On a Class of Allocation Strategies Inducing Bounded Delays Only

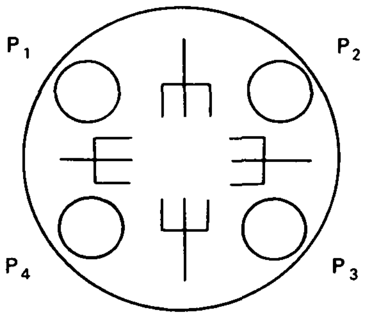
J. J. Cocu and R. E. Devillers

Faculté des Sciences, Laboratoire d'Informatique Théorique,
Université Libre de Bruxelles, Boulevard du Triomphe B-1050, Bruxelles, Belgium

From a critical examination of the class of strategies preventing individual starvation with global control, introduced by E. W. Dijkstra, another class of strategies is introduced, which is simpler to implement and whose behaviour has a straightforward interpretation. Both classes are in fact distinct sub-classes of a more general family of strategies.

INTRODUCTION

Deadlocks and starvations are well known phenomena which may arise in systems of concurrent processes competing for some resources. Historically, one of the most famous examples of such systems is given by the so-called Dining Philosophers' Problem, from which starvation got its name: *N* philosophers are living in a house where the table is laid for them, each philosopher having his own place at the table; their non-ending life consists of an alternation of thinking and eating, but the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks and there is only one fork between two neighbours at the table, so that no two neighbours may be eating simultaneously.[1]



To get a similar system in computer science, one simply has to replace the forks by unitary, reusable, non-sharable, non-preemptible resources (like tape drives, disk packs, page frames, critical sections, . . .) and the philosophers by processes needing cyclically two of these resources to enter a critical section.

Now, a deadlock, i.e. an irremediable blocking of some, or all, of the philosophers, may happen, for instance if each philosopher picks up his left fork and then tries to pick up his right fork.

If the requested forks are allocated in one slot, and not one by one, deadlocks are impossible but starvation, i.e. infinite waiting of some philosophers due to the particular history of the other ones, may arise. Starvation may be due to: deadlock, of course; unfair scheduling, e.g. if some philosophers have always the priority in the waiting queues; coalitions, e.g. if philosopher P_1 succeeds in picking up his forks and then waits until P_3 becomes hungry and picks up his forks, before ending his meal, and if P_3 in turn waits until P_1 becomes hungry again and picks up his forks, before releasing the forks, and so on, we see that P_2 and P_3 will starve; bad luck, e.g. if the preceding history arises due to unfortunate circumstances. Strategies to detect, recover from, prevent and avoid deadlocks have been extensively explored. It seems that a lot of work may still be done to extend, analyse and implement existing strategies to prevent starvation, and to discover new ones.

Dijkstra presented an interesting class of allocation strategies avoiding individual starvation with a global control, for a class of systems generalizing the Dining Philosophers' Problem.[2] He considered a set of *M* processes alternating finite 'eating' phases and possibly infinite 'thinking' phases, where the eating phases are submitted to coherent exclusion rules, such as: any subset of a permissible set of simultaneous eaters is permissible, and each process belongs to at least one permissible set. (For instance, a set of simultaneous eaters would not be permissible if their global needs for resources exceed the availability of the system, if they violate an exclusive access constraint, etc.)

The strategies associate an allowance counter ac_i with each process *i*. This counter is active when the process is hungry; it is initialized to a positive value $N_i \geqslant M-2$ when that process becomes hungry, is decremented by 1 when another process starts eating, and may not become negative. Dijkstra showed that a situation is safe, i.e. free from the danger of deadlock and starvation, iff

(S_0): $\forall k \geqslant 0$, at most k active counters have a value $< k$

or equivalently

(S_1): it is possible to order the counters in such a way that the first is $\geqslant 0$, the second is $\geqslant 1$, the third is $\geqslant 2$, etc. . .

A process is then admissible if: he is hungry; his admission to the set of eaters would preserve permissibility, i.e. would not cause violation of the simultaneity restrictions; his admission would preserve safeness.

We shall analyse here the hypotheses of the problem in order to point out some limitations of Dijkstra's approach. We shall then define a larger class of strategies, in order to cope with a variable number of processes, and we shall show that they may be implemented at a reasonable cost. We shall also examine an interesting subclass, distinct

CCC-0010-4620/82/0025-0052$02.00

from Dijkstra's example, which has a simple interpretation and a simple admissibility test (at least for its safeness part).

