4.4 Dining philosophers

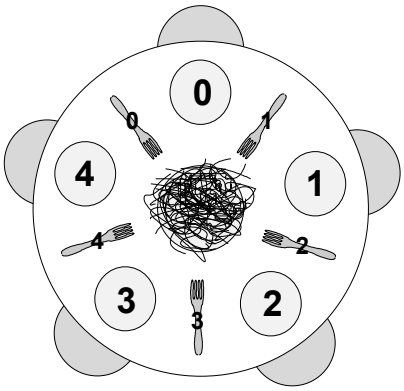
The Dining Philosophers Problem was proposed by Dijkstra in 1965, when dinosaurs ruled the earth [3]. It appears in a number of variations, but the standard features are a table with five plates, five forks (or chopsticks) and a big bowl of spaghetti. Five philosophers, who represent interacting threads, come to the table and execute the following loop:

Basic philosopher loop

```
while True:
    think()
    get_forks()
    eat()
    put_forks()
```

The forks represent resources that the threads have to hold exclusively in order to make progress. The thing that makes the problem interesting, unrealistic, and unsanitary, is that the philosophers need *two* forks to eat, so a hungry philosopher might have to wait for a neighbor to put down a fork.

Assume that the philosophers have a local variable `i` that identifies each philosopher with a value in (0..4). Similarly, the forks are numbered from 0 to 4, so that Philosopher i has fork i on the right and fork $i + 1$ on the left. Here is a diagram of the situation:



Assuming that the philosophers know how to `think` and `eat`, our job is to write a version of `get_forks` and `put_forks` that satisfies the following constraints:

- Only one philosopher can hold a fork at a time.
- It must be impossible for a deadlock to occur.
- It must be impossible for a philosopher to starve waiting for a fork.
- It must be possible for more than one philosopher to eat at the same time.

The last requirement is one way of saying that the solution should be efficient; that is, it should allow the maximum amount of concurrency.

We make no assumption about how long `eat` and `think` take, except that `eat` has to terminate eventually. Otherwise, the third constraint is impossible—if a philosopher keeps one of the forks forever, nothing can prevent the neighbors from starving.

To make it easy for philosophers to refer to their forks, we can use the functions `left` and `right`:

Which fork?

```
def left(i): return i
def right(i): return (i + 1) % 5
```

The `%` operator wraps around when it gets to 5, so `(4 + 1) % 5 = 0`.

Since we have to enforce exclusive access to the forks, it is natural to use a list of Semaphores, one for each fork. Initially, all the forks are available.

Variables for dining philosophers

```
forks = [Semaphore(1) for i in range(5)]
```

This notation for initializing a list might be unfamiliar to readers who don't use Python. The `range` function returns a list with 5 elements; for each element of this list, Python creates a Semaphore with initial value 1 and assembles the result in a list named `forks`.

Here is an initial attempt at `get_fork` and `put_fork`:

Dining philosophers non-solution

```
def get_forks(i):
    fork[right(i)].wait()
    fork[left(i)].wait()

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
```

It's clear that this solution satisfies the first constraint, but we can be pretty sure it doesn't satisfy the other two, because if it did, this wouldn't be an interesting problem and you would be reading Chapter 5.

Puzzle: what's wrong?

4.4.1 Deadlock #5

The problem is that the table is round. As a result, each philosopher can pick up a fork and then wait forever for the other fork. Deadlock!

Puzzle: write a solution to this problem that prevents deadlock.

Hint: one way to avoid deadlock is to think about the conditions that make deadlock possible and then change one of those conditions. In this case, the deadlock is fairly fragile—a very small change breaks it.

4.4.2 Dining philosophers hint #1

If only four philosophers are allowed at the table at a time, deadlock is impossible.

First, convince yourself that this claim is true, then write code that limits the number of philosophers at the table.

4.4.3 Dining philosophers solution #1

If there are only four philosophers at the table, then in the worst case each one picks up a fork. Even then, there is a fork left on the table, and that fork has two neighbors, each of which is holding another fork. Therefore, either of these neighbors can pick up the remaining fork and eat.

We can control the number of philosophers at the table with a Multiplex named `footman` that is initialized to 4. Then the solution looks like this:

Dining philosophers solution #1

```
def get_forks(i):
    footman.wait()
    fork[right(i)].wait()
    fork[left(i)].wait()

def put_forks(i):
    fork[right(i)].signal()
    fork[left(i)].signal()
    footman.signal()
```

In addition to avoiding deadlock, this solution also guarantees that no philosopher starves. Imagine that you are sitting at the table and both of your neighbors are eating. You are blocked waiting for your right fork. Eventually your right neighbor will put it down, because `eat` can't run forever. Since you are the only thread waiting for that fork, you will necessarily get it next. By a similar argument, you cannot starve waiting for your left fork.

