes into deterministic and nondeterministic processes. In a strictly deterministic process the sequence of states that the machine goes through is unique for a particular input (or for a particular initial state of the machine in case there is no input). However, if the process shares resources with other processes, this fact should be reflected in the design of the machine, and the machine would go through different sequences of states depending on the availability or lack of availability of the resources. We then have a nondeterministic process. Still, some measure of determinism must be retained. Following Habermann (HA76), define I/O determinism: A process is I/O deterministic if its output is a function of its input (or of the initial state in case there is no input). We require I/O determinism, but can often benefit by not requiring strict determinism.

Besides classification of processes into deterministic and nondeterministic, we can classify them into sequential and parallel. A sequential process is a sequence of atomic processing steps. For example, process (ii) is by definition sequential. In a parallel process, provided more than one processor is available, some atomic processing steps may be executed simultaneously. An example is process (v). Suppose matrices A and B are both square and of order n. Then, given n^2 processors, each processor could be used to generate one of the n^2 elements of matrix C. The processors could proceed in parallel. A distinction must be made between a parallel process and concurrent processes. Regarded as process (v), the computation of C by the n^2 processors working in parallel

is a parallel process. But matrix C can also be computed by process (vi) performed n^2 times. Given the same n^2 processors, if they are associated with the n^2 instances of process (vi), and they do their processing at the same time, we have then a system of n^2 concurrent processes.

Through all this the notion of a process has remained still rather vague. It shares the vagueness with the notion of a module. We might consider processes as modules, and could find some guidance for the separation of a large task into processes in the criteria for separating a program into modules discussed by Parnas (PA72), and by Goos and Kastens (GK78).

3. SHARING OF RESOURCES

Assume that we have a programming language in which the ability to execute operations in parallel is indicated by the delimiter ||, and assume that the assignment of a value to a variable is an atomic action. Then, in executing

```
begin  i := 1  ||  i := 2  end
```

because of the indivisibility of the assignment operation the two operations have to be performed sequentially. But the sequence can be either of

```
i := 1;  i := 2;
```

and

```
i := 2;  i := 1;
```

i.e., the computation is nondeterministic.

Next, suppose that it is required to replace the current value of variable i by the result of the computation i + j + k. Suppose we have

```
i := 0;  j := 1;  k := 2;
begin  i := i + j  ||  i := i + k  end
```

Here, because of the indivisibility of the assignment operation, the two assignments are again performed in sequence, but this

does not help us. The two additions can be carried out in parallel, and the same value i = 0 could be picked up for both additions. Again the computation would be nondeterministic, with the result possibly 1 or 2. It could even be the 3 we want (in case two processors had not been available at the same time for the two additions). One solution is to regard the entire sequence of operations expressed by, say, i := i + j as an atomic operation. But where does one stop then? By carrying this too far all the benefits that we hope to gain from parallelism would become unobtainable. These trivial examples illustrate very well that sharing of resources may introduce nondeterminism, and this problem has to be attacked in a systematic manner.

A simple cure for eliminating nondeterminism is to give a process exclusive control over the resources it requires. In our instance the location associated with variable i would not be accessible to the process "i := i + k" until the process "i := i + j" completes, or vice versa. Unfortunately such mutual exclusion creates new problems, notably deadlock. How deadlock arises is well illustrated by Fig.1, which first appeared in (CE71).

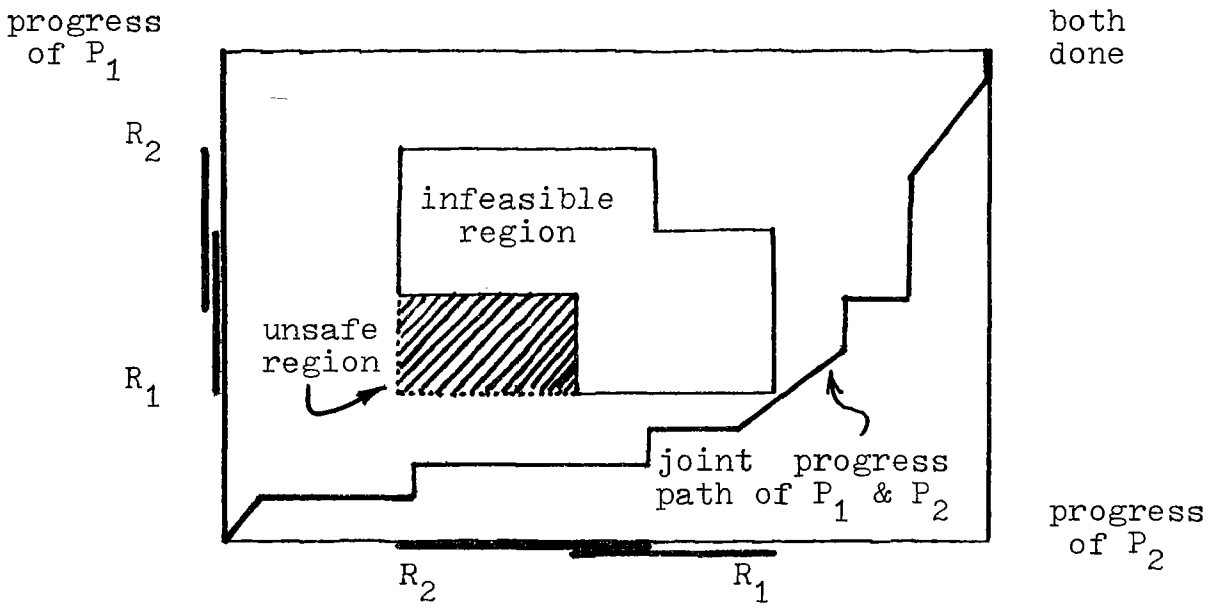

Fig.1

Fig.1 represents the progress of two processes, P_1 and P_2, both of which have to have exclusive control of resources R_1 and R_2 at some time. The time at which process P_i holds resource R_j is indicated by a bar on the P_i-axis. The infeasible region corresponds to the impossible situation of both processes having exclusive control of the same resource. The shaded region corresponds to P_1 holding R_1, and P_2 holding R_2. Neither process can proceed once the boundary between the shaded region and the infeasible region is reached, and the two processes are then said to be deadlocked. At this point, for either process to proceed, it needs both the resource it already holds and the resource held by the other process. In order to proceed, one of the processes would have to give up the resource it holds.

The joint process path consists of vertical and horizontal segments, and of sloped lines. A vertical (horizontal) segment represents process P_1 (P_2) executing alone; a sloped line represents execution of P_1 and P_2 in parallel. Parallel execution is possible unless the path hits the boundary of the infeasible or of the unsafe region. In such an event it has to move along the boundary, and only one process can be executing during this move.

4. SYNCHRONIZATION PROBLEMS

Referring back to the computation of i + j + k, the problem there is to ensure that the processes "i := i + j" and "i := i + k" are so timed or synchronized that they do not interfere with each other. There are quite a few different aspects to process synchronization, and some very interesting models have been devised to illustrate these aspects. Here we shall describe some of the models that have become classics. In all cases we shall recognize that a process can be in one of the following three states:

-- it is dormant;
-- it is running;
-- it is waiting (it wants to run, but cannot because

a resource that it needs is unavailable).

A process is called <u>active</u> when it is either waiting or running.

4.1 Dining Philosophers (DI72).

Five philosophers sit around a table, as shown in Fig.2 (there is an obvious generalization to n philosophers). A philosopher either thinks or eats--from a bowl set in the middle of the table. There is one chopstick between each pair of philosophers, and, in order to eat, a philosopher must have control of both chopsticks accessible to him or her. We shall call this the DP_5 problem (or DP_n in the general case).

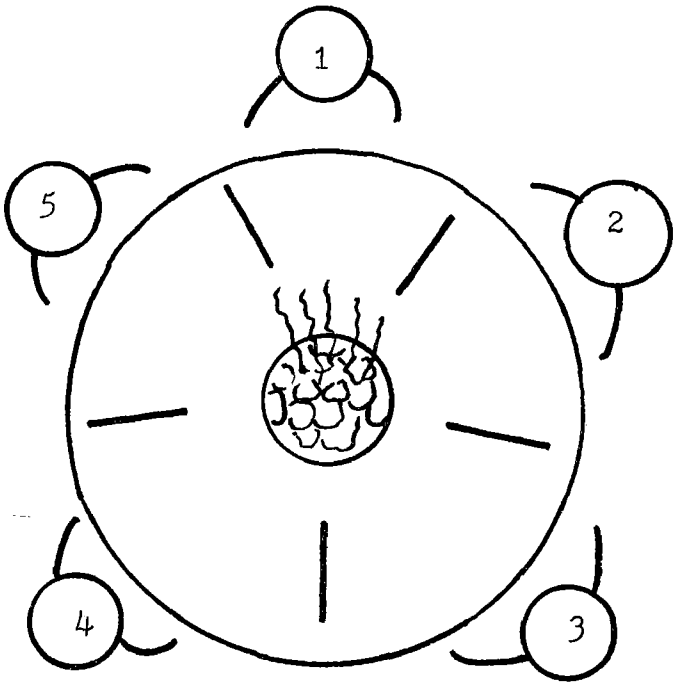

Fig.2

Obviously at most two philosophers can eat at any one time. Also, if each philosopher is in possession of one chopstick, there is deadlock. Moreover, since these are very civilized philosophers who would rather starve than eat with a single chopstick or their fingers, a phenomenon known as <u>starvation</u> (or <u>livelock</u>) may arise.

Each philosopher can be regarded as a process that alternates between the following three states:

-- it thinks;
-- it eats;
-- it wishes to eat, but cannot do so because it does not have the required resources (two chopsticks), i.e., it is in a wait state.

A system is deadlocked when all of its processes are in a wait atate. One way to avoid deadlock is to require that a philosopher may pick up chopsticks only if both the chopsticks within his or her reach are available. Now, however, it is still possible for two philosophers to take turns thinking and eating in such a way that the philosopher between them is permanently prevented from eating. This arises, for example, if the system keeps switching between the following two system states:

-- 1 and 3 think, 2 and 4 eat, 5 waits;
-- 2 and 4 think, 1 and 3 eat, 5 waits.

Starvation is prevented by the further stipulation that no philosopher is to commence eating if one of the neighbors of this philosophers has been longer in a wait state than this philosopher (R078). A further concept is fairness. This is difficult to make precise--one interpretation is that in a fair system no waiting time may exceed a specified constant bound.

4.2 Readers and Writers (CH71).

Here one has a system of r readers and w writers that all access a common data base (or some other resource). A reader may share the resource with an unlimited number of other readers, but a writer must be in exclusive control of the resource. We call this the RW problem. Two additional constraints characterize variants of the problem.

Problem RW1. As soon as a writer is ready to write, no new reader should get permission to run. Starvation of readers is a possibility here.

Problem RW2. No writer is permitted to start running if there are any waiting readers. Here it is possible to starve the writers.

4.3 Message Buffer (DI68).

The message buffer is a resource shared between a "producer" process and a "consumer" process. The producer appends messages to the buffer at one end; the consumer removes messages at the other end. This is also called the bounded buffer problem because the size of the buffer is fixed, or the producer-consumer problem. We shall denote it by MB. The usual assumption is that it takes much longer to produce and consume messages than it takes to append or remove them. Hence it is reasonable to require that that exclusive control of the buffer be given to one or other of the processes that requests it. There are two synchronization problems here: (i) the producer may not deposit a message in the buffer when it is already full; (ii) the consumer must be prevented from overtaking the producer, i.e., from trying to consume something that has not yet been produced.

4.4 Disk Head Scheduler (H074)

Suppose several processes are waiting to use a moving head disk. Average waiting time is reduced by reducing the total distance moved by the disk heads. The obvious approach is to select at all times that waiting process for which the head motion is least. Unfortunately this heuristic may localize all the action within one set of cylinders, leading to starvation of processes wishing to access cylinders outside this set. The solution is to minimize the number of changes of direction of the movement of the disk head assembly. Suppose that at some point in time the head assembly has outward direction. If there are cylinders requested by waiting processes in this direction, then these requests are acted on in the order in which the cylinders are reached. If there are no such requests, the direction changes, and the head assembly makes a sweep in the other (inward) direction. This

is known also as the elevator problem because disk head scheduling and the scheduling of an elevator in a building are essentially equivalent. Denote this problem by DH.

4.5 Smokers' Problem (PA71)

Three smokers sit around a table. Each has a permanent supply of precisely one of three resources, namely tobacco, cigarette papers, and matches, but is not permitted to give any of this resource to a neighbor. An agent occasionally makes available a supply of two of the three resources. The smoker who has the permanent supply of the remaining resource is then in a position to make and smoke a cigarette. On finishing the cigarette this smoker signals the agent, and the agent may then make again available a supply of some two resources.

The smokers are three processes, and the agent can be regarded as a set of three processes. As regards the latter, either none or exactly two of them run at any one time. The problem is to have the six processes cooperate in such a way that deadlock is prevented, e.g., that when the agent supplies paper and matches, it is indeed the smoker with the supply of tobacco who gets both, instead of one or both of these resources being acquired by the other two smokers. Call this problem TS.

