In computer science, the **dining philosophers problem** is an example problem often used in concurrent algorithm design to illustrate synchronization issues and techniques for resolving them.

It was originally formulated in 1965 by Edsger Dijkstra as a student exam exercise, presented in terms of computers competing for access to tape drive peripherals. Soon after, Tony Hoare gave the problem its present formulation.[1][2][3]

Problem statement

Illustration of the dining philosophers problem.

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when they have both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can take the fork on their right or the one on their left as they become available, but cannot start eating before getting both forks.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

Problems

The problem was designed to illustrate the challenges of avoiding deadlock, a system state in which no progress is possible. To see that a proper solution to this problem is not obvious, consider a proposal in which each philosopher is instructed to behave as follows:

- think until the left fork is available; when it is, pick it up;
- think until the right fork is available; when it is, pick it up;
- when both forks are held, eat for a fixed amount of time;
- then, put the right fork down;
- then, put the left fork down;
- repeat from the beginning.

This attempted solution fails because it allows the system to reach a deadlock state, in which no progress is possible. This is a state in which each philosopher has picked up the fork to the left, and is waiting for the fork to the right to become available, vice versa. With the given instructions, this state can be reached, and when it is reached, the philosophers will eternally wait for each other to release a fork.[4]

Resource starvation might also occur independently of deadlock if a particular philosopher is unable to acquire both forks because of a timing problem. For example, there might be a rule that the philosophers put down a fork after waiting ten minutes for the other fork to become available and wait a further ten minutes before making their next attempt. This scheme eliminates the possibility of deadlock (the system can always advance to a different state) but still suffers from the problem of livelock. If all five philosophers appear in the dining room at exactly the same time and each picks up the left fork at the same time the philosophers will wait ten minutes until they all put their forks down and then wait a further ten minutes before they all pick them up again.

Mutual exclusion is the basic idea of the problem; the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources. These issues are studied in the branch of concurrent programming. The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties exemplified by the dining philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems such as operating system

kernels use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

Solutions

Resource hierarchy solution

This solution to the problem is the one originally proposed by Dijkstra. It assigns a partial order to the resources (the forks, in this case), and establishes the convention that all resources will be requested in order, and that no two resources unrelated by order will ever be used by a single unit of work at the same time. Here, the resources (forks) will be numbered 1 through 5 and each unit of work (philosopher) will always pick up the lower-numbered fork first, and then the higher-numbered fork, from among the two forks they plan to use. The order in which each philosopher puts down the forks does not matter. In this case, if four of the five philosophers simultaneously pick up their lower-numbered fork, only the highest-numbered fork will remain on the table, so the fifth philosopher will not be able to pick up any fork. Moreover, only one philosopher will have access to that highest-numbered fork, so they will be able to eat using two forks.

While the resource hierarchy solution avoids deadlocks, it is not always practical, especially when the list of required resources is not completely known in advance. For example, if a unit of work holds resources 3 and 5 and then determines it needs resource 2, it must release 5, then 3 before acquiring 2, and then it must re-acquire 3 and 5 in that order. Computer programs that access large numbers of database records would not run efficiently if they were required to release all higher-numbered records before accessing a new record, making the method impractical for that purpose.[2]

Arbitrator solution

Another approach is to guarantee that a philosopher can only pick up both forks or none by introducing an arbitrator, e.g., a waiter. In order to pick up the forks, a philosopher must ask permission of the waiter. The waiter gives permission to only one philosopher at a time until the philosopher has picked up both of their forks. Putting down a fork is always allowed. The waiter can be implemented as a mutex. In addition to introducing a new central entity (the waiter), this approach can result in reduced parallelism: if a philosopher is eating and one of their neighbors is requesting the forks, all other philosophers must

wait until this request has been fulfilled even if forks for them are still available.

Chandy/Misra solution

In 1984, K. Mani Chandy and J. Misra[5] proposed a different solution to the dining philosophers problem to allow for arbitrary agents (numbered P_1, ..., P_n) to contend for an arbitrary number of resources, unlike Dijkstra's solution. It is also completely distributed and requires no central authority after initialization. However, it violates the requirement that "the philosophers do not speak to each other" (due to the request messages).

1. For every pair of philosophers contending for a resource, create a fork and give it to the philosopher with the lower ID (n for agent P_n). Each fork can either be *dirty* or *clean*. Initially, all forks are dirty.

2. When a philosopher wants to use a set of resources (*i.e.* eat), said philosopher must obtain the forks from their contending neighbors. For all such forks the philosopher does not have, they send a request message.

3. When a philosopher with a fork receives a request message, they keep the fork if it is clean, but give it up when it is dirty. If the philosopher sends the fork over, they clean the fork before doing so.

4. After a philosopher is done eating, all their forks become dirty. If another philosopher had previously requested one of the forks, the philosopher that has just finished eating cleans the fork and sends it.

This solution also allows for a large degree of concurrency, and will solve an arbitrarily large problem.

