APPENDIX A

ALGORITHMS FOR CONCURRENT PROGRAMMING CONTROL

A.1. Introduction

It is quite difficult to realize exclusions between two or more asynchronous concurrent processes without using special primitive operations, like LOCK and UNLOCK.
In 1962 the Dutch mathematician Dekker found an algorithm which solved the problem for two processes. Dijkstra found a solution for N processes, a few years later. **(Dijkstra '65, '68).**
A series of improvements and alternative solutions have been published since then. **(Knuth '68; deBruyn '67; Eisenberg & McGuire '72; Lamport '74; Lamport '76).**
We will first give a concise problem statement, then present one of the more recent solutions **(Lamport '74; '76),** and finally discuss some alternative strategies for concurrent programming control.

A.2. Problem statement

- N asynchronous concurrent processes share a data base in which they can execute read and write operations.
- Each process executes a cyclic program. The shared data base is only accessed in one specific part of that program, which is called the *critical section*.
- Only one of the N processes may be executing in its critical section at a time. To realize this exclusion the processes must be synchronized whenever they attempt to enter their critical sections.
- To establish an exclusion one may only assume the indivisibility of read and write operations per memory location. It is assumed that all data entities are stored in single words. (Multiple-length arithmetic would present extra problems.)
- It must be impossible to devise such finite speeds for the processes that the decision of which of them will enter its critical section first is postponed indefinitely. It is assumed that each process has an average progress which is nonzero.
- The processes are allowed to halt in their non-critical sections, but this halting may not cause the blocking of other processes via the synchronization mechanisms.
- Access to the shared data base must be acquired in less than N turns, at all times.
- Ideally, no process may be favored systematically at all, in the arbitration.

We can interpret this requirement as follows: if M processes try to enter their critical sections simultaneously, each of these processes must have equal chances of gaining access as the first, the second, ..., or, the M-th process from that batch[1].

— — — — —

(1) Another, less stringent, interpretation of this requirement is given in section A.4 **(Alternatives).**

The last two requirements can be called, respectively, a *fairness* and a *symmetry requirement*. The fairness requirement was added to the problem by **Eisenberg & McGuire '72**. The symmetry requirement is a further sophistication, which was not discussed until now.

> The fairness requirement states that no process may occupy its critical section more than once, in consecutive cycles, while other processes are waiting to enter their critical sections.
> The symmetry requirement is not as such 'stronger' than the fairness requirement. The symmetry requirement relates to the arbitration policy for process requests which have arrived simultaneously in the arbitration algorithm. (The relative times of arrival can be deduced from the values of state variables in most algorithms.) Note that an algorithm can be fair and not symmetric, but also symmetric and not fair.

To realize exclusions with algorithms as discussed here is a rather inefficient method. We may expect that the more additional requirements we pose to these algorithms, the more inefficient they will still be. Furthermore, the obeyance of a symmetry, or even a fairness requirement, will in most applications be superfluous anyway[1]. To study the problem in this context is, however, still interesting from a theoretical point of view. We may ask the question whether algorithms which obey all requirements listed can be constructed at all; we may study how we can derive such algorithms in a systematic way, how one can prove properties of them, how they can be optimized, etc. Further, it is not unthinkable that some of the algorithms which are developed in this line of research can be used in specific practical applications. (No special primitives available on short notice, long delays, high waiting probabilities, etc.)

In the following we will consider one of the more recent algorithms (the one given by Lamport in 1974), and discuss some alterations to it.