THE GOAL

Clearly, various starvation prevention goals may be considered. First, one might require that the hungry periods are all bounded by a fixed value. We denote this property by 'strong starvationfreeness'; it is probably the more interesting one. This is the goal achieved here, and in Ref. 3 for instance. Second, one might only require that the hungry periods are finite. This property may be called 'weak starvationfreeness'; this is the case considered in Ref. 4 for instance, due to the few hypotheses made on the processes and the synchronization mechanism. Finally, one might require that each hungry period is bounded but that the set of the upperbounds may be unbounded. So, the first hungry period of a process could be bounded by 1, the second by 2, etc.... This property could be called 'locally strong starvationfreeness'. Observe that two subcases may be distinguished, depending on whether these bounds are known beforehand, or only when the hungry period starts. Hybrid cases are also possible, where for each process it is specified if it has to guarantee a strong, a weak or a locally strong starvationfreeness, or if it may starve.

THE PROCESSES

The set of processes

Dijkstra considers a fixed, finite set of processes. Finiteness is quite natural in a computer environment but it may be interesting to examine how strategies are sensitive to this hypothesis. This was done in Ref. 3, for instance, where the authors showed that, with their general class of strategies with distributed control for systems only containing unitary resources, the delays remain bounded if each process has a finite bounded set of neighbours (two processes are considered neighbours if they need a same unit of resource: thus they do not occur in a same admissible set). This is not the case here, of course, due to the initialization condition $N_i \geqslant M - 2$. More generally, it may be observed that strategies with global control are likely to be incompatible with infinite sets of processes.

The fact that the set of processes is fixed beforehand may be more annoying for the application of the strategies to real problems, since it is well known that processes may be created or destroyed in a computer, even in the simplest multiprogramming O.S. Consequently, one should allow the creation and destruction of processes, in their thinking phases, with adequate and coherent modifications of the exclusion rules. For instance, when a process is destroyed, the new admissible sets of simultaneous eaters could be those of the old ones where the destroyed process does not occur; when a process is created, the new set of admissible sets should be such that if the process is destroyed, one gets back the old situation (if the system was not empty, the solution is not unique).

If the number of processes is bounded, Dijkstra's strategies may be used if we replace M by the bound, but the limitation on the initial values may be uselessly restrictive. If the number of processes is unbounded, the strategies have to be revised.

The eating periods

Dijkstra also supposes that the eating periods (including their associated synchronization actions) are all larger than some positive lowerbound and smaller than some finite upperbound.

The first limitation is natural in a computer context but the strategies are not really sensitive to it. Simply, if some processes may have eating periods with null duration, or durations tending to zero, it is not always necessary to limit the number of times they may eat before another hungry process. But it remains sufficient to do so.

The second limitation is necessary to achieve the strong starvationfreeness; only specifying that the eating periods are finite would lead to weak starvationfreeness. It may be interesting, however, to examine what would be the impact of a failure of some process during its critical phase, which would be equivalent to the occurrence of an infinite eating period. In Ref. 3 for instance, the authors showed that, for their general strategies, only the neighbours of the failing process, and their neighbours, are affected. Here, this is not the case since, if a hungry process has a null counter and a failing neighbour blocks it, all the hungry processes will wait for ever.

THE SYSTEM

Dijkstra's strategies are based on two properties of the system: (A) the existence of a sleeping process (i.e. a process whose preceding request for eating has been delayed) implies at least one process who is eating or leaving the table; (B) for any process i it can be guaranteed that during a period of his hungriness the decision to admit someone else to the table will not be taken more than N_i times, where N_i is a given, finite upperbound for process i. Dijkstra showed that, in his context, these properties are necessary and sufficient to satisfy strong starvationfreeness.

For property (A), this is due to the fact that each process operates the cycle

```
do think; ENTRY; eat; EXIT od
```

with the (implicit) hypothesis that, if a process is not immediately admitted to the table, through its inspection procedure ENTRY, then it may only be woken up by another process, performing its inspection procedure EXIT, when it has finished eating, and with the (explicit) hypothesis that the inspection procedures are exclusive.

Now it may happen that these hypotheses are not satisfied, and that property (A) is not necessary. For instance, let us consider a system where the resources are managed by a separate system process, let us call it the distributor, which periodically looks if some processes have signalled the release of their resources (in EXIT) and if some have signalled their need for new resources (in ENTRY); the ENTRY procedure may have the form

initialize the counter; flag the process hungry;
wait for the resources

and it may happen that all the processes are hungry simultaneously, until the next working period of the distributor.