4.6 Banker's Problem (DI68).

A banker has a finite amount of capital, expressed in, say, kronor. The banker enters into agreements with customers to lend money. A borrowing customer is a process. The following conditions apply:

a. The process is created when the customer specifies a "need", i.e., a limit that his indebtedness will never be permitted to exceed.

b. The process consists of transactions, where a transaction is either the advance of a krona by the banker to the customer,

or the repayment of a krona by the customer to the banker.

c. The process ends when the customer repays the last krona to the banker, and it is understood that this occurs within a finite time after the creation of the process.

d. Requests for an increase in a loan are always granted as long as the current indebtedness is below the limit established at the creation of the process, but the customer may experience a delay between the request and the transfer of the money.

Here a means has to be found for the banker to determine whether the next payment of a krona to a customer creates the risk of deadlock. This problem will be denoted BP.

4.7 On-the-Fly Garbage Collection (DØ78)

A model for the underlying representation of the list processing system Lisp consists of a "binary graph" containing a fixed number of nodes. Each node is provided with a left link and a right link, either or both of which may be missing. A link originates and terminates at a node, and is directed away from the node of which it is the left or right link. A fixed set of nodes are called "roots". A node is "live" if it is reachable from at least one root along a directed path of links. A subpath on all the live nodes is the "data structure"; the nodes that do not belong to the data structure are "garbage", and they are to be reclaimed into a "free list" for future use.

The reclamation process is performed by a garbage collector. Classically a Lisp process is interrupted when the free list is nearly exhausted, and a two-phase garbage collection takes place. In the first phase all live nodes are marked; in the second phase nodes that have remained unmarked are added to the free list.

Under on-the-fly garbage collection the Lisp process and the garbage collection process are required to execute in parallel. Call the two processes mutator and collector, respectively. Design requirements are: (i) synchronization

and exclusion constraints between the mutator and the collector are to be weak; (ii) the overhead for the mutator due to the need to cooperate with the collector is to be small; (iii) the activity of the mutator should affect minimally the ability of the collector to identify garbage. It is this need to minimize in the individual processes the perturbations brought about by the cooperation of the processes that makes this problem interesting. We denote the problem by GC.

4.8 Swimming Pool Problem (LA80)

The problem here is to synchronize the arrivals and departures at a swimming pool facility. There are two classes of resources, both in limited supply, n dressing rooms (or cubicles) and k baskets (where generally $n < k$). The process that a bather goes through:

a. Find available basket and cubicle.
b. Change into swimwear and put one's street clothes in the basket.
c. Leave cubicle and deposit the basket with the attendant.
d. Swim (the pool is assumed to have unlimited capacity).
e. Collect one's basket from the attendant.
f. Find free cubicle and change back into street clothes.

To increase the degree of possible concurrency it helps to decompose these operations. Thus (a) and (b) become:

a1. Find available cubicle.
b1. Change into swimwear.
a2. Find available basket.
b2. Put street clothes into basket.

Similarly (f) becomes:

f1. Find free cubicle and empty the basket (thus making the basket available to someone else).
f2. Change into street clothes.

Now, however, it is possible to have deadlock: Arrivals occupy cubicles waiting for baskets to become available, but in so doing lock out prospective departures from the cubicles, thus preventing baskets from becoming available. This problem will be called SP.

5. SEMAPHORES

In many of the examples discussed in Section 4 an important issue was mutual exclusion: No process is permitted into a critical region (is given access to a resource) when some other process is already in the critical region (is holding the resource). An early solution of the mutual exclusion problem was to provide lock and unlock operations. On entering a critical region a process locks it, and no other process may enter the critical region until the locking process again unlocks it at the time that it leaves it. The excluded processes have to test again and again whether or not the critical region is still locked, i.e., they are in a busy wait state. There is no queing mechanism, and this implies that at best there is no way of implementing fairness criteria, and at worst there may be starvation. Moreover, the lock and unlock affect an actual physical critical region, which is not what is always wanted (see Subsection 5.1 below).

Dijkstra (DI68) introduced operations P and V, whose purpose is to act on a semaphore. The terminology derives from railroad traffic control, where a semaphore either permits a train to pass a certain point or prevents it from doing so. When the semaphore allows a train to pass, this passage changes the semaphore setting, i.e., debars other trains from entering this section of the railroad track.

The P is short for the Dutch word Passeren (to let through), and V for Vrijgeven (to release). The purpose of P is to acquire permission for a process to enter a critical region; V signals exit from a critical region. Definition if the two operations now follows.

P(sem): If sem > 0, then it is decremented by 1, where the test and decrementation is one atomic action. Otherwise the process in which P occurs is put into a waiting queue associated with sem, i.e., it is put to sleep.

V(sem): Semaphore sem is increased by 1 in an atomic action. If there is a queue of sleeping processes, then a process is woken up.

There is an alternative formulation (see, e.g. (LA80)) in which a semaphore may have its value decreased below zero. A negative value of the semaphore indicates the existence of a queue, and its absolute value then measures the length of the queue. On the other hand, the possibility of negative values complicates the representation of semaphores by places in a Petri net, a topic to be discussed in the next section. The advantage of semaphores is that the introduction of (implicit) queues eliminates busy waiting. A disadvantage is that semaphores are a very low level construct, and as such out of place in a program written in a high level language. Implementation of semaphores is discussed in (SH74).

We now present examples of problems that are solved by means of semaphores. In these examples, because only the P and V operations may change the value of a semaphore, assignment cannot be permitted. Hence a semaphore has to be initialized in its declaration.

5.1 Mutual Exclusion.

Semaphore mutex controls access to the critical region in this rather trivial example. Two processes are shown side by side as an indication that they may proceed concurrently. Only when one process has entered its critical region does the other have to wait. The critical regions are s1 and s2, respectively. Note in particular here the increase in generality over what a lock-unlock feature provides. The lock-unlock protect one specific segment of code; here s1 and s2 may be quite dissimilar.

```
            mutex: semaphore (=1);

p1: loop                          p2: loop
        P(mutex);                         P(mutex);
        s1;                               s2;
        V(mutex);                         V(mutex);
    end loop;                         end loop;
```

5.2 Problem SP.

Here we consider the version in which the greater paralleism is possible (but also deadlock). First declare the semaphores:

```
cub: semaphore (=n);
bas: semaphore (=k);
```

Each swimmer is now represented by a concurrent process having the following structure:

```
begin
   P(cub);
   change into swimwear;          -- Action A1
   P(bas);
   fill basket;                   -- Action A2
   V(cub);
   swim;                          -- Action A3
   P(cub);
   empty the basket;              -- Action A4
   V(bas);
   change into street clothes;    -- Action A5
   V(cub);
end;
```

A new instance of this process is created for each new arrival at the swimming pool; the process dies when the swimmer departs from the facility.

5.3 Problem MB.

Three semaphores are needed: "busy", which ensures that only one process has access to the message buffer at the one time; "queue", which counts the number of messages waiting to be appended to the buffer; and "space", which counts the number of free positions in the buffer. Procedures "producer" and "consumer" are written side by side to indicate that they may run concurrently. They run forever. The size of the buffer is n.

```
busy:  semaphore (=1);
queue: semaphore (=0);
space: semaphore (=n);

producer:                    consumer:
loop                         loop
   produce message;             P(queue);
   P(space);                    P(busy);
   P(busy);                     extract message;
   append message;              V(busy);
   V(busy);                     V(space);
   V(queue);                    process message;
end loop;                    end loop;
```

5.4 Problem TS.

This was originally proposed as a problem that cannot be solved in a simple fashion by semaphores (PA71). However, Parnas has given a solution in terms of an array of semaphores that is still reasonably simple (PA75). It consists of twelve processes. Each of the twelve processes is running unless it has been put in a wait state by a P-operation.

Semaphores tobacco, paper, matches report a supply of these three respective resources; mutex and s are mutual exclusion semaphores; semaphores smokert, smokerp, smokerm are used by the smokers to report that they are done with the resources; C is an array of semaphores which have to match up a pair of resources with the smoker who has the permanent supply of the third resource. The primary purpose of this example is to illustrate the degree of complexity that one faces in programming synchronization problems. First, let us declare the semaphores, and also a counter t:

```
tobacco, paper, matches: semaphore (=0);
smokert, smokerp, smokerm: semaphore (=0);
mutex, s: semaphore (=1);
C: array (1 .. 6) of semaphore (=0);
t: integer (=0);
```

The processes come in four groups of three.

Agent processes:

```
at: loop            ap: loop            am: loop
      P(s);               P(s);               P(s);
      V(paper);           V(tobacco);         V(paper);
      V(matches);         V(matches);         V(tobacco);
    end loop;           end loop;           end loop;
```

Order processes:

```
ot: loop            op: loop            om: loop
      P(smokert);         P(smokerp);         P(smokerm);
      V(s);               V(s);               V(s);
    end loop;           end loop;           end loop;
```

Policing (coordinating) processes:

```
pt: loop            pp: loop            pm: loop
      P(tobacco);         P(paper);           P(matches);
      P(mutex);           P(mutex);           P(mutex);
      t := t+1;           t := t+2;           t := t+4;
      V(C(t));            V(C(t));            V(C(t));
      V(mutex);           V(mutex);           V(mutex);
    end loop;           end loop;           end loop;
```

Finally the smoker processes themselves:

```
st: loop            sp: loop            sm: loop
      P(C(6));            P(C(5));            P(C(3));
      t := 0;             t := 0;             t := 0;
      smoke;              smoke;              smoke;
      V(smokert);         V(smokerp);         V(smokerm);
    end loop;           end loop;           end loop;
```

6. PETRI NETS

A popular formal device for studying process synchronization is the Petri net. A Petri net is a directed graph in which there are two types of nodes, called places and transitions, which denote conditions and events, respectively. No two nodes of the same type may be adjacent to each other. Consider the

The problem of Fig.4 is essentially one of sequencing the computational steps so that they do not interfere with each other, and the problem of mutual exclusion does not arise there, but mutual exclusion can be represented very graphically by Petri nets. The net of Fig.5 models the system of Subsection 5.1. We represent a semaphore by a place in the net. In Fig.5 this place is named mutex. A V-operation adds a token to the place representing a semaphore; a P-operation removes a token.

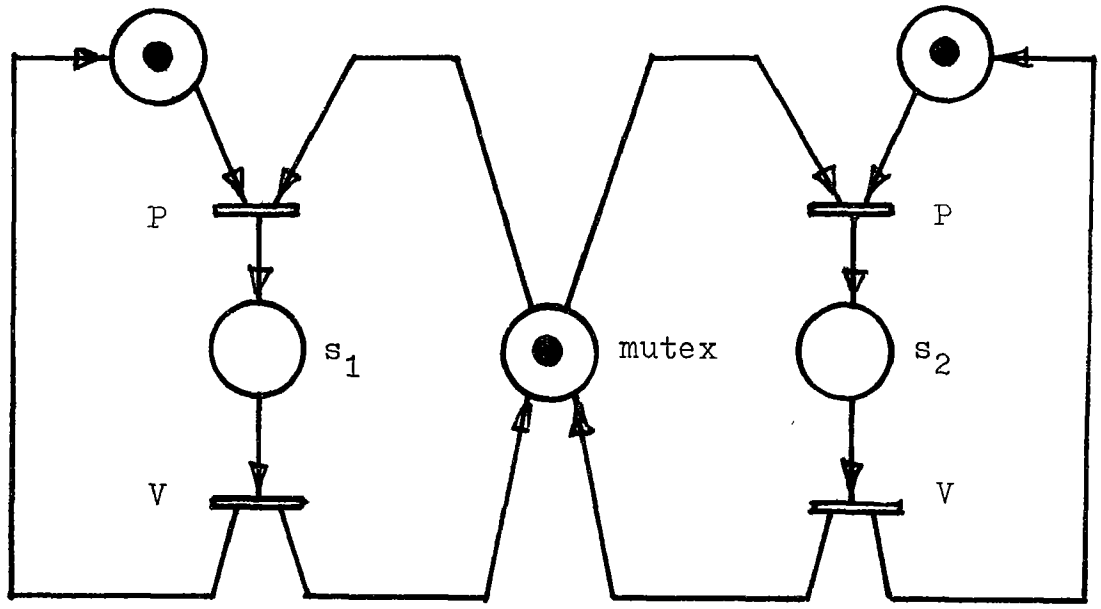

Fig.5

Next we consider Petri net analysis of problem SP, which shows that deadlock is possible there. This analysis is taken from (LA80). The system is shown in Fig.6. Since every user of the swimming pool goes through the same actions (A1 through A5, see p.15) the Petri net displays both the sequence of events as they relate to an individual user of the pool, and the behavior of the system as a whole.

