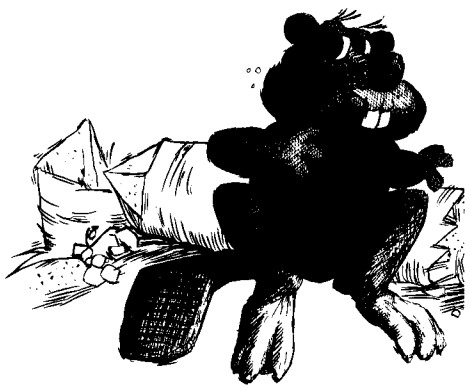
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT



A Robust Distributed Solution to the Dining Philosophers Problem

T.A. Cargill

CS-81-20

May, 1981

A Robust Distributed Solution to the Dining Philosophers Problem*

T.A. Cargill

Department of Computer Science,
University of Waterloo,
Waterloo, Ontario
Canada N2L 3G1

ABSTRACT

A distributed solution to Dijkstra's "Dining Philosophers" problem is presented. There is no central synchronization mechanism and it is robust in that the impact of failures is local. The solution is amenable to implementation using a wide class of synchronization primitives. A concrete implementation is presented in Ada.

Keywords and phrases: Ada tasking, concurrency, deadlock, dining philosophers, distributed computing, robustness.

CR Categories: 4.32

1. Introduction

Dijkstra's "Dining Philosophers" problem [1] is one of the standard programming exercises in resource allocation. An arbitrary number of "philosophers" are seated at a circular table. In front of each is a plate of "a very difficult kind of spaghetti". Between each pair of philosophers is a fork. The philosophers alternately "think" and "eat" in an unpredictable manner. When thinking a philosopher requires no forks. In order to eat a philosopher must acquire the two adjacent forks. The problem is to provide a deadlock-free and starvation-free fork allocation discipline.

Most of the solutions which have been proposed in the literature, for example [1,2,3,4], have needed some kind of centralized synchronization mechanism, by means of shared semaphores, monitors or a central process. Moreover, the solutions presented in these references allow starvation, modification of the solution being left as an exercise.

The global synchronizaton required in these solutions is unsatisfactory for two reasons. First, the problem is inherently distributed in nature. It seems

* Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A5046, and the University of Waterloo.

unacceptable that two philosophers positioned an arbitrary distance apart at the table should be so closely coupled; the allocation of forks in two distant parts of the table should be able to proceed in parallel. Second, these solutions are not robust. Because of the tight coupling, the failure of one component of the solution can have a widespread impact. What happens if a philosopher process dies inside a monitor or before executing a semaphore V operation? What happens if the controller process dies?

Two previous distributed solutions are known to the author. Chang [5] proposes a pair of processes per philosopher: a "philosopher" process and a "control" process. Philosopher processes think, get hungry and eat. Control processes know the states of their respective philosophers and form a message-passing ring. Deadlock is detected by a message, originated by a control process whose philosopher is blocked, which travels round the entire ring finding all other philosophers to be also blocked. When deadlock is detected, the holder of a special token is obliged to relinquish a fork and pass on the token. In addition to being quite complex to describe, this solution has several deficiencies. First, every philosopher at the table must be blocked before any resolution is attempted. Second, deadlock detection takes time proportional to the number of philosophers, since a message must circumambulate the table. Third, the entire algorithm fails if any control process is lost. Francez and Rodeh [6] also propose a distributed solution, where the avoidance of starvation depends on the asymptotic behaviour of stochastic synchronization primitives. The probabilistic nature of their approach makes it difficult to evaluate against standard criteria.

2. A Robust Distributed Solution

We therefore propose the following solution. It is distributed in the sense that synchronization and communication is limited to the immediate neighborhood of each philosopher. There is no central mechanism. It is robust in the sense that the impact resulting from the failure of any component is limited to its immediate neighborhood.

Let the forks be labelled consecutively around the table by the integers 1 through N. Let the allocation of each fork be controlled independently, such that philosopher processes may acquire and release forks. This may be accomplished by means of semaphores, monitors or independent processes, but the control of each fork is isolated from that of the others.

When a philosopher wishes to eat, his two forks are acquired in turn *starting with an odd numbered fork*. When a fork is not acquired immediately, the requesting philosopher blocks until it is freed by the philosopher holding it. Having finished eating, the philosopher releases both forks.

Each philosopher interacts with only two forks, one on either side, and the only interaction is that of acquisition (possibly after blocking) and release. However, there are now two classes of philosopher alternating round the table, those who pick up a left fork first and those who pick up a right fork first. There is no additional control or synchronization.

3. Example

An implementation of this strategy is given in Ada in Fig. 1. An instance of the task type Fork_proprietor is associated with each fork. Each Fork_proprietor task has two entries, Allocate and Release, for allocating its fork to a philosopher and for allowing the philosopher to return it. The body of Fork_proprietor is a loop which repeatedly accepts an Allocate entry call followed by a Release entry call, behaving like a binary semaphore.