Therefore, the time a philosopher can spend at the table is bounded. That implies that the wait time to get into the room is also bounded, as long as `footman` has Property 4 (see Section 4.3).

This solution shows that by controlling the number of philosophers, we can avoid deadlock. Another way to avoid deadlock is to change the order in which the philosophers pick up forks. In the original non-solution, the philosophers are "righties"; that is, they pick up the right fork first. But what happens if Philosopher 0 is a leftie?

Puzzle: prove that if there is at least one leftie and at least one rightie, then deadlock is not possible.

Hint: deadlock can only occur when all 5 philosophers are holding one fork and waiting, forever, for the other. Otherwise, one of them could get both forks, eat, and leave.

The proof works by contradiction. First, assume that deadlock is possible. Then choose one of the supposedly deadlocked philosophers. If she's a leftie, you can prove that the philosophers are all lefties, which is a contradiction. Similarly, if she's a rightie, you can prove that they are all righties. Either way you get a contradiction; therefore, deadlock is not possible.

4.4.4 Dining philosopher's solution #2

In the asymmetric solution to the Dining philosophers problem, there has to be at least one leftie and at least one rightie at the table. In that case, deadlock is impossible. The previous hint outlines the proof. Here are the details.

Again, if deadlock is possible, it occurs when all 5 philosophers are holding one fork and waiting for the other. If we assume that Philosopher j is a leftie, then she must be holding her left fork and waiting for her right. Therefore her neighbor to the right, Philosopher k, must be holding his left fork and waiting for his right neighbor; in other words, Philosopher k must be a leftie. Repeating the same argument, we can prove that the philosophers are all lefties, which contradicts the original claim that there is at least one rightie. Therefore deadlock is not possible.

The same argument we used for the previous solution also proves that starvation is impossible for this solution.

4.4.5 Tanenbaum's solution

There is nothing wrong with the previous solutions, but just for completeness, let's look at some alternatives. One of the best known is the one that appears in Tanenbaum's popular operating systems textbook [12]. For each philosopher there is a state variable that indicates whether the philosopher is thinking, eating, or waiting to eat ("hungry") and a semaphore that indicates whether the philosopher can start eating. Here are the variables:

Variables for Tanenbaum's solution

```
state = ['thinking'] * 5
sem = [Semaphore(0) for i in range(5)]
mutex = Semaphore(1)
```

The initial value of `state` is a list of 5 copies of `'thinking'`. `sem` is a list of 5 semaphores with the initial value 0. Here is the code:

Tanenbaum's solution

```
def get_fork(i):
    mutex.wait()
    state[i] = 'hungry'
    test(i)
    mutex.signal()
    sem[i].wait()

def put_fork(i):
    mutex.wait()
    state[i] = 'thinking'
    test(right(i))
    test(left(i))
    mutex.signal()

def test(i):
    if state[i] == 'hungry' and
    state[left (i)]  != 'eating' and
    state[right (i)] != 'eating':
        state[i] = 'eating'
        sem[i].signal()
```

The `test` function checks whether the ith philosopher can start eating, which he can if he is hungry and neither of his neighbors are eating. If so, the `test` signals semaphore i.

There are two ways a philosopher gets to eat. In the first case, the philosopher executes `get_forks`, finds the forks available, and proceeds immediately. In the second case, one of the neighbors is eating and the philosopher blocks on its own semaphore. Eventually, one of the neighbors will finish, at which point it executes `test` on both of its neighbors. It is possible that both tests

will succeed, in which case the neighbors can run concurrently. The order of the two tests doesn't matter.

In order to access `state` or invoke `test`, a thread has to hold `mutex`. Thus, the operation of checking and updating the array is atomic. Since a philosopher can only proceed when we know both forks are available, exclusive access to the forks is guaranteed.

No deadlock is possible, because the only semaphore that is accessed by more than one philosopher is `mutex`, and no thread executes `wait` while holding `mutex`.

But again, starvation is tricky.

Puzzle: Either convince yourself that Tanenbaum's solution prevents starvation or find a repeating pattern that allows a thread to starve while other threads make progress.

4.4.6 Starving Tanenbaums

Unfortunately, this solution is not starvation-free. Gingras demonstrated that there are repeating patterns that allow a thread to wait forever while other threads come and go [4].

Imagine that we are trying to starve Philosopher 0. Initially, 2 and 4 are at the table and 1 and 3 are hungry. Imagine that 2 gets up and 1 sit downs; then 4 gets up and 3 sits down. Now we are in the mirror image of the starting position.

If 3 gets up and 4 sits down, and then 1 gets up and 2 sits down, we are back where we started. We could repeat the cycle indefinitely and Philosopher 0 would starve.

So, Tanenbaum's solution doesn't satisfy all the requirements.