The initial labeling M_0 is defined by

$$M_0(1) = n,$$
$$M_0(5) = k,$$
$$M_0(i) = 0, \quad i \neq 1 \text{ and } i \neq 5.$$

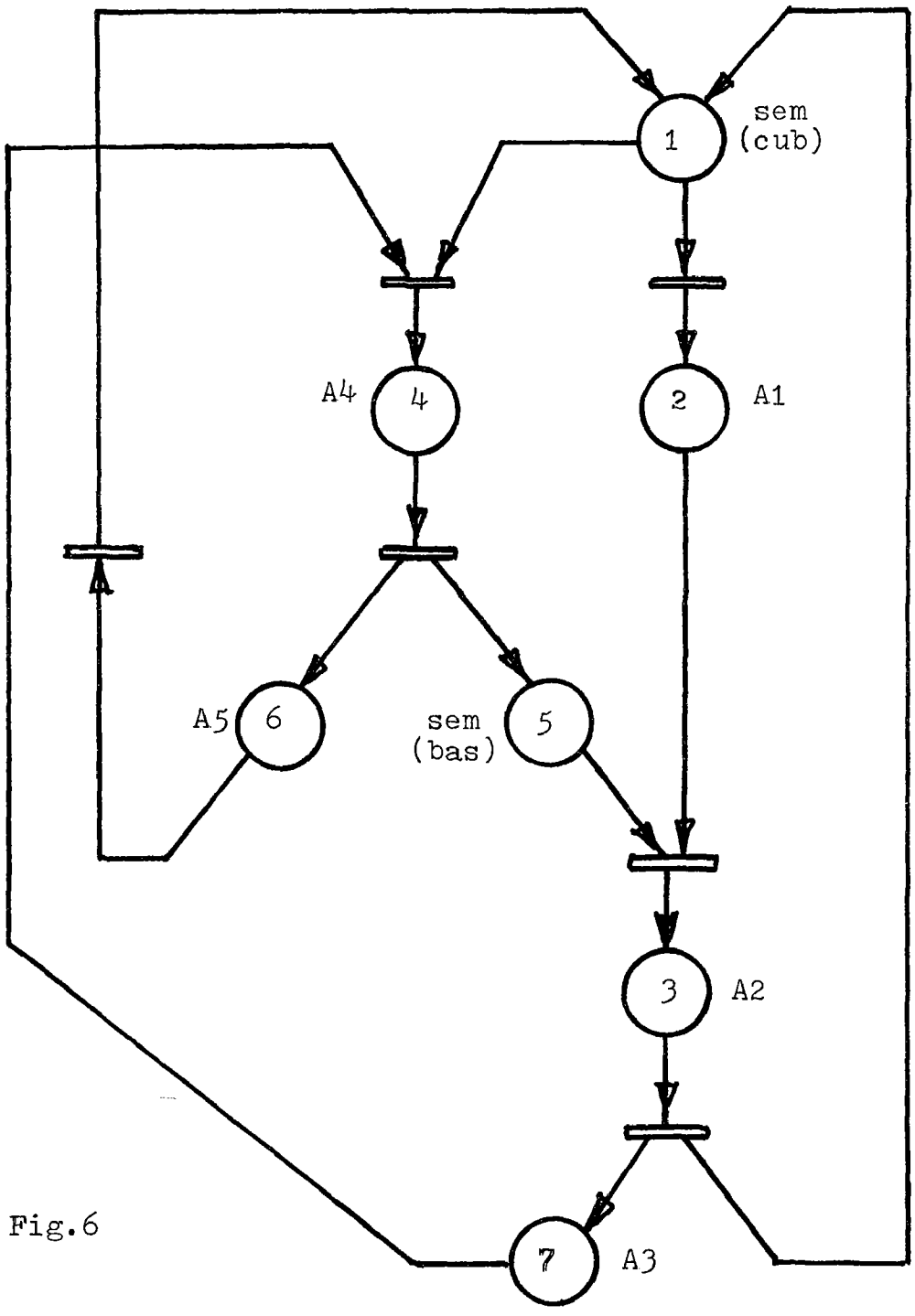

Fig.6

The significance of a labeling M, obtained from M_0 as a result of a sequence of firings, is as follows:

M(1) - number of free cubicles;
M(2) - number of cubicles in use for changing into swimwear;
M(3) - number of cubicles in use for filling baskets;
etc.

set of arcs terminating at and originating from an event node. Call the set of nodes at which the first set of arcs originates the enabling set of the event, and call the set of nodes at which the second set of arcs terminates the enabled set of this event. A net may be used to model the behavior of a system: The holding of a condition in the system may be indicated by placing a token into the corresponding place in the net. An event is enabled to fire if all nodes in its enabling set hold tokens, i.e., if all of the prerequisite conditions for the firing are satisfied. The firing of an event consists of removing a token from each node in its enabling set, and adding a token to each node of its enabled set. A distribution of tokens over the places of the net is called a marking of the net. A succession of firings of events produces a sequence of markings, and such a sequence of markings may be used to model the behavior of a system of concurrent processes. Fig.3 shows a Petri net before and after it has fired. Circles denote places; bars denote transitions. The black dots within circles represent tokens.

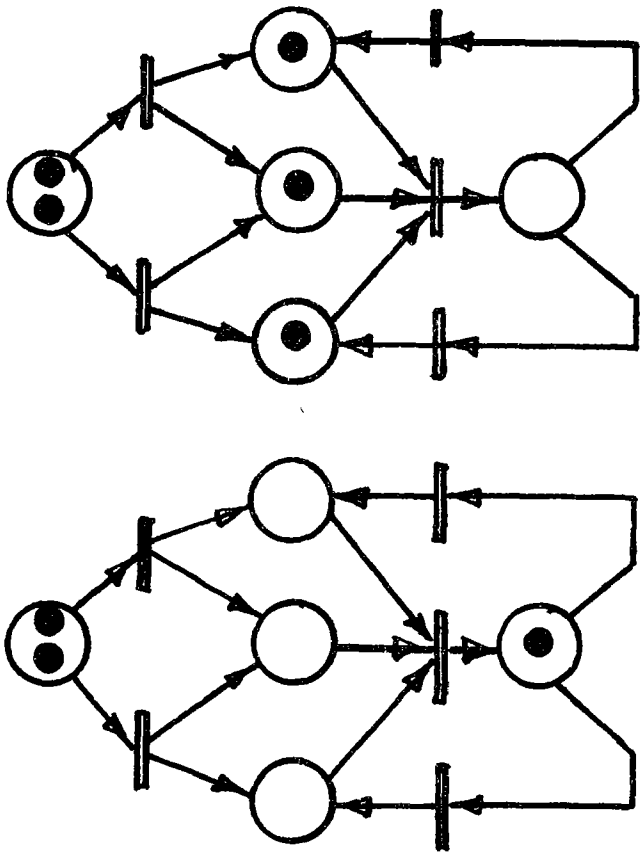

Fig.3

A Petri net can be used to model the behavior of a system of concurrent processes. Consider the computation of the factorial of a number n. This can be broken down into two processes:

(a) Decrementation of n by 1, and test of n against zero;

(b) Accumulation of the factorial f by succesive multiplication of the current value of f with k = n, n-1, n-2, etc.

To begin, f has to be initialized to 1, and throughout the computation the decremented value of n has to be assigned to an auxiliary variable k so that operations on n and with n (now k) can proceed in parallel. The process stops when n has been reduced to zero. The Petri net representation of the system, shown as Fig.4, is due to Mazurkiewicz (MA77).

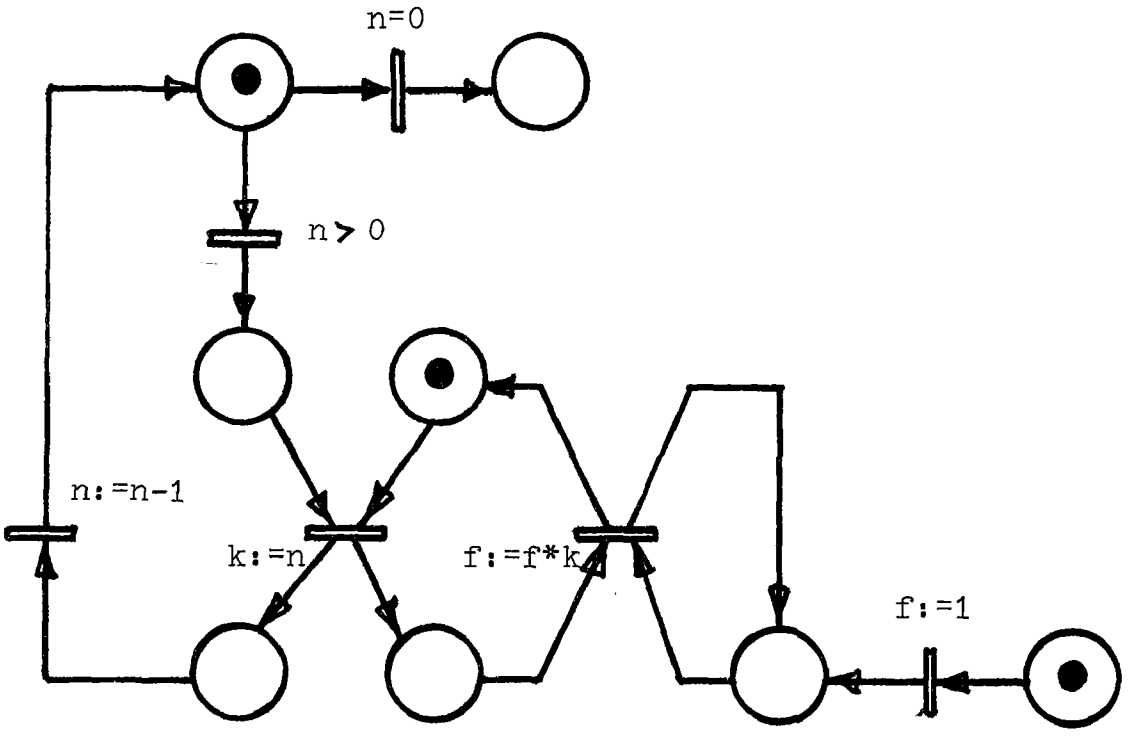

Fig.4

It is easy to see that there are two invariants:

$$n = M(1) + M(2) + M(3) + M(4) + M(6),$$
$$k = M(3) + M(7) + M(4) + M(5).$$

The first invariant corresponds to loops (1,2,3,1) and (1,4,6,1), the second to the loop (3,7,4,5,3). Of particular interest is the fact that the marking

$$M(2) = n,$$
$$M(7) = k$$

can be derived from M_0 by a sequence of firings. Under this marking the system is deadlocked. Techniques for showing that this deadlock marking is in fact attainable from M_0 are discussed by Memmi and Roucairol (MR80).

Peterson (PE77) provides a survey of Petri nets. A collection of papers on Petri nets and their generalization has recently appeared (BR80); of these the highly theoretical survey (JV80) discusses nets that are closest to the nets as defined here. (KE76) deals with the use of Petri nets in the verification of parallel programs. (LC75) contains an extensive discussion of solutions of the TS problem in terms of Petri nets. Andler's survey of synchronization primitives (AN79) is recommended reading in any case--it contains in particular a Petri net representation of the MB problem.

7. DEADLOCK AND STARVATION

We have discussed some examples of deadlock and starvation (livelock) informally, and a definition of these phenomena should now be given. For much of what follows we are indebted to (SH74), which contains an excellent chapter on deadlock.

A <u>system</u> is a pair $\langle S,P \rangle$, where S is a set of system states $\{s_1,s_2,\ldots,s_n\}$, and P is a set of processes $\{p_1,p_2,\ldots,p_k\}$. A <u>process</u> p_i is a function from S into $\mathcal{P}(S)$,

$$p_i: S \longrightarrow \mathcal{P}(S).$$

The null set is included in the range of the process: $p_i(s_u) = \emptyset$ means that p_i is in effect undefined for the state s_u.

If $s_v \in p_i(s_u)$, then p_i can change the state of the system from s_u to s_v by means of an operation. Let this be denoted by

$$s_u \xrightarrow{i} s_v.$$

Also,

$$s_u \xrightarrow{*} s_w$$

means that

(i) $s_u = s_w$, or

(ii) $s_u \xrightarrow{i} s_w$ for some p_i, or

(iii) $s_u \xrightarrow{i} s_v$ for some p_i and s_v, and $s_v \xrightarrow{*} s_w$.

A process p_i is blocked in state s_u if there exists no s_v such that $s_u \xrightarrow{i} s_v$. State s_u is a deadlock state if every process is blocked in this state. A process p_i is livelocked in state s_u if for all $s_v \in S$ such that $s_u \xrightarrow{*} s_v$, p_i is blocked in state s_v, i.e., a livelocked process remains blocked no matter what future changes the system goes through. State s_u is safe if for all s_v such that $s_u \xrightarrow{*} s_v$, s_v is not a deadlock state. By contrast, an unsafe state is one for which every sequence of operations (state changes) that begins at this state ends up in a deadlock state before the length of this sequence has exceeded the number of states in the system, i.e., the system, after it has entered an unsafe state, necessarily ends up deadlocked. There are states that are neither safe nor unsafe.

Refer now to Fig.1 (p.5). There we have a set of states represented by the right and upper edges of the diagram in which, according to our definition, one or other of the processes is blocked; the state represented by the upper right corner is a deadlock state. The definitions of deadlock and related phenomena have arisen from the study of operating systems which are regarded as nonterminating, and in which the processes too may be regarded as cyclic (going on forever).