A.3. Lamport's solution

Lamport used two integer arrays, called array CHOOSING and array NUMBER. Both arrays have N elements, which are to be initialized to zero. The process with identity "i" must execute the program given below.
The function MAXIMUM selects the largest element from array NUMBER. We may assume that this is done by the consecutive testing of each element in the array, which implies that the function MAXIMUM is not indivisible as a whole.
In the 8-th line of the algorithm Lamport uses the following abbreviation:

(number(j),j) < (number(i),i) for
{number(j) < number(i) **OR** *(number(j) = number(i)* **AND** *j < i)}*.

```
1 begin integer j;
2 Li1: choosing(i) := 1;
3      number(i) := 1 + maximum(number(1),..., number(N));
4      choosing(i) := 0;
5      for j := 1 step 1 until N do
6      begin
7 Li2:      if choosing(j) ≠ 0 then goto Li2;
8 Li3:      if number(j) ≠ 0 AND (number(j),j) < (number(i),i) then goto Li3;
9      end;
```

— — — — —

(1) They may even be avoided deliberately. For instance, it is not unusual in telephone systems and in railway-management systems to *decrease* the priorities of calls/trains which have incurred a (large) delay, in favor of the other calls/trains. (In order to avoid the transference of delays.)

```
10      critical section;
11      number(i) := 0;
12      non-critical section;
13      goto Li1;
14  end
```

Array CHOOSING is used to prevent processes from overtaking one another in the arbitration algorithm. Observe that a process will always detect all processes which have previously or simultaneously executed line 2, but not yet line 11. Such processes are detected at line 8, as guaranteed by the test in line 7. The working of the algorithm is further self-explanatory.
We can make two objections against it:

1. The elements of array NUMBER can become arbitrarily large, if at least one process is executing between line 2 and line 11, at all times.
2. Processes with a low identity number ("i"), are systematically favored, each time they arrive simultaneously with other processes (see line 8).

Lamport noted the first objection, but gave no solution. He did give a hint that this arbitrary increase would be unavoidable. It is tempting to make the MAXIMUM function circular, for instance, by adding a MODULO function to it (modulo 2*N-1 would do). The fact that the MAXIMUM function is not indivisible is, however, a 'spoil-sport'. Observe that the execution of line 11 in one process may interfere with the execution of the MAXIMUM function in two or more others. In that case the values of the corresponding elements of array NUMBER are no longer correct representations of the relative times of arrival.

It is easier to remedy the second problem. Note that one can use the exclusion which is realized on the execution of line 10, to implement a more symmetric strategy for those processes which have arrived simultaneously. The process which is allowed to proceed to line 10 in Lamport's algorithm, should then first check whether there are other processes in the algorithm with the same value of their element in array NUMBER (note that a test like in line 7 can now be skipped), and if so, determine an ordering for all processes with this value *at random* (in so far as possible of course).
The process then returns to line 5, and awaits its turn in this ordering.
The objection to such a solution is that it makes the algorithm rather bulky.

A more elegant solution results if we restrict ourselves to a slightly weaker requirement than full symmetry, as will be elaborated in the next section.

A.4. Alternatives

One alternative strategy for the arbitration between simultaneously arrived process requests was suggested by **Bredt '70** (in another context). Bredt suggested acknowledging these requests in the order of their previous accesses, which implies that the process which most recently acquired access to its critical section is disfavored.
Obviously, with this strategy, processes can still be disfavored systematically. For instance, a process which is disfavored once will acquire access to its critical section later than its competitors, which could lead to it being disfavored again and again. We can solve this problem simply by reversing the rule: simultaneous requests are acknowledged in the *reverse* order of their previous accesses.
To improve Lamport's algorithm on this point we can include a queue with two standard functions LEVELENTRY(m), and PRIORITY(i).
The length of the queue is N. Each queue element corresponds to a priority level, each queue entry corresponds to a process. The standard function

LEVELENTRY(m) yields the identity of the process with priority m, where m is any number from 1 to N. The standard function PRIORITY(i) yields the priority of the process identified by "i".
The following correspondences hold:

LEVELENTRY(PRIORITY(i)) = i, for all $1 \leq i \leq N$ and similarly
PRIORITY(LEVELENTRY(m)) = m, for all $1 \leq m \leq N$

We can initialize the queue by inserting each process identification on the priority level with the same nominal value, such that LEVELENTRY(1) = 1, LEVELENTRY(2) = 2, ..., etc.

