Progress with Java threads as dining philosophers

Wim H. Hesselink, 31st August 2001

Abstract

The classical problem of the dining philosphers is solved and implemented with Java threads, without central coordination. Apart from the initialization the program is symmetric. The philosophers are the synchronized objects whereas the forks between the philosophers are unsynchronized. It is proved that every hungry philosopher eats within bounded delay, in a certain formalized sense.

Keywords concurrency, thread, Java, program correctness, bounded delay.

1 Introduction

This paper has four goals. Firstly, it presents a new and simple solution to the classical problem of the dining philosophers without individual starvation, cf. [8]. Secondly, it shows the adequacy of Java threads for this touch stone of multiprogramming. Soundness of the solution is proved formally by means of invariants. Thirdly, the absence of individual starvation is formalized in terms of bounded delay, a concept stronger than the classical concepts of weak and strong fairness. The proof of bounded delay is sketched. Finally, we assess the progress that has been made in this field since the appearance of [8] in 1971. Thus, the word "progress" in the title refers both to the absence of individual starvation and to the evolution of the field.

The problem of the dining philosophers

The problem of the dining philosophers was presented in [8] with a solution that allows individual starvation of philosophers. The paper mentioned this shortcoming but gave no explicit solution for it. Since then many solutions have appeared. We here present a slight generalization of the problem, cf. [10], and a new solution, which is a shared memory simplification of the distributed solution in [5, 6].

The problem is as follows. There are N philosophers with fixed seats in an undirected graph (classically, $N = 5$ and the graph is a ring). Each philosopher alternatingly thinks and eats. It is required that philophers with seats next to each other never eat concurrently. It follows that the

neighbours of a hungry philosopher may conspire to keep him from eating forever by eating alternatingly. The progress requirement is therefore that every hungry philosopher is allowed to eat eventually.

For concreteness, we specify that the philosophers are threads that execute the repetition

```
(1)     while (true) {
            think ;
            toTable() ;
            eat ;
            fromTable() ;
        }
```

The commands `think` and `eat` are supposed to be given. It is given that `eat` always terminates within bounded delay. The commands `toTable` and `fromTable` are to be refined in such a way that philosophers at neighboring nodes never `eat` concurrently and that every philosopher that calls `toTable` will `eat` within bounded delay.

Historical context

We don't aim at a historical survey. The next paragraphs are therefore only a very incomplete sketch. Yet, treating such an old problem with younger means asks for an assessment of the evolution of the field in the meantime.

The ideas of sequential processes and multiprogramming occur as early as 1968, when Dijkstra [7] introduced the semaphore, one of the first synchronization primitives. Synchronization by means of monitors was introduced in 1974/75 by Hoare [14] and Brinch Hansen [3]. Later, Milner [21] and Hoare [15] proposed algebraic approaches to processes. In [6], Chandy and Misra introduced UNITY as a programming formalism without any sequentialization. All this work on multiprogramming was mainly applied in operating systems design, since application programmers usually (wisely) kept to sequential programs.

This changed in the nineties, primarily because of the advance of windowing systems. Now, application programmers had to cope with concurrent threads of control and the term thread was introduced since, in the sphere of operating systems, the term process is also used for a collection of reserved resources, cf. [22]. Newer programming languages like ADA, SR, Modula-3, and Java, offered facilities for thread programming. In 1998, the POSIX

committee specified a standard (1003.1c) for the handling of threads and their synchronization under UNIX, cf. [4]. The synchronization primitives of Modula-3, Java and the POSIX threads are variations of the monitors of Hoare [14] and Brinch Hansen [3].

Although the dangers of deadlock and individual starvation were well-known in 1968, cf. [7], the formal analysis of progress properties of concurrent processes started around 1981 with the introduction of weak and strong fairness, cf. [18, 11, 16]. Roughly speaking, weak fairness means the assumption that, in every infinite execution, a step that is enabled from a certain point onward is taken infinitely often. Strong fairness means the assumption that, in every infinite execution, a step that is enabled infinitely often is taken infinitely often. Weak fairness is easily implemented, but, in general, strong fairness is regarded as too costly. Therefore, most programming formalisms are based on weak fairness, e.g., UNITY [6].