A similar behaviour, with deferred decision, may also be observed with some synchronization mechanisms, like Petri Nets[5] or Path Expressions,[6] where the decision to fire an enabled transition may be delayed a certain amount of time. Moreover, for some implementations, it may be more natural to use ENTRY parts which are exclusive only during the examination of the situation, and not during the initializations: ENTRY may then have the form

initialize the counter; flag the process hungry;
exclusive examination

with the consequence that it may again happen that all the processes are simultaneously hungry, until one of them reaches the examination part.

Consequently, a more general hypothesis would be (as in Ref. 4): (A′) a situation where some processes are hungry and the other ones are thinking, is unstable; after a finite (bounded) period of time, at least one process will be admitted to the table. It may be observed that Dijkstra's strategies are not really sensitive to this generalization. Simply, as the M processes of the system may be simultaneously hungry for a while, in order to avoid a deadlock due to the occurrence of an unsafe situation when a new process becomes hungry, it is necessary to choose $N_i \geqslant M - 1$, instead of $M - 2$.

For property (B), this is due to the hypotheses on the duration of the eating periods quoted in the discussion on eating periods. Beyond the associated remarks, it may be mentioned that, even if property (B) is necessary and sufficient, it is not necessary to base the strategies on it; simply, they have to imply it. That is what happens for instance in the distributed strategies,[3,4] where the behaviour of each process is only influenced by his immediate neighbours.

The class of exclusion rules considered here is applicable to systems with interchangeable, partly sharable, multiple resources. This is much more general than the systems considered in Refs. 3 and 4 for developing distributed strategies, since the neighbourhood concept is equivalent to the hypothesis that resources are unique, non-interchangeable and non-sharable (a relaxation of some of these restrictions may be found in Ref. 7). One must observe, however, that the resources are requested and granted in one slot. Therefore, the strategies considered here do not apply to the banker's problem, for instance (as Lauensen's strategies do).[8]

A LARGER CLASS OF STRATEGIES

It may be observed that the safeness property is independent of the static aspects of the systems Dijkstra considered. Let us consider now finite systems, where the number of processes may vary dynamically, by the creation and destruction of processes in their thinking phase. Initially, the situation is safe; the creations and the destructions of processes do not modify the safeness, nor the realizability, of the present situation; if the arrivals and admissions of hungry processes preserve safeness, then, deadlocks and starvations will be avoided.

Consequently, if a strategy is such that: when a process becomes hungry, its counter is initialized to a value which preserves the safeness, without further restrictions; a hungry process is admitted only if this does not violate the exclusion rules, and if this admission preserves the safeness; property (A′) is enforced—then, locally strong starvationfreeness is achieved. Moreover, if the number of processes is bounded, the initial values of the counters may be chosen bounded, and the strong starvationfreeness is obtained, as in the subclass considered by Dijkstra.

IMPLEMENTATION

Even in the more general cases, these strategies may be implemented at a reasonable cost, by using the ideas already developed in Ref. 9. The starting point is that the safeness criteria (S_0) and (S_1) are equivalent to the following one: (S_2) the non-decreasing sequence of the active counters is such that the kth is $\geqslant k - 1$, for $k = 1,2,\ldots$ In order to be able to use this property, it is not necessary to sort systematically the set of counters, but simply to maintain an ordered list during the new arrivals (insert the counter at the right place) and the admissions (drop the counter).

Now, if F is the first (and L the last) counter value which is equal to its rank -1, in the ordered sequence of the active counters of a presently safe situation ($F = \infty$ and $L = -\infty$ if no such counter exist), then (T1) when a process becomes hungry, the situation remains safe iff the initial value of its counter is $> L$; (T2) when a hungry process is admitted, the situation remains safe iff its counter is $\leqslant F$.

These properties (T1) and (T2) are trivial if one looks at the evolution of the ordered sequence of the counters, before and after the critical values F and L. They give simple criteria for the safe choice of the initial value and for the same admission of a hungry process.

AN INTERESTING SUBCLASS

Although the implementation of the general strategies mentioned above is reasonably efficient, it requires the maintenance of an ordered list of active counters and of two critical values. This may be avoided for the subclass of strategies where all the active counters are different. Indeed (T3) if the active counters are all different, then (a) the situation is safe, (b) if one of the counters is null, the associated process is the only admissible one, (c) if the counters are strictly positive, any hungry process may be admitted if this does not violate the exclusion rules, (d) if an admissible process is admitted, the remaining counters remain all different. The proofs are immediate.