In line 8 of Lamport's algorithm we replace:

$(number(j),j) < (number(i),i)$

with

$(number(j), priority(j)) < (number(i), priority(i))$

which is an abbreviation of:

$\{number(j) < number(i)$ **OR** $(number(j) = number(i)$ **AND** $priority(j) \leq priority(i))\}$

Finally we insert the following line between line no.'s 10 and 11:

rearrange(i);

which is an abbreviation of:

```
for j := priority(i) - 1 step - 1 to 1 do
      levelentry(j+1) := levelentry(j);
levelentry(1) := i;
```

Observe that the operation *rearrange(i)* is executed in exclusive mode. Let us consider whether the execution of this operation can interfere with the test actions in line 8.
The priority of the first processes in the queue is lowered by 1.
In the initial state, no two processes have the same priority, so if in this initial state one process is blocked by another on its priority, this single updating action cannot release it. The priorities of the two processes can at the most have become equal, which is still a blocking condition in line 8.
Note further that two priorities can only become equal if one of the two related processes is executing the operation REARRANGE.
Only the last operation in the REARRANGE macro can release a process, but only one that arrived simultaneously with the process considered. After the completion of REARRANGE the initial state is restored, in the sense that there are no two processes with equal priorities any more.
The exclusion on the code between line no.'s 9 and 11 is again guaranteed.
The protection on the execution of line 11 itself has disappeared, but this cannot cause any problem, clearly.
During the execution of the macro REARRANGE, the correspondences stated above may be temporarily invalid, but this is also harmless.

Independence of N

Still, another criticism on the algorithm discussed above can be that the number of competing processes must be known. Preferably one should be able to abstract from this number.
An arbitration algorithm which is independent of N seems feasible when we use

another type of system structure (at least within the algorithm).
Bredt '70, for instance, discussed a hierarchical system organization, in which each single process communicates with no more than 3 processes. A process need only be aware of the existence (and identity) of these 3 processes. The structure can be represented as a binary tree in which each node represents a process. Each node has two 'children'-nodes and one 'parent'-node. The parent will arbitrate between the access requests of its children, and will in turn apply to its own parent for a 'higher level' permission. Most probably this approach will lead to a less elegant, and less efficient solution than the ones discussed, so we shall not elaborate it further here. (Consider, for instance, the 'propagation times' of requests in the hierarchy.)

A.5. Conclusions

We have discussed the concurrent programming control problem, which was introduced in the literature by Dijkstra in 1965. We have stated the problem and the requirements. One new requirement was added, which we called the 'symmetry requirement'. We have given two interpretations of the term 'symmetry':

1. If M processes try to enter their critical sections simultaneously, each of these processes must have equal chances of gaining access as the first, the second, ..., or the M-th process from that batch.
2. If M processes try to enter their critical sections simultaneously, their requests must be acknowledged in the reverse order of their previous accesses.

We have discussed alterations in one of the more recent algorithms (the one presented by Lamport in 1974) for each of these two interpretations. It was further considered how the arbitration algorithms could be made independent of the number of processes N.
The alteration of Lamport's algorithm for the obeyance of the symmetry requirement with the second (weaker) interpretation is fairly straightforward. The other alterations are more cumbersome.

A.6. References

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

deBruyn, N.G. (1967), *Additional comments on a problem in concurrent programming control*, Comm. ACM, Vol. 10, No. 3, March 1967, 137-138.

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

Dijkstra, E.W. (1968), *Cooperating sequential processes*, In: Programming Languages, Genuys, F. (ed.), Academic Press, New York, 1968.

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

Knuth, D.E. (1966), *Additional comments on a problem in concurrent programming control*, Comm. ACM, Vol. 9, No. 5, May 1966, 321-322.

Lamport, L. (1974), *A new solution of Dijkstra's concurrent programming problem*, Comm. ACM, Vol. 17, No. 8, Aug. 1974, 453-455.

Lamport, L. (1976), *The synchronization of independent processes*, Acta Informatica, Vol. 7, 1976, 15-34.