Weak and strong fairness have no implicationss for finite executions. So, in a solution of our problem that guarantees that every hungry philosopher will eventually eat under weak fairness, a hungry philosopher that has to wait indefinitely, can only complain after having reached immortality. We therefore prefer to use the concept of bounded delay or bounded fairness, as introduced in [13]. It is a formal concept that implies bounded delay in a real-time sense under weak assumptions on the scheduler and the processor(s).

Solutions and additional requirements

There are several solutions, e.g., the first solution of [10], that use global coordination. This is regarded as undesirable since it can lead to contention. We therefore stipulate that the only communication allowed is between neighbours. In Java terms, there must be no object with synchronized methods that can be called by all philosophers.

The simplest solution known to me is the second one of [10]: it uses an arbitrary numbering of the edges, associates a semaphore to every edge, and lets a hungry philosopher invoke its incident semaphores in the order of the numbering. This solution has bounded delay since Dijkstra's semaphores always have a waiting queue, as formalized in [20]. As an aside, if one would use the plain semaphore of [1], progress would require strong fairness (we leave this to the interested reader). The Java solution of [19] is a special case of this idea for $N = 5$. Notice that the edges (forks) are the synchronized objects in this solution.

We aim at a solution without an asymmetry as induced by a numbering

of the edges. Yet, as shown in [17], any deterministic solution without global coordination has to be asymmetric. We therefore allow asymmetry in the initialization. In our solution, the nodes (philosophers) are the synchronized objects, not the edges (forks). It has the special feature that a philosopher that remains hungry when leaving the table, eats at least as often as its neighbours.

Overview

In section 2, we describe our solution in natural language. Section 3 contains the implementation in Java. In section 4, we give a formal proof of the safety property. Here we formalize the properties of Java's synchronized methods. Section 5 contains a definition of bounded delay and a proof that every philosopher that calls `toTable` will `eat` within bounded delay. Since our solution does not satisfy the condition of "economy" [5], we present two economic alternatives in section 6. In section 7, we evaluate our solution and discuss its migration to other synchronization primitives.

2 Using forks and borrowing them

We enforce the safety requirement in the traditional way by introducing one fork on every edge between philosophers and by stipulating that a philosopher can only `eat` when it holds the forks of all incident edges. This clearly implies that neighbours can never `eat` concurrently.

The problem now becomes: how does a hungry philosopher acquire the forks it needs for eating? There are several ways to regulate this. The simplest way that we know of is as follows: a philosopher that executes `fromTable` surrenders all its forks to its neighbours. This idea of scheduling by edge reversal [2] is due to [5]. It could be sufficient if the neighbours would be guaranteed to become hungry within bounded delay.

Since philosophers can think indefinitely, however, we have to allow a hungry philosopher to obtain a fork held and owned by a thinking neighbour. We do this by introducing the possibility that a philosopher borrows the fork. This raises the question what to do when the thinking neighbour gets hungry? Well, the owner can claim a borrowed fork back, when it is not yet in use.

This regulation does not yet preclude deadlock or individual starvation. Indeed, if all philosophers get hungry at the same time and the forks are distributed in such a way that no philosopher can eat, we have deadlock.

We have to break the symmetry and introduce priorities in some way. For this purpose, we introduce an integer ghost variable `level` for each philosopher, with the invariant that neighboring philosophers always have different levels and that the lower level philosopher always owns the fork. This can be initialized easily. Whenever a philosopher executes `fromTable` and surrenders its forks to its neighbours, it sets its level to $M + 1$ where M is the maximum level of its neighbours. This clearly preserves the invariant.

Let us now show that this precludes individual starvation and hence, a forteriori, deadlock. A philosopher is regarded as hungry when it is not thinking. We have to prove that all hungry philosphers eventually eat and start thinking again. For a hungry philosopher, we therefore count the number of its *predecessors*, i.e., the number of philosophers with lower levels.

A hungry philosopher without hungry predecessors can borrow all missing forks from its thinking neighbours and can then eat and depart. If however before it starts eating, the owner of one of its borrowed forks claims it back, that is a hungry predecessor.