Clearly, the state in which a terminating system has correctly terminated should not be regarded as a deadlock state; rather, it should be added to the set of safe states.

With this interpretation in force, all states in the region of Fig.1 above the upper limit and to the right of the righthand limit of the unsafe region are safe; all states within the unsafe region are unsafe. Consider the region that lies below the upper limit and to the left of the righthand limit of the unsafe region, but does not include the unsafe region itself. States in this region are neither safe nor unsafe. Of course, some regions of the diagram are empty of states: they represent impossible combinations of process states.

Consider now two processes that both require the same two resources, and are cyclic, i.e., they loop forever.

```
p1: loop                -- A      p2: loop                -- J
      request r1;       -- B            request r2;       -- K
      acquire r1;       -- C            acquire r2;       -- L
      request r2;       -- D            request r1;       -- M
      acquire r2;       -- E            acquire r1;       -- N
      release r2;       -- F            release r1;       -- O
      release r1;                       release r2;
    end loop;                         end loop;
```

We shall define the states for this system in terms of states of the individual processes, but in this we have to be very careful. There are two stages in p_1 when this process holds the single resource r_1 and no unsatisfied request has been made (at C and at F), and there are two analogous stages in p_2 relative to r_2 (at L and O). These two stages are fundamentally different, i.e., they must be distinguished as separate states. Define a system state as follows: the system is in state s_{ij} when p_1 is in state i and p_2 is in state j. Now suppose i is "p_1 holds r_1" and j is "p_2 holds r_2". Then four realizations of s_{ij} might exist:

(i) p_1 is at C, p_2 is at L;
(ii) p_1 is at C, p_2 is at O;
(iii) p_1 is at F, p_2 is at L;
(iv) p_1 is at F, p_2 is at O.

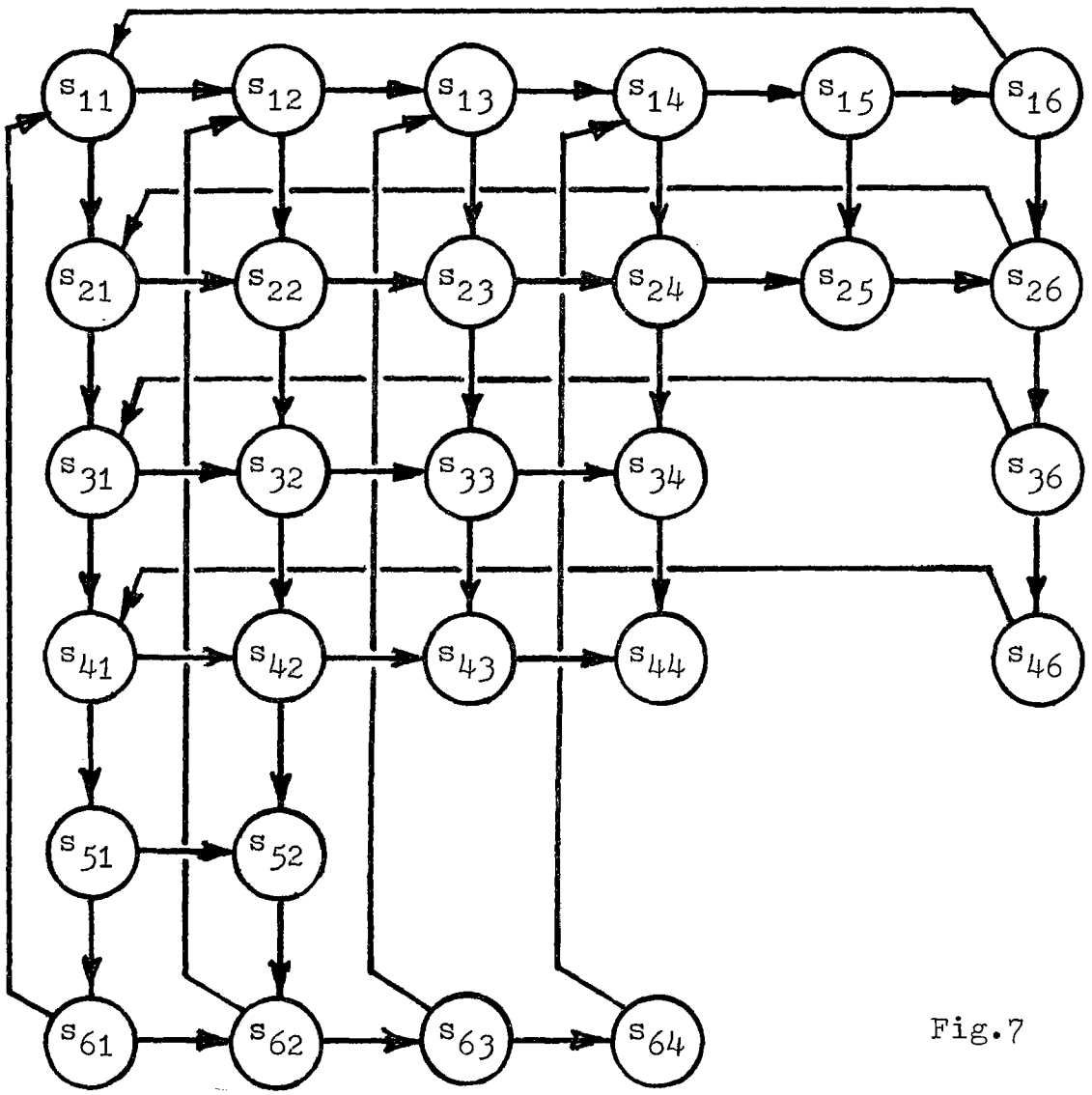

Fig.7

Note now that (iv) is an impossible situation, that (i) leads to deadlock, but that (ii) and (iii) are possible and need not result in deadlock. A clear distinction is made between these cases if the process states are defined as follows:

	states of p_1	states of p_2
1 -	process is at A	process is at J
2 -	process is at B	process is at K
3 -	process is at C	process is at L
4 -	process is at D	process is at M
5 -	process is at E	process is at N
6 -	process is at F	process is at O

Then the system has the representation shown in Fig.7.

Examination of Fig.7 shows that the system is deadlocked in state s_{44}; states s_{33}, s_{34}, and s_{43} are unsafe. Process p_1 is blocked in states s_{25}, s_{43}, s_{44}, and s_{46}; of these, states s_{25} and s_{46} are not unsafe. Because of symmetry, there exists an analogous set of states in which p_2 is blocked. There are no safe states in this system.

The objective of deadlock prevention is to make all states of a system safe. This we hope to achieve by means of synchronization mechanisms. However, in the general case, undecidability of the halting problem puts a theoretical limit on what can be achieved.

Deadlock can arise only when the following four conditions are all satisfied (CE71):

(a) resources cannot be shared;
(b) processes continue to hold already acquired resources while they are waiting for requested resources;
(c) resources may not be preempted while they are being held by a process (taken away from a process before the process is ready to release them);
(d) there exists a circular chain of processes such that each process holds resources that are being requested by the next process in the chain.

The very essence of many resources enforces (a) -- a terminal is impossible to share. Moreover, much of our synchronization effort is directed precisely at mutual exclusion. Condition (c) is also difficult to come to grips with. Preempting a printer would result in interleaving of output from different processes; the overhead of saving and restoring the state of a process at the time a resource is taken from it by preemption can be very high. This leaves conditions (b) and (d), and in practice one aims at preventing deadlock by assuming that one or other of these conditions is not satisfied. A survey of techniques for deadlock prevention can be found in (CE71).

Some systems contain so many processes and resources that deadlock prevention is an unrealistic goal. There one can only

hope to detect deadlock when it has arisen, and take the appropriate corrective action. An algorithm is presented in (KA80) that determines a priori whether or not starvation (and, hence, deadlock) is possible in a system of n processes and m types of serially reusable resources, and this with time complexity $O(mn^{1.5})$. Since most resources are serially reusable, this algorithm is quite general. Unfortunately, in the complex systems that we have in mind here--for example, data base applications--processes come and go, so that one does not know beforehand what processes there will be and what resources they will require. Detection of deadlocks in this latter dynamic setting is discussed in (BU79).

8. MONITORS AND RELATED CONSTRUCTS

Semaphores exhibit two features that make them inconsistent with modern programming practice. First, they are such a low level construct that semaphore solutions of synchronization problems of any complexity, particularly those in which different processes have different priorities with complicated rules governing process selection, become very difficult to understand. An example is provided by the writers-priority (RW1) and readers-priority (RW2) versions of RW (CH71). Second, and this is the more serious matter, they are totally at variance with modularity concerns. The P and V operations relating to the one semaphore may be scattered throughout a set of different process implementations--see, for example, the semaphore solution of the TS problem (p.16). This makes program verification very difficult.

In the early seventies two attempts were made more or less simultaneously to bring synchronization in line with modern programming practice. One approach was to modernize the existing concept of critical region by providing the critical region with a Boolean entry condition, and a queue in which processes would wait their turn if initially they were unable to satisfy the entry condition. This synchro-

nization mechanism is called a conditional critical region, and it localizes access to a shared object.

The other approach derives from Simula. For a Simula programmer a type of data objects consists of a representation of the object, and a set of operations meaningful for this representation. In Simula the specification of the representation and procedural definitions of the operations are collected in a class, which is a manifestation of a module. To this one now adds the synchronization conditions, i.e., synchronization becomes localized entirely within the type definition. A class generalized in this manner is called a monitor. There is implicit mutual exclusion in that the operations that are defined within the same monitor can operate only one at a time on a particular data object.

8.1 Conditional Critical Regions

Conditional critical regions (CCRs) were first proposed by Hoare (HO72), and developed by Brinch Hansen (BH72). The form of a CCR of res, a resource shared by several processes, is

region res when B do S

The shared resource is a variable, e.g., a record. Boolean condition B may be expressed in terms of components of res --its purpose is to guard the entry into S, where the latter is a statement (simple or compound), called the body of the CCR. Shared resources may be used only inside their CCRs.

To start off, a process enters its CCR and evaluates condition B. If B is true, the process executes S; otherwise the process enters a queue associated with res. This is a common queue for all processes put into a wait state on account of res. Whenever some process completes a CCR of res, the Boolean conditions are reevaluated for all waiting processes. A process whose condition is now true enters and completes execution of its CCR. Fairness criteria that prevent an eligible process from staying in the queue for long periods of time must be built into the queue scheduler.

Let us first look at a CCR solution of the MB problem. Here the shared variable is the message buffer (we shall call it exchange), which is a record of two components--the actual buffer mb, and an integer variable length (with initial value zero). Component mb is declared to be of type messagequeue, where it is assumed that the latter has been specified elsewhere as a queue of objects of type T. It is assumed further that the data type queue supports operations pop and push.

```
var exchange: shared record
                 mb: messagequeue;
                 length: integer;
              end;
    inmessage, outmessage: T;
```

```
producer:
loop
   produce inmessage;
   region exchange
   when length < capacity do
      begin
         push(mb,inmessage);
         length := length+1;
      end;
end loop;
```

```
consumer:
loop
   region exchange
   when length > 0 do
      begin
         outmessage := pop(mb);
         length := length-1;
      end;
   process message;
end loop;
```

A very interesting discussion of the CCR approach can be found in Latteux's survey (LA80). Consider the SP problem with a slight modification: If there remains just one empty cubicle, then a person coming into the facility (entering from the street) is given access to the cubicle only if there are available baskets as well. Now the shared record is

```
var check: shared record
              cub, bas: integer;
           end;
```

Each user of the facility is represented by the following process:

```
begin
   region check when cub< n-1 ∨
                     (bas< k ∧ cub< n) do
      cub := cub+1;
```

```
        change into swimwear;
        region check when bas < k do
           bas := bas+1;
        fill basket;
        region check when true do
           cub := cub-1;
        swim;
        region check when cub < n do
           cub := cub+1;
        empty the basket;
        region check when true do
           bas := bas-1;
        change into street clothes;
        region check when true do
           cub := cub-1;
    end;
```

Latteux uses the techniques developed by Owicki and Gries (OG76) to prove that the system cannot become deadlocked. He makes the interesting observation that the real difficulty of the proof is in the search for auxiliary variables and for invariants, which precedes the actual verification stage, and goes on to comment that since the first stage is entirely constructive (i.e., there is no proving going on), and the second stage could be automated, the risk of obtaining an erroneous proof is minimized. Here one can argue that the risk of an erroneous proof is indeed reduced, but that one may well be proving a "wrong" system in the sense that errors could well have been introduced during the construction of the invariants. See (AN81) for a general discussion of verification issues as they relate to parallel programs.

Evaluation of the Boolean conditions in the CCR solution of the MB problem introduce virtually no overhead, and the same can be said about the SP problem, but this is not so in general. The need for repeated evaluation of the Boolean conditions may make the CCR approach very costly in some instances. Ford (FO78) suggests modifications that would reduce the cost.

