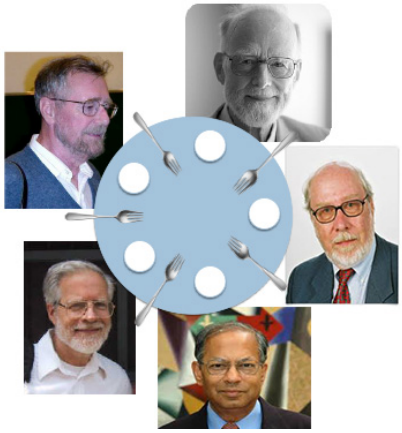
Deadlock

At times, a process will want to obtain a lock on a resource. Once it has the lock, other processes cannot access that resource until the process gives up the lock. A resource could be a data structure that many processes share, like a stack, a tree, a graph, a database. A resource could also be a file on a hard drive.

There's a good reason for allowing a process to do this. If two or more processes could access the resource at the same time, a *race condition* might occur.

Deadlock occurs when each of a bunch of processes has a lock on a resource and needs another resource in order to be able to continue, but that needed resource is locked by another process in the bunch.

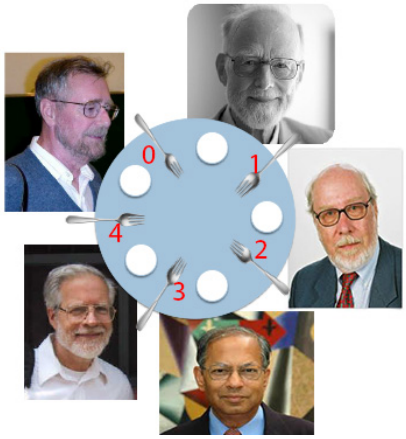
We make this clear with an example developed by Edsger W. Dijkstra, the Dining Philosophers' Problem, shown to the right[1]. Five philosophers sit around the table, alternately thinking and eating spaghetti. Now, eating spaghetti requires two forks, so when a philosopher is tired of thinking and is ready to eat, the philosopher picks up the fork to the left, picks up the fork to the right, eats, puts down the fork to the right, and puts down the fork to the left.



Once, all at the same time, the five philosophers get tired of thinking and want to eat. They all pick up the fork to their left. Then they all try to pick up the fork to the right. But there is no fork to the right because their neighbor to the right has it! So they all wait forever for the fork to their right, until they starve to death. *Deadlock*.

A simple solution to the deadlock problem

There's a simple solution to the deadlock problem, again illustrated with the Dining Philosopher's Problem. Order the resources. In the image to the right, the forks are given numbers in 0..4. A philosopher who wants to eat must follow this protocol:



> Pick up the lower numbered fork.
> Pick up the higher numbered fork.
> Eat.
> Put down the higher numbered fork.
> Put down the lower numbered fork.

Here's a proof that deadlock cannot happen. Suppose some philosopher would like to eat but can't pick up a fork because another philosopher is holding it. We show that another philosopher is either eating or can pick up a fork and eat, so there is no deadlock.

Let h be the highest numbered fork held, say by philosopher p. If h is p's high fork, then, by the protocol, p also holds p's low fork and is eating. If h is p's low fork, then, since h is the highest held fork, p can pick up p's high fork and start eating.

---

[1] The five philosophers pictured here were all interested in programming methodology and programming language design and knew each other well. They shared a passion for simplicity, beauty, elegance, and the mixture of theory and practice. On the upper left is Edsger W. Dijkstra, who did seminal work in compiler writing, operating systems, programming methodology, and much more. Going clockwise, we have Tony Hoare, author of quicksort, the first to define a language in terms of program correctness instead of execution, and a huge contributor to concurrency. Nicklaus Wirth, the designer of *Pascal*, *Algol W*, and *Modula*, also formulated and introduced stepwise refinement as a program development tool in the early 1970s. Jay Misra is most interested in applying formal methods in practice and (among other things) developed the concurrent programming language *Orc*. His only weakness that I've seen in 39 years is not having a beard. The last is David Gries, who wrote this little essay and has had the pleasure of working with these people over the years.

Is it fair?

The protocol may not be fair, because there is a sequence of eating and thinking in which one philosopher never gets a chance to eat. For example, an eating philosopher can put down two forks but quickly pick them up again before a neighbor grabs one. Fairness is different issue, which operating systems have to come to grips with.

Write your own deadlock code

The three classes to the right shows how easy it is to write code that deadlocks.

Class DdLck has two static fields a and b, which are used as resources to be synchronized. Method DdLick.main creates instances of P1 and p2 and calls their start methods.

Thread P1 sychronizes on DdLck.a, prints a message, and attempts to synchronize on DdLck.b.

Thread P2 sychronizes on DdLck.b, prints a message, and attempts to synchronize of DdLck.a.

The output of execution of this application is always

P1 has a
P2 has b

The statements that P1 is finished and P2 is finished never get printed. There is deadlock.

```
public class DdLck {
   public static Integer a= 3;
   public static Integer b= 4;
   public static void main(String[] args) {
      (new P1()).start();
      (new P2()).start();
   }
}

class P1 extends Thread {
   public void run() {
      synchronized (DdLck.a) {
         System.out.println("P1 has a");
         synchronized (DdLck.b) {
         }
      }
      System.out.println("P1 finished");
   }
}

class P2 extends Thread {
   public void run() {
      synchronized (DdLck.b) {
         System.out.println("P2 has b");
         synchronized (DdLck.a) {
         }
      }
      System.out.println("P2 finished");
   }
}
```