It follows that, for every hungry philosopher, some hungry predecessor will eat and depart within bounded delay. In other words, the number of predecessors decrements within bounded delay. Since the number of philosophers is bounded, a hungry philosopher will eat within bounded delay.

This idea is a variation of the idea of a directed graph as used in [5]. No central coordination is needed and all philosophers and edges are treated in the same way. The only asymmetry is in the initialization. Our solution is a simplification of the one of [5, 6] in that we do not encode the fork requests in the data structure. Note that the levels are ghost variables that are absent from the implementation and only occur in the proof of progress.

3 An implementation with Java threads

The language Java offers the possibility to declare methods as *synchronized*. When one thread enters a synchronized method, Java guarantees that, before it has finished, no other thread can execute any synchronized method on the same object.

Java also offers primitives for thread suspension. When a thread calls `wait` inside a synchronized method of an object, it is deactivated and put into the waiting queue associated with the object. Threads waiting in the queue of an object can be awakened by calls of `notify` and `notifyAll` for

this object. The call of `notify` awakens precisely one waiting thread (if one exists), whereas `notifyAll` awakens all threads waiting at the object.

The first step for implementing the dining philosophers in Java is to decide upon the synchronization. The main synchronization point is when a philosopher holds all forks but a neighbour who owns a fork claims it back. The philosopher can then decide to start eating or to yield the fork. In order to start eating, the philosopher must have a synchronized overview of its forks. We therefore decide that the philosophers form the synchronized objects, not the forks.

We construct a class Philosopher that extends Thread. We give it an integer instance variable `self`, an array of neighbours `nhb` of length `degree`, and an array `phil` to hold the philosophers. We introduce a method `missingFork` that, if possible, returns a number `mf` of a missing fork in the list of neighbours of the philosopher. If so, `nhb[mf]` is the number of that philosopher and `phil[nhb[mf]]` is the corresponding philosopher object. The result equals `degree` when there is no missing fork.

We provide every philosopher with two boolean instance variables `hungry` and `eating`. Command `toTable` is now refined as

```
setHungry () ;
while ((mf = missingFork()) < degree)
    phil[nhb[mf]].yieldForkTo(self) ;
```

where `setHungry` is a synchronized method that only sets `hungry` to true and where `yieldForkTo` is a method to yield a fork to a neighbour philosopher.

The method `missingFork` performs a linear search for a missing fork and set `eating` true when all forks are present.

```
synchronized int missingFork () {
10      int mf = 0 ;
11      while (mf < degree && gr.holds(self, nhb[mf])) mf++ ;
12      if (mf == degree) eating = true ;
13      return mf ;
}
```

The graph object `gr` keeps the administration of the forks: `holds(q,r)` says whether `q` holds the fork on the edge between `q` and `r`. Similarly, `owns(q,r)` says whether `q` is the owner of the fork on the edge between `q` and `r`. The object `gr` has methods `move` and `give` to transfer forks and ownership: the

call of `move(q,r)` makes `r` the holder of the fork between `q` and `r`, whereas `give(q,r)` makes `r` the owner of the fork between `q` and `r`.

A philosopher may begin to eat when it holds all forks, but an eating philosopher must keep its forks. We have therefore declared `missingFork` to be synchronized and we have let it set `eating` to true when the philosopher holds all forks.

A philosopher that holds a fork, does not yield it to a neighbour when it is eating or when it is hungry and owns the fork. Otherwise it can transfer the fork.

```
        synchronized void yieldForkTo (int ph) {
20          if (eating || hungry && gr.owns(self, ph))
21              WAIT
22          else gr.move(self, ph) ;
        }
```

Here, `WAIT` stands for the `try-wait-catch` clause in Java. Note that `yieldForkTo` is called in a state where it is not guaranteed that `self` holds the fork. Indeed, it may well be that after completion of `missingFork` the holder of the fork departs from the table and yields the fork before `yieldForkTo` is called.