8.2 Monitors

The monitor was introduced by Dijkstra (DI72), who called it secretary, and developed by Brinch Hansen--see, e.g., (BH73) --and Hoare (HO74). Hoare also provided proof rules for monitors, which were developed further by Howard (HO76), who used them to study the monitor solution of the DH problem. Monitors have been incorporated into the programming languages Concurrent Pascal (BH75), Modula (WI77), and Mesa (MM79).

A monitor is an encapsulated set of procedures for operating on shared resources that are accessible to all processes of a system. Mutual exclusion is achieved by permitting only one monitor procedure to be running at any one time, but finer tuning within a monitor is made possible by low level synchronization primitives wait and send (in Modula terminology; Hoare introduced the primitives wait/signal, Concurrent Pascal has delay/continue, Mesa has wait/notify--all are similar, but have their points of difference).

Assume that process P is in control of a procedure. If the procedure contains the statement wait(s), then execution of the wait(s) causes P to be suspended at this point until it receives the signal s. If the procedure contains the statement send(s), then P is suspended, and a suspended process Q that may be waiting for signal s is activated. Process P is automatically resumed when Q has completed its execution of the monitor procedure. If no processes are waiting for the signal s, then send(s) has no effect. Actually Modula has a more general form of the wait statement that permits precedence classes of waiting processes to be established.

The wait/send primitives communicate between processes, i.e., the strict exclusion enforced by the monitor mechanism is somewhat relaxed as regards these primitives. Nevertheless, only one process is actually running at any one time within a monitor. Joseph (JO78) compares the use of monitors in Concurrent Pascal and Modula, and finds the wait/send facility of Modula superior to the delay/continue of Concurrent Pascal. The difficulty with comparisons is to find objective criteria for comparisons, but some work

towards arriving at such criteria is being attempted (BL79). One criterion that suggests itself is expressive power. However, any reasonable variant of the monitor construct is sufficient for the implementation of semaphores; conversely, a monitor can be implemented using semaphores. Hence monitors are equivalent in expressive power to semaphores, and different variants of the monitor are themselves equivalent.

Considerable experience has been gained with monitors. Brinch Hansen has used Concurrent Pascal to write the Solo operating system (BH76); the Pilot operating system has been written in Mesa--the experience gathered in this effort is documented in (LR80).

8.3 Examples of Monitors

Our examples will be expressed as actual Modula programs. Therefore, in contrast to earlier examples given above, more detail will appear in these programs. We begin with the MB problem. Note that in Modula a monitor is called an interface module, and that our buffer has size 10.

```
interface module message buffer;
   define put, get;
   var buffer: array 1:10 of datatype;
       inpos, outpos, count: integer;
       nonfull, nonempty: signal;
   procedure put(inmessage: datatype);
   begin
      if count = 10 then wait(nonfull) end;
      buffer[inpos] := inmessage;
      inpos := inpos mod 10 + 1;  count := count+1;
      send(nonempty)
   end put;
   procedure get(var outmessage: datatype);
   begin
      if count = 0 then wait(nonempty) end;
      outmessage := buffer[outpos];
      outpos := outpos mod 10 + 1;  count := count-1;
      send(nonfull)
```

```
      end get;
   begin
      inpos := 1;  outpos := 1;  count := 0
   end messagebuffer;
```

The actual processes are now simply

```
producer:                    consumer:
loop                         loop
   produce message;             get(message);
   put(message)                 process message
end;                         end;
```

Comparison of this monitor solution with the corresponding CCR solution (p.29) shows that a fair degree of separation of the synchronization component from the actual processes has been achieved, and this in a natural way.

Our second example is from (WI77); it is a monitor for the DH problem. Note the use made here of the system function awaited(s). This is a Boolean function that is true if there is at least one process waiting for signal s, and false otherwise. Note further that a second argument is used in procedure wait to establish priorities among waiting processes. This is the delay rank. If several processes are waiting for signal s, then s is received by the process with the smallest delay rank; if two processes with the same rank contend, signal s goes to the process that has waited longer. wait(s) is interpreted as wait(s,1).

```
interface module diskhead;
   define request, release;
   use cylmax;  (*no. of cylinders*)
   val headpos: integer;
       up, busy: Boolean;
       upsweep, downsweep: signal;
   procedure request(dest:integer);
   begin
      if busy then
         if headpos<dest then wait(upsweep,dest)
         else wait(downsweep,cylmax-dest) end
      end;
      busy := true;  headpos := dest
   end request;
```

```
procedure release;
begin
   busy := false;
   if up then
      if awaited(upsweep) then send(upsweep)
      else up := false;  send(downsweep)
      end
   else
      if awaited(downsweep) then send(downsweep)
      else  up := true;  send(upsweep)
      end
   end release;
begin
   headpos := 0;  up := true;  busy := false
end diskhead;
```

A comparison of this program with Hoare's solution (HO74) will show that there is very little difference between the wait/send and wait/signal formulations.

We remarked before on the separation of the synchronization component from the actual computations that take place once a process has been permitted to proceed. This is an important concern as regards the RW problem. Since any number of readers can read at the same time, but only one monitor procedure can be active, it would be a serious mistake to include actual read (or write) operations in the monitor. Let it be strongly emphasized that a monitor is to contain only synchronization procedures; the actual process is external to the monitor and only invokes the monitor procedures. A solution of the RW problem is to be found in (HO74).

A significant modification of the monitor concept has been proposed by Kessels (KE77). Counts are kept of active processes, i.e., those running or wishing to run. In the case of problem RW1 appropriate counters would be readers and writers. Conditions are specified at the head of the monitor in terms of the counters and other data. Wait statements can appear in any monitor procedure any number of times, and they are identified with the names of the conditions. For example,

wait thiscondition passes a process through if thiscondition is true, but makes it wait otherwise. This makes the concept similar to CCR, with which it shares the possibly very high cost of condition reevaluation at each completion of a monitor procedure. What complicates matters is that the same condition can be associated with more than one wait statement. Kessels' solution of RW1 follows

```
interface module readerswriters;
   define startread, endread, startwrite, endwrite;
   val readers, writers: integer;
       writerbusy: Boolean;
       readpermit: cond writers=0;
       writepermit: cond readers=0 and not writerbusy;
   procedure startread;
   begin
      wait readpermit;
      readers := readers+1
   end;
   procedure endread;
   begin
      readers := readers-1
   end;
   procedure startwrite;
   begin
      writers := writers+1;
      wait writepermit;
      writerbusy := true
   end;
   procedure endwrite;
   begin
      writerbusy := false;
      writers := writers-1
   end;
begin
   readers := 0;  writers := 0;  writerbusy := false
end readerswriters;
```

Each reader starts and ends its reading action with calls to startread and endread, respectively. Similarly for a writer.

8.4 Ada Tasks

The fact that the components of monitors are procedures is a significant cause of the complexity of the specification of a monitor. Consider the implementation of Boolean semaphores in Modula (WI77):

```
interface module resourcereservation
   define semaphore, P, V, init;
   type semaphore = record taken: Boolean;
                           free:  signal
                    end;
   procedure P (var S: semaphore);
   begin if S.taken then wait(S.free) end;
         S.taken := true
   end P;
   procedure V (var S: semaphore);
   begin S.taken := false;
         send(S.free)
   end V;
   procedure init (var S:semaphore);
   begin S.taken := false
   end init
end resourcereservation;
```

In Ada (US80) this same specification is achieved by the much simpler

```
task semaphore is
   entry P;
   entry V;
end;

task body semaphore is
begin loop accept P;
           accept V;
      end loop;
end;
```

A task (the Ada term for a process) specification is in two parts: the `task ... is` starts a declarative part that may be

physically separated from the actual task body, which starts with task body ... is. The statement "entry P;" declares P to be an entry, i.e., an object that can be invoked from other tasks by a call that has the same form as a procedure call. However, an entry has little resemblance to a procedure. Suppose we have task A that calls entry X in task B. Completion of this call involves the statement "accept X;" in the body of task B. The two tasks are proceeding side by side. Now, if an "accept X;" is reached in B before X has been called, task B remains suspended in a wait state at this statement; if a call to X is made before an "accept X;" has been reached in the execution of B, then task A is suspended to allow task B to catch up with it. In Ada terminology, the two tasks meet in a rendezvous. Now a measure of resemblance to a procedure may enter. An accept can take the form

```
accept X(parameters of X) do
   ... ... ...
   ... ... ...
end X;
```

If this were the form of our "accept X", then the statements between the do and the "end X" would have to be executed before task A could proceed on its way. In other words, tasks A and B are made to come together by means of a call and an accept, possibly task B then executes a particular piece of code, and then tasks A and B go their separate ways again.

Returning to task semaphore, note that to begin with the task has to be set running. This is done by means of the statement

```
initiate semaphore;
```

[Revised version provides implicit initiation

In a task that uses the task semaphore to achieve mutual exclusion, the critical section would be bracketed by calls to P and V, as follows:

```
P;
... ... ... -- critical section
V;
```

Note that more than one task can call an entry, and the called

task may then have difficulty generating the accepts quickly enough. When this happens, calling tasks are put into a queue associated with the appropriate entry.

Undoubtedly the Ada implementation of the semaphore is simpler than the corresponding Modula implementation. One could argue that this is achieved at the cost of new programming constructs (entry and accept), but it should be kept in mind that the wait and send of Modula are also new. A more serious argument in favor of monitors is that while the one instance of a monitor can be used with numerous semaphores, a separate task has to be created for each semaphore that we wish to use. The Ada response is to allow for creation of a family of tasks. Thus

```
task semaphore(1 .. 10) is
   entry P;
   entry V;
end;
```

is the declarative part of a task consisting of ten individual semaphores that can be initiated separately or en bloc. Reference to entry P of the fifth semaphore is now

```
semaphore(5).P
```

An Ada task that implements a solution of the MB problem is taken from (IC79). It illustrates parametrized entries, accepts that consist of some explicit processing, and the select statement. The latter is similar to the case statement and should be self-explanatory.

```
task mergebuffer is
   entry put(inmessage: in datatype);
   entry get(outmessage: out datatype);
end;
task body messagebuffer is
   buffer: array (1..10) of datatype;
   inpos, outpos: integer := 1;
   count: integer := 0;
begin
   loop
```

```
        select
           when count < 10 =>
              accept put(inmessage: in datatype) do
                 buffer(inpos) := inmessage;
              end;
              inpos := inpos mod 10 + 1;
        or    count := count + 1;
           when count > 0 =>
              accept get(outmessage: out datatype) do
                 outmessage := buffer(outpos);
              end;
              outpos := outpos mod 10 + 1;
              count := count - 1;
        end select;
     end loop;
  end messagebuffer;
```

Ada has numerous synchronization features not introduced here. Many more interesting examples of tasking can be found in (IC79). This is an important document in general, detailing the reasons for making the design decisions that finally resulted in the programming language Ada. A highly formal definition of parallelism in Ada is given by Mayoh (MA80); critiques of the synchronization features of Ada are provided by van den Bos (VB80), and Silberschatz (SI81).

9. PATH EXPRESSIONS AND CONTROL MODULES

Modularization of programs, and the localization of the synchronization component have two advantages. First, modification of a program in response to changes in synchronization rules is confined to well defined modules in the program. Second, program verification is made much easier. In contrast to semaphores, monitors provide localization of the synchronization component, thus making the effect of changes in the synchronization rules reasonably easy to cope with. However, the dependence on low level primitives within a monitor (wait/send) is detrimental to program validation. Rather unexpectedly, the Ada rendezvous mechanism goes very much

against the localization criterion. However, in justice to the Ada designers, it should be noted that the Ada design reflects anticipation of distributed processing. One would have some difficulty in providing distributed localization.

Deliberate attempts at trying to separate synchronization of processes from their specification have resulted in the development of path expressions and control modules. In both instances localization was an important concern, but so also was the provision of a sound theoretical base for the new mechanisms. Path expressions (CH74) are regular expressions, and are thus tied to formal languages and automata (HA75). They can also be described in terms of Petri nets (LC75). A precise statement of the semantics of path expressions has been provided by Berzins (BE77). An extensive study of various aspects of path expressions has been made by a group of researchers at the University of Newcastle upon Tyne. A bibliography of their work is given in (LB78); updates can be found in (SL80). Kotov (KO78) describes control types, which resemble path expressions. Two programming languages have been developed that make use of path expressions: COSY (LT79), and Path Pascal (KC80).

Verjus and his collaborators have introduced control modules based on system status counters (RV77), and have developed this concept further (BB78). A major concern in the design of this mechanism has been the ease of proofs of correctness. Proofs of correctness make use of invariants. Since the synchronization conditions in control modules are invariants, proofs are being made easier. Some extensions of the control module concept are described in (LA80), which also contains good examples of the use of control modules.

9.1 Path Expressions

Contrary to the more common synchronization schemes that set out to delay or prohibit processes from running, path expressions state what sequencing of processes is permissible.