Initially all the counters are different (there are none of them). From (T3), it results that if the initial values of the counters are chosen in order to be different from the counters currently in the system, we get automatically a strategy of the subclass. To get a fast test for the safe admission of a hungry process, one simply has to maintain a flag indicating if there is a null counter.

A SIMPLE FAMILY OF STRATEGIES

To get easily a strategy of this subclass, one may for instance choose the initial value of the new counter greater than the counter values presently in the system. That is what is done in the family of strategies defined by the following actions. (We shall not give here a detailed algorithm for these strategies, embedding the quoted actions in a specific program, with a particular choice for the synchronization mechanism; such an implementation may be found in Ref. 7.):

```
choose a non-negative value $d$
initialize: $M := d$ and nozero := **true**
when process $i$ becomes hungry, initialize his counter by
    $ac_i := M$; **if** $M = 0$ **then** nozero := **false** **fi**; $M := M + 1$
the safe admissibility of process $i$ is indicated by the boolean expression:
    nozero $\vee$ $ac_i = 0$
when a process is admitted, perform:
    nozero := **true**; **for** each process $j$ remaining hungry
    **do** $ac_j := ac_j - 1$; **if** $ac_j = 0$ **then** nozero := **false** **fi**
    **od**; $M := M - 1$
```

Admissions may be performed by a separate controller, or by the processes themselves, while signalling their hungriness or leaving the table. Of course, adequate mutual exclusions have to be ensured.

It may be observed that these strategies present the following invariant relations: (T4) $M = d +$ number of hungry processes; $\forall$ hungry process i: $ac_i < M$; $\forall$ hungry processes i, j: $ac_i < ac_j$ iff i is hungry since a longer period than j.

The behaviour of these strategies is quite simple and interesting. If $d = 0$, we simply have a FIFO queue: the oldest hungry process has a null counter and is the only one admissible (the strategy then has some similarities with Lamport's bakery's algorithm[10]). If $d > 0$, we have what may be called a FIFO queue with d degrees of freedom, in the sense that each hungry process may be overtaken at most d times, i.e. at most d processes may start eating before it, while they became hungry after it. This is due to the fact that, if a process has been overtaken d times, so are the processes preceeding it in the ordered list; now, if we consider the first of them, i.e. the oldest hungry one, its counter was initialized to: $d +$ the number of hungry processes preceding it; as all these processes, plus d other (urgent) ones have started eating, we see that the first process has a null counter and is the only one which may be admitted; when he will start eating, the situation will be the same for the next one, until all the processes overtaken d times have been admitted. The processes overtaken d times thus constitute a FIFO queue. Let us also mention that, for $d > 0$, one may drop the test on M when a process becomes hungry.

CONCLUSION

From a critical examination of the hypotheses underlying Dijkstra's strategies, we have obtained a larger class of strategies, with a reasonable implementation cost, and a distinct subclass, which is very easy to implement and presents a simple behaviour. In particular, we meet the FIFO handling as a special case.

REFERENCES

1. E. W. Dijkstra, Hierarchical ordering of sequential processes. *Acta Informatica* **2** (No. 1), 115–138 (1971).
2. E. W. Dijkstra, A class of allocation strategies inducing bounded delays only. *Spring Joint Computer Conference*, 933–936 (1972).
3. P. J. Courtois and J. Georges, On starvation prevention. *RAIRO Informatique* **11** (No. 2), 127–141 (1977).
4. R. E. Devillers and P. E. Lauer, A general mechanism for avoiding starvation with distributed control. *Information Processing Letters* **7** (No. 3), 156–158 (1978).
5. J. L. Peterson, Petri nets. *Computing Surveys* **9** (No. 3), 223–252 (1977).
6. P. E. Lauer and R. Campbell, Formal semantics for a class of high level primitives for coordinating concurrent processes. *Acta Informatica* **5** (No. 4), 297–332 (1975).
7. J.-J. Cocu, *Stratégies de Prévention de Famine*, Internal Report 61, L.I.T., CP 212, University of Brussels (1979).
8. S. Lauensen, Job scheduling guaranteeing reasonable turn-around time. *Acta Information* **2** (No. 1), 1–11 (1973).
9. R. E. Devillers, *On the Implementation of a Class of Allocation Strategies Inducing Bounded Delays Only*, Technical Report 95, L.I.T., CP 212, University of Brussels (1980).
10. L. Lamport, A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM* **17** (No. 8), 453–455 (1974).

Received November 1980