A philosopher that departs from the table, resets `eating` and `hungry` and surrenders its forks with ownership to its neighbours. It also notifies the neighbours that may be waiting for its departure.

```
        synchronized void fromTable() {
30          eating = false ;
31          hungry = false ;
32          for (int k = 0; k < degree ; k ++) {
33              gr.move(self, nhb[k]) ;
34              gr.give(self, nhb[k]) ;
            }
35          notifyAll () ;
        }
```

The four synchronized methods described above are the only synchronized methods in the system. In particular, we do not assume any synchronization in the administrative object `gr`. This is allowed since the methods `move` and `give` are always called by a synchronized method and moves forks or ownership away from the synchronized object.

Let us finally consider what happens with greedy philosophers. A philosopher p is called greedy when it remains `hungry` in `fromTable` and skips the next thinking period. In that case, its neighbours can only eat when they own the fork shared with p. They can therefore eat at most once before having to give priority to p. It follows that a greedy philosopher eats at least as often as its neighbours. In particular, greedy neighbours eat alternatingly.

4 The verification of the implementation

The safety property we have to prove is that neighbours never eat concurrently. We use q and r as free variables that range over philosophers (threads). We write $f(q, r)$ to indicate the thread that holds the fork on the edge between q and r. The instance variables of thread q are prefixed with q. We write $Nhb(q)$ for the set of neighbours of q. Since the graph is undirected, we have $r \in Nhb(q)$ iff $q \in Nhb(r)$. Array q.`nhb` is an enumeration of $Nhb(q)$.

Now safety is expressed by the requirement that an eating philosopher holds all incident forks:

(Safety) $\quad q$ **at** `eat` $\quad \wedge \quad r \in Nhb(q) \quad \Rightarrow \quad f(q, r) = q$.

This predicate follows from

(J0) $\quad q$ **at** `eat` $\quad \Rightarrow \quad q$.`eating` ,
(J1) $\quad q$.`eating` $\quad \wedge \quad 0 \leq i < q$.`degree` $\quad \Rightarrow \quad f(q, q.\texttt{nhb}[i]) = q$.

In order to reason about the synchronized methods, we write $syn(q)$ to denote the set of objects of which thread q is in synchronized methods. Java guarantees the general invariant

(Syn) $\quad syn(q) \cap syn(r) \neq \emptyset \quad \Rightarrow \quad q = r$.

For the present algorithm, $syn(q)$ always contains at most one object, which is a philosopher. We have $syn(q) = \{q\}$ if and only if q is in `setHungry`, `missingFork`, or `fromTable`, and $syn(q) = \{r\}$ with $r \neq q$ if and only if q is in `yieldForkTo` and r is the object that q denotes by `phil[nhb[mb]]`.

Predicate (J0) is an invariant since q.`missingFork` sets q.`eating` true whenever it returns q.`degree`.

Preservation of (J1) is harder to prove. It is convenient to use the line numbers in the methods as locations. Predicate (J1) is threatened only by the assignment to `eating` in line 12 and by the assignments to forks in lines 22 and 33. Preservation of (J2) at 12 follows from the new postulate

(J2) $\quad q$ **in** $\{11, 12\} \quad \wedge \quad 0 \leq i < q.\texttt{mf} \quad \Rightarrow \quad f(q, q.\texttt{nhb}[i]) = q$.

We now show that the fork assignments in 33 preserve (J1) and (J2). A fork assignment in 33 by a process $p \neq q$ preserves (J1) and (J2) since the fork stays on its edge and the assignment sets the fork to a value $\neq p$. The fork assignments in 33 by process q itself preserve (J1) and (J2) since $q.\texttt{eating}$ is false again and q is not at 11 or 12.

In order to show that the fork assignment in 22 preserves (J1) and (J2), we postulate the additional invariant that, when thread p executes line 22, the corresponding synchronized object is not eating:

(J3) $\quad p$ **at** 22 $\quad \wedge \quad r \in syn(p) \quad \Rightarrow \quad \neg\, r.\texttt{eating}$.