A path expression is a regular expression from which all possible execution sequences can be derived (HA75). It

consists of procedure names, and, in precedence order, the operators * (Kleene star), ; (sequencer, which may be omitted when the meaning is clear), and + (exclusive alternative selector). Parentheses may be used to override normal precedence. A path expression is delimited as follows:

path path expression end

where path/end represent an implicit Kleene star. The operator ; denotes sequencing. Thus

path a; (b;c)*; d end

which may also be written as

path a(bc)*d end

states that execution of a may be followed by any number of executions of b followed by c, which is then followed by the execution of d, and that this entire execution sequence can be repeated again and again.

The operator + denotes exclusive selection from a set of alternatives, e.g.,

path a(b+c+d)e end

indicates that one and only one of the execution sequences abe, ace, and ade is permitted. However, in repetition one may switch between the sequences, e.g., it is possible to have abeabeaceabeade. The distribution law of ; over + lets us write a(b+c)d as (ab+ac)d or as abd+acd.

An expression followed by * may be executed zero or any number of nonzero times. Note that a* + b* stands for a sequence composed of all a's or of all b's, but that (a+b)* stands for a sequence in which any number of a's and b's may appear intermixed.

Path expressions may be modified by means of conditional elements. A conditional element in a path expression has the form [c1,c2,...,cn,celse], where each of c1,c2,...,cn has the form

condition: path expression,

but the optional celse is just a path expression. Priority may be introduced by means of symbols > and <, which have

the same precedence as operator +. For instance,

path f(g > h)k end

means that after f has been executed, either g or h can execute, but, if execution of both g and h is requested, g will be scheduled first.

It is possible to describe the desired behavior of a system by more than one path expression. Thus

path pr end
path qr end

specify that the execution of two p's must be separated by an r, and that the execution of two q's must also be separated by an r. Then two r's must be separated by both a p and q, but the order of the two does not matter (they may be even executed in parallel). To keep all the other properties of this system, but eliminate the possible execution of p and q in parallel, one has to combine the two path expressions given above into a single path expression of the form

path pr & qr end

Let us now write a path expression solution for the MB problem, as adapted from (HA75).

```
class message_buffer is
   array (1..10) of message;
   type slotstate is (empty,inuse, full);
   mesnum: integer:= 0;
   slotnum: integer:= 10;
   inpos, outpos: integer:= 0;
   state: array (1..10) of slotstate;
   path [mesnum >0:searchmes] + addmes &
        [slotnum >0:searchslot] + addslot end;
   function searchmes return integer;
      x: integer:= inpos+1;
      begin
         while state(x) /= full loop
            x:= x mod 10 + 1;
         end loop;
         state(x):= inuse;
         mesnum:= mesnum-1;
```

```
            inpos:= x;
            return inpos;
         end;
      function searchslot return integer;
         y: integer:= outpos+1;
         begin
            while state(y) /= empty loop
               y:= y mod 10 + 1;
            end loop;
            state(y):= inuse;
            slotnum:= slotnum-1;
            outpos:= y;
            return outpos;
         end;
      procedure addmes(k: in integer);
         begin
            mesnum:= mesnum+1;
            state(k):= full;
         end;
      procedure addslot(k: in integer);
         begin
            slotnum:= slotnum+1;
            state(k):= empty;
         end;
      procedure put(messagein: in message);
         y: integer:= searchslot;
         begin
            message_buffer(y):= messagein;
            addmes(y);
         end;
      procedure get(messageout: out message);
         x: integer:= searchmes;
         begin
            messageout:= message_buffer(x);
            addslot(x);
         end;
   end class;
```

Here the path expression makes sure that no put takes place when there are no empty slots, and no get when there are no

messages. It also specifies that the search and add operations cannot overlap to preserve the integrity of the state vector and variables slotnum and mesnum. However, the path does not require the execution sequences "searchslot; addmes" or "searchmes; addslot". Hence more than one sender or receiver can access the buffer, but only one at a time can search or add. This means that the program, although more complicated than other solutions we have seen for the MB problem, is also more general.

9.2 Control Modules

A control module maintains for each procedure P that it controls a set of five state counters:

- #requ(P): total number of requests for P since the control module was started up;
- #auth(P): total number of execution authorizations since the control module was started up;
- #term(P): total number of terminations of P since the control module was started up;
- #runs(P) = #auth(P) - #term(P), the number of instances of P currently running;
- #wait(P) = #requ(P) - #auth(P), the number of requests for P currently queued up.

The control module also has to maintain queues for the processes that are in wait states.

Control modules for three versions of the RW problem are provided in (RV77). We give here the two that correspond to our RW1 and RW2.

```
RW1: control module;
begin READ, WRITE: procedure;
   condition(READ) : #runs(WRITE) + #wait(WRITE) = 0;
   condition(WRITE): #runs(WRITE) + #runs(READ)  = 0;
end;
```

```
RW2: control module;
begin READ, WRITE procedure;
   condition(READ):  #runs(WRITE) = 0;
   condition(WRITE): #runs(WRITE) + #runs(READ)
                                  + #wait(READ) = 0;
end;
```

The advantage of this approach is that the synchronization module is completely removed from the actual procedures it synchronizes. Also, a change in priority rules, such as we have in going from RW1 to RW2 is implemented merely by changing conditions in the control module.

In (LA80) can be found a discussion of numerous extensions of the control module idea. To start, by associating a formal language with a control module, a methodology is developed for proving that a particular solution is free from deadlock, or showing that a solution does allow starvation. Consider a solution of the DP problem:

```
philosophers: control module;
begin M1, M2, M3, M4, M5 procedure;
   condition(M1): #runs(M5) +#runs(M2) = 0;
   condition(M2): #runs(M1) +#runs(M3) = 0;
   condition(M3): #runs(M2) +#runs(M4) = 0;
   condition(M4): #runs(M3) +#runs(M5) = 0;
   condition(M5): #runs(M4) +#runs(M1) = 0;
end;
```

This solution prevents deadlock, but starvation may result. This is demonstrated in (LA80).

In (BB78) control module implementations are given of binary and general semaphores. Hence any synchronization problem that can be solved with semaphores can be solved with control modules. Nevertheless, control modules do have their limitations of convenience, arising from the fact that the counters do not provide a sufficiently detailed history of the system. Only counts of events are available, not the sequence in which the events took place. Some finer distinctions can be made by the use of what Latteux (LA80) calls final counters, but we prefer to term conditional counters.

Let P and Q be two procedures, and let A and B be operations of the basic kind that relate to these procedures (requ, auth, term). Then #(A(P)/B(Q)) is a conditional counter that is incremented by 1 at each operation A(P), but reduced to zero at each operation B(Q). By means of conditional counters a rather complicated readers-writers control module can be specified:

```
READWRITE: control module;
   begin READ, WRITE: procedure;
      condition(READ):  #runs(WRITE)=0 ∧
         (#wait(WRITE)=0 ∨ #(requ(READ)/term(WRITE))=0);
      condition(WRITE): #runs(WRITE)+#runs(READ)=0 ∧
         (#wait(READ)=0 ∨ #(auth(READ)/auth(WRITE))> 0);
   end;
```

We leave it as an exercise to determine what priority rules this solution implements, and what the deadlock-starvation properties of this solution are.

10. SYNCHRONIZATION OF DISTRIBUTED PROCESSES

So far we have considered only systems that share memory, i.e., systems that communicate with each other by updating the contents of shared memory locations. Further, there is the implicit assumption of centralized control over the system. In a distributed network of processes there is some measure of autonomy of the individual processors, and they can communicate with each other only in a limited way by means of message passing. Suppose one were to use semaphores or counters in this setting. Then a copy of a semaphore or counter would have to be maintained at each site, and one would have to ensure that the same value is held in each copy. This would take too long for the semaphores or counters to be of much use as synchronization mechanisms. More fundamentally, just ensuring that the copies hold the same value is difficult. In other words, the maintenance of system integrity is more difficult for a distributed system. Another difference between centralized and distributed processing is

concern with system failure. The failure of a centralized system has to be accepted with greater or lesser grace. With distributed systems, on the other hand, one of the reasons for their development is precisely the desire to continue to have an operational system even when one or more of its component processors or message links are out of action. For a survey of distributed processing see (SD79).

Reference to early research on message passing can be found in (HU79), a major component of which is a critical discussion of some of this work (which started in the early 1970's). The approach that seems to have achieved greatest prominence is Hoare's Communicating Sequential Processes, and we shall be discussing this approach further down in some detail.

A most extreme attitude to programming by message passing is found in Hewitt's actor systems--see, e.g., (HE77). An actor is an object that behaves like a data structure or like a procedure, and interacts with other actors by sending messages. It may be dormant or active, and it is roused into the active state by means of a message that it receives. Messages sent to actors that behave like data structures correspond to requests for operations to be performed with or on the data structures; messages sent to actors that behave like procedures correspond to parameter lists passed to procedures. Synchronization in an actor system is by means of serializers (AH79), which represent an extension of the monitor concept. A process waiting in a monitor queue can be made to run only by an explicit signal to this process. In a serializer, the condition for a process to resume running must be explicitly stated when the process enters a queue, thus making a signal unnecessary. However, it remains to be seen what efficiency serializers can achieve, i.e., their practicability is still rather uncertain.

Other work of importance has related to the interpretation of coroutines as a special case of parallel processing, which has led Kahn and MacQueen (KM77) to propose a programming system for distributed processing based on the coroutine concept. Cook (C080) has extended Modula into a language for distributed programming. Data flow languages

provide a very radical departure from current practice, but they have not as yet found practical application. It seems certain that they will gain in importance as VLSI makes available the support architecture that they neen. A very brief introduction can be found as part of (BD79).

Other work going on is the PLITS project at Rochester (FE79,FN80), the development of a general mathematical model of computation by message passing (MI79), Brinch Hansen's development of his "distributed processes" (BH78), which has been extended by Mao and Yeh in their work on communication ports (MY80), and the introduction of a communication data type (CL80,CM80).

Three additional papers dealing with the nature of distributed computing need to be drawn attention to. The first is a discussion of time in the context of distributed systems by Lamport (LA78). The second shows that mutual exclusion can be created in a computer network with only 2*(N-1) messages between N system nodes (RA81). The third is an analysis of the DP problem in a distributed setting (LR81), and is recommended as an example of the essential changes a problem undergoes in changing from a centralized to a distributed system.

10.1 Communicating Sequential Processes

Hoare's language for CSP's (HO78) is exceedingly simple. The system on which it is based is assumed to consist of processes that do not share any variables, but communicate by means of input and output operations, expressed by primitives of the language. For each communication source and target processes have to be specified, and the messages that are being passed are strongly typed.

Instead of specifying the complete syntax, which can be found in (HO78), we take again the MB example, and will then explain the language features that are made use of in this example. There are three processes to consider: messagebuffer is the synchronization process, and it communicates with a producer process and a consumer process.

```
message buffer::
   buffer: (1..10) message;
   inpos, outpos, count: integer;
   inpos:= 1; outpos:= 1; count:= 0;
   *[ count<10; producer?buffer(inpos) →
        inpos:= inpos mod 10 + 1; count:= count+1
    [] count>0; consumer?more() →
        consumer!buffer(outpos);
        outpos:= outpos mod 10 + 1; count:= count-1
    ]
```

The communication component in CSP consists of input and output commands. In general an input command is written PB?a, and, if it appears in process PA, it expresses an input request of PA from PB. This is to result in the assignment of an input value to variable a, which is local to PA. The input command may be executed only when PB is ready to execute a corresponding output command PA!b, which is to export the value of variable b (local to PB) to process PA. The concept is similar to the Ada rendezvous, and similarly to the latter, can be used to synchronize PA and PB.

In our example producer?buffer(inpos) requests input from the producer, which is to contain the matching output command messagebuffer!inmessage. Communication with the consumer is more complicated: consumer?more() requests a signal from the consumer indicating that it is ready, and the actual transfer of outmessage is effected by consumer!buffer(outpos); the matching commands in the consumer process being messagebuffer!more() and messagebuffer?outmessage, respectively. The more() is a structured value with no components, i.e., it acts purely as a signal.

The construct [.....] is an alternative command, consisting in our case of two guarded commands, which are separated by the guard separator []. The text to the left of the arrow (→) is the guard; the text to the right is a command list. Note that guards may consist of Boolean conditions and input commands. In our example both guards have this form. A guarded command list can be executed only if the guard is passable, which happens when the Boolean conditions are true

and the input commands have awaited their corresponding output commands in the source processes. If more than one guard is passable, then precisely one of the guarded commands is executed; if no guard is passable the alternative command has no effect. Note, however, that when in the execution of a guard all Boolean conditions are found true, and an input command is reached that cannot be immediately executed, the guard is not regarded impassable. The evaluation of the guard simply has to be delayed. As soon as a guard is established as passable, the command list that it is guarding can be executed. Starvation arises when the evaluation of all potentially passable guards is delayed forever. The * preceding an alternative command denotes repetition, i.e., the alternative command is executed again and again as long as there are passable guards. If repetition or termination depends on input commands in the guards, then termination will take place when all processes named as input sources by the input guards have terminated, but starvation remains of course a possibility.

Critiques and suggested extensions of CSP's can be found in (KS79) and (BA80); formal semantics of CSP's are provided in (FH79), and a proof system is developed in (AF80).