An instance of task type Philosopher is associated with each philosopher. The body of Philosopher first determines the identities of its forks by calling the Place_assignment entry of the Maitre_d task. Before the Philosopher task eats, it calls the Allocate entries of the appropriate Fork_proprietor tasks in the correct order. It may block on either of these entries. After eating, the Release entries of the same Fork_proprietors are called. These do not block.

The Maitre_d task allocates places at table to philosophers. Once allocated, they remain fixed. The body of Maitre_d loops accepting one Place_assignment entry call from each Philosopher, allocating places sequentially round the table.

4. Deadlock and Starvation

The strategy is deadlock-free and starvation-free. This is established by showing that any philosopher will always be allocated both forks successfully after a finite waiting time (bounded, if eating times are bounded).

Consider the case where a philosopher is blocked in attempting to acquire his second fork. The second fork is an even fork* and is therefore the second fork of the neighbor who is using it. The neighbor is therefore eating and may continue to eat for a finite time. When the neighbor releases this fork, the philosopher in question can be the only one waiting for it. If the fork is allocated fairly (for example, Ada's entries are handled in a first-come-first-served order), then this philosopher acquires the fork and eats.

Now consider the case where the philosopher is blocked in attempting to acquire his first fork. This fork is odd and is the first fork allocated by a neighbor*. This neighbor is either eating or blocked waiting for a second fork. From above, if he is blocked waiting for a second fork, the neighbor eats after a finite time. Therefore, after a finite time the neighbor holding the first fork of the philosopher in question releases it and, again assuming fairness, it can be acquired. The philosopher may now block on the second fork, but the wait is again finite.

5. Robustness

In the proof a philosopher never depends on another which is further removed than a neighbor's neighbor. To illustrate this, consider what happens if a philosopher one further removed than this fails while holding both forks. If p_0, p_1, p_2 and p_3 are consecutive philosophers, we must show that the failure of p_3 does not affect p_0. Now, p_2 and p_3 share either an even or an odd fork. If even, then p_2 may block indefinitely holding the odd fork it shares with p_1, but p_1

*If there are an odd number of forks, then one philosopher has two odd forks. This special case lengthens the proof, but does not change the result.

cannot acquire the even fork it shares with p_0. If the fork shared by p_2 and p_3 is odd, then p_2 may block, but p_1 and p_0 are unaffected.

6. A Weakness of the Strategy

If the requests for forks are made serendipidously, it is possible to attain the upper bound of N/2 eating philosophers. But there is no guarantee that when all philosophers wish to eat that N/2 will do so. In the worst case only N/4 philosophers eat.

7. Acknowledgements

I wish to acknowledge fruitful discussions with K.S. Booth and E.J.H. Chang.

8. References

1. E.W. Dijkstra, Hierarchical ordering of sequential processes. Acta Informatica 1:2 pp 115-138 1971.
2. P. Brinch Hansen, Distributed Processes: A Concurrent Programming Concept. Communications ACM 21:11 pp 934-941 1978.
3. C.A.R. Hoare, Communicating Sequential Processes. Communications ACM 21:8 pp 666-677 1978
4. R.C. Holt, G.S. Graham, E.D. Lazowska, M.A. Scott, *Structured Concurrent Programming with Operating Systems Applications.* Addison-Wesley, Reading, Mass 1978.
5. E. Chang, n-Philosophers: an Exercise in Distributed Control. Computer Networks 4 pp 71-76 1980.
6. N. Francez, M. Rodeh, A distributed abstract data type implementation by a probabilistic communication scheme. Proc. 21st Annual Symp. on the Foundations of Computer Science, pp 373-379 1980.

```
N : constant integer;                -- no value is specified here
subtype Fork is integer range 1..N;

task Maitre_d is
  Place_assignment(left,right : out Fork);
end Maitre_d;
task body Maitre_d is
begin
  for f in 1 to N loop
    accept Place_assignment(left,right : out Fork) do
      left := f;
      right := f mod N + 1;
    end Place_assignment;
  end loop;
end Maitre_d;

task type Fork_proprietor is
  entry Allocate;
  entry Release;
end Fork_proprietor;
task body Fork_proprietor is
begin
  loop
    accept Allocate;
    accept Release;
  end loop
end Fork_proprietor;

task type Philosopher;
task body Philosopher is
   left, right : Fork;
begin
  Place_assignment(left,right);
  loop
    -- think
    if odd(left)
      then
        Allocate(left);
        Allocate(right);
      else
        Allocate(right);
        Allocate(left);
    end if;
    -- eat
    Release(left);
    Release(right);
  end loop
end Philosopher;

Philosophers      : array (1..N) of Philosopher;
Fork_proprietors : array (1..N) of Fork_proprietor;
```

Fig. 1 The solution expressed in Ada.