Now, if p executes 22 with $p = q.\texttt{nhb}[i]$ and $f(q, p) = q$, the fork is moved to $syn(p).\texttt{ph} = p.\texttt{self}$. This threatens (J1) and (J2) only when $syn(p) = \{q\}$. Now (J3) implies $\neg\, q.\texttt{eating}$, so that (J1) is preserved. On the other hand, (Syn) implies that q cannot be at 11 or 12, so that (J2) is preserved.

Predicate (J3) is invariant because of the test in 20, the invariant (Syn), and the fact that $q.\texttt{eating}$ is modified only by synchronized methods of q itself. This concludes the proof of safety.

5 Proof of progress

We first have to discuss the concept of bounded delay. This term goes back to [9] and its formalization to [13]. We use the following execution model. An *execution* is a sequence of (atomic) steps taken by the threads. A step can be various things: a test, an assignment, the entering of, or exiting from a method call, etc. It can also be a skip (no-op) when the thread is disabled. A thread is disabled when it has called `WAIT` and also when it needs to enter a synchronized method that is held by another thread.

A *round* is an execution in which every thread takes at least one step (possibly a skip). An execution has *delay* at least k if it is a concatenation of k rounds. If P and Q are assertions about the state of the system, we say that P leads to Q within bounded delay (notation $P \; o\!\!\rightarrow \; Q$) if there is a number k such that every execution of delay at least k that starts in a state where P holds contains a state where Q holds.

This models the informal notion of bounded delay if we assume that the threads are scheduled preemptively in *round robin* fashion with a fixed time

quantum, say τ, which includes the time needed for thread switching. For, in that case, every execution that lasts longer than $k \cdot N \cdot \tau$ contains at least k rounds. In principle, this idea can be used for real-time applications.

For the dining philosophers, we assume that `eat` always terminates within bounded delay, but `think` need not do so. The progress assertion to be proved is that all calls of `toTable` and `fromTable` terminate within bounded delay.

For this purpose, we first have to show that a thread can execute `setHungry` within bounded delay. A thread q can only be kept from its own synchronized methods by another thread r that occupies a synchronized method of q. This only happens when r calls q to yield a fork. When q has yielded all its forks, it cannot be kept from declaring itself hungry. It follows that q can enter `setHungry` within bounded delay.

We now have to show that a thread that has set itself to `hungry` cannot be kept from calling its synchronized methods. For this purpose, we first observe that, during a hungry period, a thread calls at most three times `yieldForkTo` for every edge: once to borrow the fork when it does not own it, once to wait to become the owner, and once to claim a borrowed fork back.

It follows that, for each serving, a philosopher does only a bounded number of synchronized method calls. The neighbours of a hungry philosopher q can therefore do only a bounded number of synchronized method calls before they have to give priority to q and to start waiting. This shows that a hungry philosopher can do each of its synchronized calls within bounded delay.

We now formalize that every eating philosopher surrenders its forks within bounded delay:

$$\text{(Yield)} \quad q.\texttt{eating} \;\wedge\; r \in Nhb(q) \quad o\!\!\rightarrow \quad H(r,q)\ ,$$

where $H(r,q)$ means that r holds the fork between q and r **and** is its owner. We also need that a hungry philosopher never surrenders its own forks:

$$\text{(Keep)} \quad q.\texttt{hungry} \;\wedge\; H(q,r) \quad \rhd \quad H(q,r)\ .$$

Here, we write $P \;\rhd\; Q$ to mean that, if a step is taken in a state where predicate P holds, the resulting state satisfies predicate Q.

We finally have to show that the fork transfers in 22 are done according to the rules of section 2 and that a waiting philosopher will be notified within bounded delay.

As above, we write $owns(q, r)$ to indicate that q is the owner of the fork between q and r. Since the methods `setHungry` and `fromTable` are declared synchronized, it follows from (Syn) that we have the invariants

$$\text{(J4)} \qquad q \in syn(r) \;\wedge\; r \textbf{ at } 22 \;\wedge\; q.\texttt{hungry} \;\Rightarrow\; \neg\, owns(q, r)\ ,$$
$$\text{(J5)} \qquad q \in syn(r) \;\wedge\; r \textbf{ at } 21 \;\wedge\; \neg\, q.\texttt{eating} \;\Rightarrow\; q.\texttt{hungry} \;\wedge\; owns(q, r)\ .$$

In words, if thread r is visiting q at 22 and q is hungry, then q does not own the fork. If thread r is visiting q at 21 and q is not eating, then q is hungry and owns the fork.