11. CONCURRENCY IN DATA BASES

The dominant activity of data base processing is writing into files and reading from files. It would seem that the RW problems discussed before would therefore be very important in the data base setting. However, the problems associated with data bases are of a totally different order of magnitude. The RW solutions given above require a synchronization mechanism for each resource. Thus, in terms of a data base, either entire large files would be regarded as resources, in which case potential for parallelism would be reduced, or individual records would become the resources, which would result in an extremely primitive mechanism or in unacceptable overhead costs. Solutions have therefore been application specific.

Other quite distinct problems have been created by distribution of data bases over sites that may be geographically widely dispersed. Indeed, at present nearly all proposals for truly distributed processing that are practically motivated are related to data bases. What complicates matters here is redundancy: a copy of the same data item may have to be stored at more than one site for reasons of robustness and efficiency. A reader then accesses some one copy of the data item, but a writer has to update all stored copies.

Our discussion of concurrency in data bases will be rather brief. The best starting point for a course of readings on this topic is (BG79), which, although it deals primarily with concurrency control in distributed data bases, contains a useful list of references on concurrency control in a centralized setting as well.

Let us define some general terms. A transaction is a sequence of reader and writer operations on a data base. The read-set (write-set) of a transaction is the set of data items to which a reader (writer) has to gain access. Two transactions are in conflict if the write-set of one intersects the read-set or write-set of the other.

Consider a set of transactions, each of which is a sequence of reader and writer operations. The total activity can be represented by a log. If no transactions execute concurrently, then the log is just a concatenation of sequences of the individual transactions, and the log is called serial. Suppose now that each transaction if performed on its own leaves the data base in a "correct" state, i.e., a state in which all the data satisfy the integrity constraints associated with the data base. Since each transaction maps the data base from a correct state to a correct state, a sequence of such transactions must also produce a correct state in the end. However, at intermediate stages in the execution of a transaction, the state of the data base need not be correct. This means that the data base may not be in a correct state after completion of a set of transactions that are executed concurrently. With concurrency of transactions, the operation sequences that define the individual transactions are inter-

leaved in the log. However, if the entries in such a nonserial log NS can be permuted to produce a serial log S, and the effects of NS and S on a given data base are identical, then NS is said to be serializable. Unfortunately Papadimitriou (PA78) has shown that the determination of whether or not a log is serializable is NP-complete, but he also discusses useful subclasses of the set of serializable logs.

Two-phase locking (EG76) is a mechanism that guarantees serializability of a nonserial log. Locking is aimed at detecting and eliminating adverse effects of conflicts. Two-phase locking is applicable where it is possible to divide a transaction into distinct growing and shrinking phases. All locks must be requested in the first of these phases, and they must all be released in the second phase. This is equivalent to requiring that no lock be requested after it has been released, and it is this rule that guarantees serializability. Deadlock (or starvation) may exist, but in the data base setting it would be too expensive to try to prevent it from arising. Instead, when deadlock is detected (and a detection procedure must be provided), one of the transactions that contribute to it is backed up and restarted. This may lead to cyclic restart in which the transaction keeps getting into a blocked state, is backed up and restarted, only to get into the blocked state yet again. See (BG79) for references to techniques for avoiding cyclic restart.

12. DATA ABSTRACTION AND SYNCHRONIZATION

In the preceding section we saw that the RW problem can become very complicated in the particular context of bata base processing. Complications also arise when synchronization is to be combined with data abstraction. To cope with this, Raynal (RA78) suggests the creation of three libraries, comprising (i) specifications of data types, e.g., sets, graphs; (ii) representations of the abstract types; (iii) access disciplines or synchronization modes, containing reader-writer, producer-consumer, and some very few additional modes. our experience suggests the opposite: there is little loss of

convenience in limiting abstract data types to a small number of standard types, but synchronization disciplines should not be so restricted.

What is particularly important is that very often the need for synchronization arises in a setting in which no elaborate mechanisms are necessary. We contend that where synchronization can be achieved by means of minor extensions of existing programming languages, this simple approach should be taken.

Consider the merging of two files that are objects belonging to an abstract data type. We have to traverse both files, but an advance is made only in one file at a time, and the traversals have to be synchronized to this effect. In (BE80) we argue that the merge operation should not be associated with a single data type. Further, iterators, which are provided by some modern programming languages, should belong to data types, but the synchronization of iterators then becomes a problem. Our solution is controlled iteration.

12.1 Controlled Iteration

The convenience that convetional loop variables and subscripts provide for array computations is fully appreciated only after one has worked with generalized iterators. In a conventional for loop the current value of the loop parameter can be used as a subscript value for access into any number of arrays not only directly, but also as modified by some suitable function. In other words, several arrays can be accessed in different orders in the same loop, as for example in

```
for I in A'FIRST..A'Last loop
   C(I):= A(I/2) + B(A'LAST-I+1);
end loop;
```

However, if a single for loop drives several iterators, then very much the same effect can be achieved without too great a sacrifice of the basic simplicity of the for loop. We can thus regard traversals as separate processes that can proceed at their own rates, i.e., achieve the effect of coroutines,

but retain control over their synchronization in a for loop.

Our generalized for loop takes the form

```
for T1, T2, ..., Tn loop
   loop body;
end loop;
   loop tail;
end for;
```

Here each iteration clause Ti is a string

[controlled] var [(attributes)] in Ii [:done]

where components enclosed in brackets [] are optional. The Ii is an iteration sequence, abbreviated IS, and elements of IS are bound to loop parameter var in turn. The Ii may be a conventional IS, such as the Ada forms

```
      K .. L
reverse K .. L
```

or Ii may be a generalized IS that follows the CLU model. For example, the generalized iteration sequence Y.tra delivers elements of structure Y in the order defined by tra, which is an iterator belonging to the data type of which Y is an instance. If a loop parameter is preceded by controlled, then, on reaching the end of the loop body, an advance is made to the next element of its IS only if an authorization to this effect has been issued in the body of the loop in the current pass through it. The authorization for

```
controlled x in Y.tra
```

is

```
promote x;
```

The items that make up the generalized IS may be composite, and (attributes) indicates which of the components of the item are made accessible as attributes of the loop parameter. Optional completion flag done is FALSE while the IS has not yet been exhausted; it changes to TRUE on reaching the end of the loop body if (i) the loop parameter is currently bound to the last element of the IS, and (ii) (for a controlled IS) a promote relating to this IS has been executed in the pass through the loop body that is just being completed.

Let us consider the file merge as our example. Files A and B are composed of records. Iterators trax and tray deliver records of A and B, respectively, in ascending order of their keys.

```
for controlled x(key:REAL) in A.trax: donex,
    controlled y(key:REAL) in B.tray: doney loop
    if donex then
       transfer y to output file;
       promote y;
    elsif doney then
       transfer x to output file;
       promote x;
    elsif x.key <= y.key then
       transfer x to output file;
       promote x;
    else
       transfer y to output file;
       promote y;
    end if; end loop;
end for;
```

Some remarks on the generalized for loop:

1. The type of the loop parameter in a conventional iteration clause is the type of the elements in its IS. For a generalized IS it is the type of the elements of the structure that is being traversed.

2. A conventional IS may be controlled; a generalized IS need not be.

3. Because different iteration sequences may become exhausted at different times, a loop parameter continues to have as value the final element of the IS even after this IS has become exhausted. The loop parameter associated with an empty IS is undefined, but then the termination flag is initiated to TRUE.

4. In each pass through the loop at most one advance is made in a controlled IS.

5. Exit is made from the loop body when all iteration sequences have become exhausted, or by means of an explicit exit. Exit is to the first statement following end loop. The scope of loop parameters and completions flags extends to end for.

Just how much this generalization of the for loop will cost is determined by the nature of the iterators. If they are simple coroutines, a stack suffices for the control: one simply pushes down a vector of activation records, one record for each of the T1, T2, ..., Tn. The implementation will tend to be distinctly more expensive if the iterators are recursive or invoke other iterators.

Proof of termination of the loop consists of two parts. First it has to be shown that the iterator generates a finite IS, and then that the IS does in fact become exhausted. The latter can be shown by demonstrating that an advance is made in each pass through the loop in some iterator, or, if this is not the case, by showing that the number of successsive passes through the loop between two passes in which advances are made is bounded by a constant. Hence, as regards termination, the generalized for loop occupies a level intermediate between the while loop and the conventional for loop.

12.2 Controlled Iteration and Concurrency

There are three opportunities for introducing a modest measure of parallelism in the context of controlled iteration. First, because traversals of data structures are carried out only in the context of for loops, buffering is possible, i.e., the iterator may begin looking for the next element in a traversal sequence as soon as it has delivered the current one. Second, under controlled iteration, the body of the loop takes quite naturally the form of an if statement. Now, if this is written in the form of Dijkstra's guarded command set, and the interpretation of the latter extended to permit execution in parallel of all commands whose guards are true, a further opportunity for concurrency arises.

We shall discuss in more detail the third opportunity: an iteration sequence may at times be partitioned into components to which the process defined by the body of the for loop can be applied independently one from the other, i.e., the components may be processed in parallel.

Consider matrix multiplication: matrix C is to receive the product of matrices A and B, and it is assumed that C has already been initialized to zeros. In conventional Ada syntax this can be written as follows:

```
for I in A'FIRST..A'LAST loop
   for K in A'FIRST(2)..A'LAST(2) loop
      for J in B'FIRST(2)..B'LAST(2) loop
         C(I,J):= C(I,J) + A(I,K)*B(K,J);
      end loop;
   end loop;
end loop;
```

Here matrix C is built up one row at a time. In building up a row in C, the corresponding row in A is traversed just once, but B is traversed in its entirety. What matters is that in generating a particular row of C only the one corresponding row of A is needed, i.e., the traversal of A can be partitioned into independent traversals of its rows.

Consequently we now consider a matrix as an array, but also as a set of vectors (its rows). We rewrite the matrix multiplication code in such a way that the separation of the matrix into a set of vectors can be used to induce parallelism in execution. It is the declaration of the data structure as a set before the for loop is entered that enables the system to recognize the opportunity for concurrency. The loop itself contains no indication to this effect.

```
for X in ROWSET.TRA loop
    I:= C'FIRST(2);
    for controlled A in X.FORWARD,
        B in MATB.ROWWISE(ENDROW) loop
        C(X.ROWNO,I):= C(X.ROWNO,I) + A.VAL*B.VAL;
        I:= I+1;
        if ENDROW then
           promote A;
           I:= C'FIRST(2);
        end if;
    end loop; end for;
end loop; end for;
```

Here we have three iterators: (i) Iterator TRA delivers a complete row of a matrix. It is understood that the object denoted by ROWSETA functions both as a set of vectors and as an array. Its guise as set permits parallelism. The body of the outer loop can be executed concurrently by as many processors as there are rows in the array. An attribute of the row delivered by TRA is the index of this row in reference to the matrix as a two-dimensional array (ROWNO). This attribute is essential to establish proper correspondence between the rows of the input matrix and the result matrix C. (ii) FORWARD delivers the elements of the matrix row supplied by TRA. (iii) ROWWISE is associated with the second input matrix in its guise as a proper matrix (MATB). It delivers elements of MATB one by one in roworder. Parameter ENDROW associated with ROWWISE is normally false, but it becomes true for any pass through the loop in which an element that terminates a row in MATB is being accessed.

One problem is the synchronization of components of several partitioned iteration sequences. Such is the case when matrices A and C are both regarded as sets of row vectors. Then it has to be ensured that the row in C generated using a particular row in A properly corresponds to this row in A (for example, that the row generated using the second row of A becomes in fact the second row of C). Iterators are used to enforce the required correspondence. Each iterator over the set (TRA in both cases in the example below) has to define a sequence of the elements of the set, and corresponding elements from the sequences are assigned to the same instance of execution of the loop body.

Matrix multiplication with two partitioned iteration sequences: Iterator WRAPAROUND delivers elements of vector Y in the infinite sequence Y(FIRST),...,Y(LAST),Y(FIRST), ...,Y(LAST),Y(FIRST),... . Because the sequence is infinite, a halter has to be used to effect termination of the loop. The concurrent interpretation of the conditional is in force here.

```
for X in ROWSETA.TRA,
    Y in ROWSETC.TRA loop
    for controlled A in X.FORWARD,
        B in MATB.ROWWISE(ENDROW): ENDB,
        C in Y.WRAPAROUND loop
        cond
           ENDB: exit []
           not ENDB: C.VAL:= C.VAL + A.VAL*B.VAL []
           ENDROW: promote A []
        end cond;
    end loop; end for;
end loop; end for;
```

Clearly the device described above does not solve general concurrency problems. Nevertheless, as regards the execution of such programs as are currently being executed on very small computers, use of our mechanism can lead to substantial reduction in execution time by enabling the computational load to be spread over several processors.

REFERENCES

AF80 Apt,K.R., Francez,N., and deRoever,W.P. A Proof System for Communicating Sequential Systems. ACM Trans. Prog. Lang. Syst. 2, 3 (July 1980), 359-385.