It also solves the starvation problem. The clean / dirty labels act as a way of giving preference to the most "starved" processes, and a disadvantage to processes that have just "eaten". One could compare their solution to one where philosophers are not allowed to eat twice in a row without letting others use the forks in between. Chandy and Misra's solution is more flexible than that, but has an element tending in that direction.

In their analysis they derive a system of preference levels from the distribution of the forks and their clean/dirty states. They show that this system may describe an acyclic graph, and if so, the operations in their protocol cannot turn that graph into a cyclic one. This guarantees that deadlock cannot occur. However, if the system is initialized to a perfectly symmetric state, like all philosophers holding their left side forks, then the graph is cyclic at the outset, and their solution cannot prevent a deadlock. Initializing the system so that

philosophers with lower IDs have dirty forks ensures the graph is initially acyclic.

Source code example

Below is an implementation of the resource hierarchy solution written in Python.

```
import threading
from time import sleep
import os

# Layout of the table (P = philosopher, f = fork):
#          P0
#       f3    f0
#     P3        P1
#       f2    f1
#          P2

# Number of philosophers at the table.
# There'll be the same number of forks.
numPhilosophers = 4

# Lists to hold the philosophers and the forks.
# Philosophers are threads while forks are locks.
philosophers = []
forks = []

class Philosopher(threading.Thread):
    def __init__(self, index):
        threading.Thread.__init__(self)
        self.index = index

    def run(self):
        # Assign left and right fork
        leftForkIndex = self.index
        rightForkIndex = (self.index - 1) % numPhilosophers
        forkPair = ForkPair(leftForkIndex, rightForkIndex)

        # Eat forever
        while True:
            forkPair.pickUp()
            print("Philosopher", self.index, "eats.")
            forkPair.putDown()

class ForkPair:
```

```
    def __init__(self, leftForkIndex, rightForkIndex):
        # Order forks by index to prevent deadlock
        if leftForkIndex > rightForkIndex:
            leftForkIndex, rightForkIndex = rightForkIndex, leftForkIndex
        self.firstFork = forks[leftForkIndex]
        self.secondFork = forks[rightForkIndex]

    def pickUp(self):
        # Acquire by starting with the lower index
        self.firstFork.acquire()
        self.secondFork.acquire()

    def putDown(self):
        # The order does not matter here
        self.firstFork.release()
        self.secondFork.release()

if __name__ == "__main__":
    # Create philosophers and forks
    for i in range(0, numPhilosophers):
        philosophers.append(Philosopher(i))
        forks.append(threading.Lock())

    # All philosophers start eating
    for philosopher in philosophers:
        philosopher.start()

    # Allow CTRL + C to exit the program
    try:
        while True: sleep(0.1)
    except (KeyboardInterrupt, SystemExit):
        os._exit(0)
```

See also

- Cigarette smokers problem
- Producers-consumers problem
- Readers-writers problem
- Sleeping barber problem

References

1. ^ Dijkstra, Edsger W. *EWD-1000* (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription)

2. ^ [a] [b] J. Díaz; I. Ramos (1981). *Formalization of Programming Concepts: International Colloquium, Peniscola, Spain, April 19–25, 1981. Proceedings* . Birkhäuser. pp. 323 , 326 . ISBN 978-3-540-10699-9.

3. ^ Hoare, C. A. R. (2004) [originally published in 1985 by Prentice Hall International]. "Communicating Sequential Processes" (PDF). usingcsp.com.

4. ^ Dijkstra, Edsger W. *EWD-310* (PDF). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin. (transcription)

5. ^ Chandy, K.M.; Misra, J. (1984). The Drinking Philosophers Problem . ACM Transactions on Programming Languages and Systems.

Bibliography

- Silberschatz, Abraham; Peterson, James L. (1988). *Operating Systems Concepts*. Addison-Wesley. ISBN 0-201-18760-4.
- Dijkstra, E. W. (1971, June). Hierarchical ordering of sequential processes . Acta Informatica 1(2): 115–138.
- Lehmann, D. J., Rabin M. O, (1981). On the Advantages of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem. Principles Of Programming Languages 1981 (POPL'81), pp. 133–138.

External links

- Illustration of the dining philosophers problem (Java applet) at the Wayback Machine (archived March 7, 2013)
- Discussion of the problem with solution code for 2 or 4 philosophers
- Discussion of various solutions at the Wayback Machine (archived December 8, 2013)
- Discussion of a solution using continuation based threads (cbthreads) at the Wayback Machine (archived March 4, 2012)
- Distributed symmetric solutions
- Programming the Dining Philosophers with Simulation
- Interactive example of the Philosophers problem (Java required)

- Satan Comes to Dinner
- Wot No Chickens? – Peter H. Welch proposed the Starving Philosophers variant that demonstrates an unfortunate consequence of the behaviour of Java thread monitors is to make thread starvation more likely than strictly necessary.
- ThreadMentor
- Solving The Dining Philosophers Problem With Asynchronous Agents
- Solution using Actors

Last edited 1 month ago by Oshwah