Predicate (J4) expresses that the fork transfer in 22 is done according to the rules of section 2: the object q must yield the fork. Predicate (J5) implies that a thread r only starts waiting because of an eating neighbour or a hungry predecessor. In either case, the waiting thread will get the corresponding fork and will be notified within bounded delay. In the second case, this is proved by induction over the number of predecessors.

6 Economical variations

The above solution has the property that a permanently thinking philosopher may have to yield forks infinitely often. This is not allowed by the so-called economical condition in [5]. We first propose the following remedy: a philosopher that leaves the table does not surrender its forks to its neighbours, but only surrenders the ownership. So, we just remove line 33. This modification is almost correct. Indeed, section 4 can be retained and, in section 5, only property (Yield) fails.

Nevertheless, progress can fail as follows. A hungry philosopher that owns all needed forks but does not hold them, can be scheduled to claim them always from an eating neighbour, which always fails. Yet, under the assumption of strong fairness, one can argue that every hungry philosopher will eat eventually in this solution. For, if a philosopher stays hungry indefinitely, there are infinitely many steps where it can effectively claim a missing fork. So, according to strong fairness, it obtains all missing forks.

Since we aim at bounded delay, we develop another alternative solution. A philosopher that leaves the table only tranfers the fork when its neighbour is hungry. This means that line 33 in `fromTable` is replaced by

```
33      if (phil[nhb[k]].getHungry())
            gr.move(self, nhb[k]) ;
```

Here, `getHungry` is the `get` method of the instance variable `hungry`.

If `getHungry` is synchronized, a thread that needs to set itself hungry may be delayed indefinitely because every time it is scheduled, some neighbour is calling `getHungry` and holds the lock. So then, again, progress would only hold under strong fairness.

We therefore let `getHungry` be unsynchronized. This has the effect that a call of `getHungry` while the neighbour is not in `setHungry` yields the latest value written, whereas a call of `getHungry` while the neighbour is in `setHungry` may yield an arbitrary boolean value. Condition (Yield) is now weakened to

$$\text{(Y)} \qquad q.\texttt{eating} \quad \wedge \quad r \in Nhb(q) \quad \wedge \quad r.\texttt{hungry} \quad o\!\!\rightarrow \quad H(r,q)\ ,$$

but this is enough to prove that every hungry philosopher eats within bounded delay.

We implemented the program of section 3 and the two alternatives and tried to measure the maximal delays for hungry philosophers. It appears that the three programs have roughly the same performance and that the main reason for variation is the activity of other processes. So, in this case, Java thread scheduling seems to support strong fairness.

7 Evaluation

What about progress of the field since the proposal of the dining philosophers in 1971? Perhaps the most striking matter of progress is the availability and quality of thread packages and thread supporting languages like Java. A second point is the Java monitor with its clear semantics. Our three solutions of the problem are clear simplifications of the solution of [5] and thus form a significant contribution since simplicity is an important objective in computer science. A distinctive feature of our solution is that the threads reside in synchronized objects whereas the resources (the forks) remain unsynchronized. Another point is that a philosopher that remains hungry when leaving the table, eats at least as often as its neighbours.

Other striking improvements since 1971 are the invention of object orientation and the development of formal methods. In our example, the main point of object orientation is the Java monitor. As a methodology, object orientation mainly serves to ask the question whether the philosophers or the forks must be synchronized. With respect to formal methods, it is now known

what can be asserted about concurrent algorithms and there are methods to prove such assertions. Above we gave a rather formal proof of safety, whereas the formal progress assertion was proved less formally.

It is easy to translate the algorithm into the primitives for POSIX threads, cf. [4]. Indeed, translating back and forth between Java threads and pthreads helped us in developing the proof of the algorithm. The fact that the queues of the condition variables of pthreads are leaky, cf. [4], is harmless for safety since `yieldForkTo` is called in a repetition. It threatens the progress argument, however, for it allows that, during a hungry period, a thread may call `yieldForkTo` more than three times. Since spurious wakeups are supposed to occur only sporadically, this should not be a problem.