AH79 Hewitt,C.E., and Atkinson,R.R. Specification and Proof Techniques for Serializers. IEEE Trans. Software Eng. SE-5 (1979), 10-23.

AN79 Andler,S. Synchronization Primitives and the Verification of Concurrent Programs. In Operating Systems: Theory and Practice (D.Lanciaux,ed.), North-Holland, 1979, 67-99. (Contains annotated bibliography.)

AN81 Andrews,G.R. Parallel Programs: Proofs, Principles, and Practice. CACM 24, 3 (March 1981), 140-146.

BA80 Bernstein, A. Output Guards and Nondeterminism in "Communicating Sequential Processes". ACM Trans. Prog. Lang. Syst. 2, 2 (April 1980), 234-238.

BB78 Bekkers,Y., Briat,J., and Verjus,J.P. Construction of a Synchronization Scheme by Independent Definition of Parallelism. In Constructing Quality Software (P.G. Hibbard and S.A.Schuman,eds.), North-Holland, 1978, 193-205 (see also discussion: 233-235).

BD79 Bryant,R.E., and Dennis,J.B. Concurrent Programming. In Research Directions in Software Technology (P.Wegner, ed.), MIT Press, 1979, 584-610.

BE77 Berzins,V. Denotational and Axiomatic Definitions for Path Expressions. Comp. Structures Group Memo 153-1, Lab. for Computer Science, MIT, Nov. 1977.

BE80 Berztiss,A.T. Data Abstraction, Controlled Iteration, and Communicating Processes. Proc. ACM Annual Conf., Nashville TN, 1980, 197-203.

BG79 Bernstein,P.A., and Goodman,N. Approaches to Concurrency Control in Distributed Data Base Systems. Proc. AFIPS vol.48 (NCC 1979), 813-820.

BH72 Brinch Hansen,P. A Comparison of Two Synchronization Concepts. Acta Informatica 1 (1972), 190-192.

BH73 Brinch Hansen,P. Operating Systems Principles. Prentice-Hall, 1973.

BH75 Brinch Hansen,P. The Programming Language Concurrent Pascal. IEEE Trans. Software Eng. SE-1 (1975), 199-207.

BH76 Brinch Hansen,P. The Solo Operating System: a Concurrent Pascal Program. Software--Practice Experience 6 (1976), 141-150.

BH78 Brinch Hansen,P. Distributed Processes: A Concurrent Programming Concept. CACM 21, 11 (Nov.1978), 934-941.

BL79 Bloom,T. Evaluating Synchronization Mechanisms. Proc. 7th Symp. Op. Sys. Principles, 1979, 24-32.

BR80 Brauer,W.,ed. Net Theory and Applications. Springer Lecture Notes in Computer Science No.84, 1980.

BU79 Bittmann,P., and Unterhauer,K. Models and Algorithms for Deadlock Detection. In Operating Systems: Theory and Practice (D.Lanciaux,ed.), North-Holland, 1979, 101-111.

CE71 Coffman,E.G., Elphick,M.J., and Shoshani,A. System Deadlocks. Computing Surveys 3, 2 (June 1971), 67-78.

CH71 Courtois,P.J., Heymans,F., and Parnas,D.L. Concurrent Control with "Readers" and "writers". CACM 14, 10 (Oct.1971), 667-668.

CH74 Campbell,R.H., and Habermann,A.N. The Specification of Process Synchronization by Path Expressions. In Springer Lecture Notes in Computer Science No.16, 89-102.

CL80 Cunha,P.R.F., Lucena,C.J., and Maibaum,T.S.E. On the Design and Specification of Message Oriented Programs. Int. J. Comp. Inf. Sciences 9 (1980), 161-191.

CM80 Cunha,P.R.F., and Maibaum,T.S.E. A Communication Data Type for Message Oriented Programming. Proc 4e Coll. Internat. sur la Programmations (Springer Lecture Notes in Computer Science No.83), 1980, 79-91.

CO80 Cook,R.P. *MOD--A Language for Distributed Programming. IEEE Trans. Software Eng. SE-6 (1980), 563-571.

DI68 Dijkstra,E.W. Cooperating Sequential Processes. In Programming Languages (F.Genuys,ed.), Academic Press, 1968, 43-112.

DI72 Dijkstra,E.W. Hierarchical Ordering of Sequential Processes. In Operating System Techniques (C.A.R.Hoare and R.H.Perrot,eds.), Academic Press, 1972, 72-93.

DL78 Dijkstra,E.W., Lamport,L., Martin,A.J., Scholten,C.S., and Steffens,E.M.F. On-the-Fly Garbage Collection: An Exercise in Cooperation, CACM 21, 11 (Nov.1978), 966-975.

EG76 Eswaran,K.P., Gray,J.N., Lorie,R.A., and Traiger,I.L. The Notions of Consistency and Predicate Locks in a Database System. CACM 19, 11 (Nov.1976), 624-633.

FE79 Feldman,J.A. High Level Programming for Distributed Computing. CACM 22, 6 (June 1979), 353-368.

FH79 Francez,N., Hoare,C.A.R., Lehmann,D.J., and deRoever, W.P. Semantics of Nondeterminism, Concurrency, and Communication. J. Comp. Syst. Sciences 19 (1979), 290-308.

FN80 Feldman,J.A., and Nigam,A. A Model and Proof Technique for Message-Based Systems. SIAM J. Comput. 9 (1980), 768-784.

FO78 Ford,W.S. Implementation of a Generalized Critical Region Construct. IEEE Trans. Software Eng. SE-4 (1978), 449-455.

GK78 Goos,G., and Kastens,U. Programming Languages and the Design of Modular Programs. In Constructing Quality Software (P.G.Hibbard and S.A.Schuman,eds.), North-Holland, 1978, 153-186 (see also the discussion following this paper, 187-191).

HA75 Habermann,A.N. Path Expressions. Dept. of Comp. Sc., Carnegie-Mellon University, June 1975.

HA76 Habermann,A.N. Introduction to Operating System Design. SRA, 1976.

HE77 Hewitt,C. Viewing Control Structures as Patterns of Passing Messages. Artificial Intelligence 8 (1977), 323-364.

HO72 Hoare,C.A.R. Towards a Theory of Parallel Programming. In Operating Systems Techniques (C.A.R.Hoare and R.H.Perrot,eds.), Academic Press, 1972, 61-72.

HO74 Hoare,C.A.R. Monitors: An Operating Systems Structuring Concept. CACM 17, 10 (Oct.1974), 549-557.

HO76 Howard,J.H. Proving Monitors. CACM 19,5 (May 1976), 273-279.

HO78 Hoare,C.A.R. Communicating Sequential Processes. CACM 21, 8 (Aug.1978), 666-677.

HU79 Hunt,J.G. Messages in Typed Languages. ACM SIGPLAN Notices 14, 1 (Jan.1979), 27-45.

IC79 Ichbiah,J.D., et al. Rationale for the Design of the Ada Programming Language. ACM SIGPLAN Notices 14, 6 (June 1979), Part B.

JO79 Joseph,M. Towards More General Implementations Languages for Operating Systems. In Operating Systems: Theory and Practice (D.Lanciaux,ed.), North-Holland, 1979, 321-331.

JV80 Jantzen,M., and Valk,R., Formal Properties of Place/Transition Nets. In (BR80), 165-212.

KA80 Kameda,T. Testing Deadlock-Freedom in Computer Systems. JACM 27, 2 (April 1980), 270-280.

KC80 Kolstad,R.B., and Campbell,R.H. Path Pascal User Manual. ACM SIGPLAN Notices 15, 9 (Nov.1980), 15-24.

KE76 Keller,R.M. Formal Verification of Parallel Programs. CACM 19, 7 (July 1976), 371-384.

KE77 Kessels,J.L.W. An Alternative to Event Queues for Synchronization in Monitors. CACM 20, 7 (July 1977), 500-503.

KM77 Kahn,G., and MacQueen,D. Coroutines and Networks of Parallel Programs. Proc. IFIP Congress 77 (1977), 993-998.

KO78 Kotov,V.E. Concurrent Programming with Control Types. In Constructing Quality Software (P.G.Hibbard and S.A.Schuman,eds.), North-Holland, 1978, 207-228 (see also discussion 233-235).

KS79 Kieburtz,R.B., and Silberschatz,A. Comments on "Communicating Sequential Processes". ACM Trans. Prog. Lang. Syst. 1, 2 (Oct.1979), 218-225.

LA78 Lamport,L. Time, Clocks, and the Ordering of Events in a Distributed System. CACM 21, 7 (July 1978), 557-565.

LA80 Latteux,M. Synchronisation de Processus. R.A.I.R.O. Informatique/Computer Science 14, 2 (1980), 103-135.

LB78 Lauer,P.E., Best,E., and Shields,M.W. On the Problem of Achieving Adequacy of Concurrent Programs. In Formal Descriptions of Programming Concepts (E.J.Neuhold, ed.), North-Holland, 1978, 301-334.

LC75 Lauer,P.E., and Campbell,R.H. Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes. Acta Informatica 5 (1975), 297-332.

LR80 Lampson,B.W., and Redell,D.D. Experience with Processes and Monitors in Mesa. CACM 23, 2 (Feb.1980), 105-117.

LR81 Lehmann,D., and Rabin,M.O. On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dinig Philosophers Problem. Conf. Record 8th Ann. ACM Symp. Princ. Prog. Lang., 1981, 133-138.

LT79 Lauer,P.E., Torrigiani,P.R., and Shields,M.W. COSY--A System Specification Language Based on Paths and Processes. Acta Informatica 12 (1979), 109-158.

MA77 Mazurkiewicz,A. Concurrent Program Schemes and Their Interpretation. DAIMI PB-78, Dept. of Comp.Sc.,Aarhus University, July 1977.

MA80 Mayoh,B.H. Parallelism in Ada: Program Design and Meaning. Proc 4e Coll. Internat. sur la Programmation (Springer Lecture Notes in Computer Science No.83), 1980, 256-268.

MI79 Milne,G., and Milner,R. Concurrent Processes and their Syntax. JACM 26, 2 (April 1979), 302-321.

MM79 Mitchell,J.G., Maybury,W., and Sweet,R. Mesa Language Manual. Xerox PARC, 1979.

MR80 Memmi,G., and Roucairol,G.P. Linear Algebra in Net Theory. In (BR80), 213-223.

MY80 Mao,T.W., and Yeh,R.T. Communication Port: A Language Concept for Concurrent Programming. IEEE Trans. Software Eng SE-6 (1980), 194-204.

OG76 Owicki,S., and Gries,D. An Axiomatic Proof Technique for Parallel Programs. Acta Informatica 6 (1976), 319-340.

OW76 Owicki,S. A Consistent and Complete Deductive System for the Verification of Parallel Programs. Proc. 8th ACM SIGACT Symp. Theory Comp., 1976, 73-86.

PA71 Patil,S.S. Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordinations amongst Processes. Computer Structures Group Memo 57, MIT Project MAC, Feb. 1971.

PA72 Parnas,D.L. On the criteria to be used in decomposing systems into modules. CACM 15, 12 (Dec.1972), 1053-1058.

PA75 Parnas,D.L. On the Solution to the Cigarette Smokers Problem (without Conditional Statements). CACM 18, 3 (March 1975), 181-183.

PA79 Papadimitriou,C.H. The Serializability of Concurrent Database Updates. JACM 26, 4 (Oct.1979), 631-653.

PE77 Peterson,J.L. Petri Nets. Computing Surveys 9, 3 (Sept.1977), 223-252.

RA78 Raynal,M. Une Expression de la Synchronisation pour les Types Abstraits. R.A.I.R.O. Informatique/Computer Science 12, 4 (1978), 307-316.

RA81 Ricart,G., and Agrawala,A.K. An Optimal Algorithm for Mutual Exclusion in Computer Networks. CACM 24, 1 (Jan.1981), 9-17.

RO78 Roucairol,G.P. Mots de synchronisation. R.A.I.R.O. Informatique/Computer Science 12, 4 (1978), 277-290.

RV77 Robert,P., and Verjus,J.P. Toward Autonomous Descriptions of Synchronization Modules. Proc.IFIP Congress 77 (1977), 981-986.

SD79 Stankovic,J., and vanDam,A. Research Directions in (Cooperative) Distributed Processing. In Research Directions in Software Technology (P.Wegner,ed.), MIT Press, 1980, 611-638.

SH74 Shaw,A.C. The Logical Design of Operating Systems. Prentice-Hall, 1974.

SI81 Silberschatz,A. On the Synchronization Mechanism of the Ada Language. ACM SIGPLAN Notices 16, 2 (Feb.1981), 96-103.

SL80 Shields,M.W., and Lauer,P.E. Verifying Concurrent System Specification in COSY. In Springer Lecture Notes in Computer Science No.88, 576-586.

US80 U.S.Dept. of Defense. Reference Manual for the Ada Programming Language. July 1980.

VB80 van den Bos,J. Comment on Ada Process Communication. ACM SIGPLAN Notices 15, 6 (June 1980), 77-81.

WI77 Wirth,N. Modula: A Language for Modular Multiprogramming. Software--Practice Experience 7 (1977), 3-35.