Since condition variables can be implemented with semaphores, our algorithm can also be implemented with semaphores, but we do not know of any illuminating shortcuts.

In our favorite design method for concurrent algorithms, the threads are first synchronized by means of atomic commands of the form $\langle$ **await** B **then** S $\rangle$, which specify to wait until predicate B holds and then to execute command S without interference, cf. [1]. To our surprize it turns out that this more abstract synchronization statement is hardly useful for the algorithm presented here.

Possibilities for future research are a mechanical verification of soundness and progress and a generalization to arbitrary resource allocation. We have set the first steps for mechanical verification: the prelude used for the verification of Peterson's algorithm [12] has been extended to Java threads. Even more promising is the possibility to extend the algorithm to the general problem of agents that occasionaly need resources for critical activities, e.g., cf. [2].

References

[1] Apt, K.R., Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer V. 1991.

[2] Barbosa, V.C., Benevides, M.R.F., Oliveira Filho, A.L.: A priority dynamics for generalized drinking philosophers. Information Processing Letters **79** (2001) 189–195.

[3] Brinch Hansen, P.: The programming language Concurrent Pascal. IEEE Trans. on Software Engineering, SE-1 (1975) 199–207.

[4] Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley 1997.

[5] Chandy, K.M., Misra, J.: The drinking philosophers problem. ACM TOPLAS **6** (1984) 632–646.

[6] Chandy, K.M., Misra, J.: Parallel Program Design, A Foundation. Addison–Wesley, 1988.

[7] Dijkstra, E.W.: Co-operating sequential processes. In: F. Genuys (ed.): Programming Languages (NATO Advanced Study Institute). Academic Press, London etc. 1968, pp. 43–112.

[8] Dijkstra, E.W.: Hierarchical Ordering of Sequential Processes. Acta Informatica **1** (1971) 115–138.

[9] Dijkstra, E.W.: A class of allocation strategies inducing bounded delays only. EWD 319 (1971). See `www.cs.utexas.edu/users/EWD/`.

[10] Dijkstra, E.W.: Two starvation free solutions to a general exclusion problem. EWD 625 (1978). See `www.cs.utexas.edu/users/EWD/`.

[11] Francez, N.: Fairness. Springer V, 1986.

[12] Hesselink, W.H.: Theories for mechanical proofs of imperative programs. Formal Aspects of Computing **9** (1997) 448–468.

[13] Hesselink, W.H.: Progress under bounded fairness. Distrib Comput **12** (1999) 197–207.

[14] Hoare, C.A.R.: Monitors, an operating system structuring concept. Commun. of the ACM **17** (1974) 549–557; Erratum in Commun. of the ACM **18** (1975) 95.

[15] Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall int., 1985.

[16] Lamport, L.: The temporal logic of actions. ACM Trans. on Programming Languages and Systems **16** (1994) 872–923.

[17] Lehmann, D., Rabin, M.O.: A symmetric and fully distributed solution to the dining philosophers problem. Proceeding of the 8th ACM Symposium on Principles of Programming Languages, 1981, 133–138.

[18] Lehmann, D., Pnueli, A., Stavi, J.: Impartiality, justice and fairness: the ethics of concurrent termination. In *Proc. Internat. Conf. on Automata, Languages and Programming*, LNCS 115 (Springer V., Berlin, etc., 1981), 264–277.

[19] Magee, J., Kramer, J.: Concurrency, state models & Java programs. Wiley (Chichester, etc.) 1999.

[20] A.J. Martin, J.L.A. van de Snepscheut: Design of synchronization algorithms. In: M. Broy (ed.): Constructive Methods in Computing Science. Springer V. 1989, pp. 445–478.

[21] Milner, R.: A Calculus of Communicating Systems. Springer V. 1980 (LNCS 92).

[22] Tanenbaum, A.S.: Modern Operating Systems (2nd ed.). Prentice Hall, New Jersey, 2001