

2.5.1 The Dining Philosophers Problem

In 1965, Dijkstra posed and solved a synchronization problem he called the **dining philosophers problem**. Since that time, everyone inventing yet another synchronization primitive has felt obligated to demonstrate how wonderful the new primitive is by showing how elegantly it solves the dining philosophers problem. The problem can be stated quite simply as follows. Five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in Fig. 2-44.

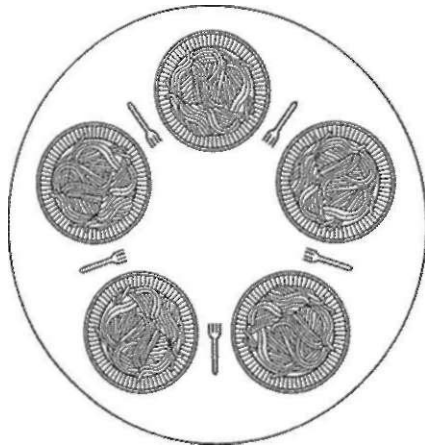


Figure 2-44. Lunch time in the Philosophy Department.

The life of a philosopher consists of alternate periods of eating and thinking. (This is something of an abstraction, even for philosophers, but the other activities are irrelevant here.) When a philosopher gets hungry, she tries to acquire her left and right forks, one at a time, in either order. If successful in acquiring two forks, she eats for a while, then puts down the forks, and continues to think. The key question is: Can you write a program for each philosopher that does what it is supposed to do and never gets stuck? (It has been pointed out that the two-fork requirement is somewhat artificial; perhaps we should switch from Italian food to Chinese food, substituting rice for spaghetti and chopsticks for forks.)

Figure 2-45 shows the obvious solution. The procedure *take-fork* waits until the specified fork is available and then seizes it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                          /* philosopher is thinking */
        take_fork(i);                      /* take left fork */
        take_ork((i+1) % N);               /* take right fork; % is modulo operator */
        eat();                             /* yum-yum, spaghetti */
        put_fork(i);                       /* put left fork back on the table */
        put_ork((i+1) % N);               /* put right fork back on the table */
    }
}
```

Figure 2-45. A nonsolution to the dining philosophers problem.

We could modify the program so that after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too, fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever. A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation. (It is called starvation even when the problem does not occur in an Italian or a Chinese restaurant.)

Now you might think that if the philosophers would just wait a random time instead of the same time after failing to acquire the right-hand fork, the chance that everything would continue in lockstep for even an hour is very small. This observation is true, and in nearly all applications trying again later is not a problem. For example, in the popular Ethernet local area network, if two computers send a packet at the same time, each one waits a random time and tries again; in practice this solution works fine. However, in a few applications one would prefer a solution that always works and cannot fail due to an unlikely series of random numbers. Think about safety control in a nuclear power plant.

One improvement to Fig. 2-45 that has no deadlock and no starvation is to protect the five statements following the call to *think* by a binary semaphore. Before starting to acquire forks, a philosopher would do a down on *mutex*. After replacing the forks, she would do an up on *mutex*. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance bug: only one philosopher can be eating at any instant. With five forks available, we should be able to allow two philosophers to eat at the same time.

The solution presented in Fig. 2-46 is deadlock-free and allows the maximum parallelism for an arbitrary number of philosophers. It uses an array, *state*, to

```

#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N    /* number of i's left neighbor */
#define RIGHT     (i+1)%N      /* number of i's right neighbor */
#define THINKING  0           /* philosopher is thinking */
#define HUNGRY    1           /* philosopher is trying to get forks */
#define EATING    2           /* philosopher is eating */
typedef int semaphore;        /* semaphores are a special kind of int */
int state[N];                /* array to keep track of everyone's state */
semaphore mutex = 1;         /* mutual exclusion for critical regions */
semaphore s[N];              /* one semaphore per philosopher */

void philosopher(int i)      /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {           /* repeat forever */
        think();             /* philosopher is thinking */
        take_forks(i);       /* acquire two forks or block */
        eat();               /* yum-yum, spaghetti */
        put_forks(i);        /* put both forks back on table */
    }
}

void take_forks(int i)       /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);            /* enter critical region */
    state[i] = HUNGRY;       /* record fact that philosopher i is hungry */
    test(i);                 /* try to acquire 2 forks */
    up(&mutex);              /* exit critical region */
    down(&s[i]);              /* block if forks were not acquired */
}

void put_forks(i)           /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);            /* enter critical region */
    state[i] = THINKING;     /* philosopher has finished eating */
    test(LEFT);              /* see if left neighbor can now eat */
    test(RIGHT);             /* see if right neighbor can now eat */
    up(&mutex);              /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

Figure 2-46. A solution to the dining philosophers problem.

keep track of whether a philosopher is eating, thinking, or hungry (trying to acquire forks). A philosopher may only move into eating state if neither neighbor is eating. Philosopher i 's neighbors are defined by the macros *LEFT* and *RIGHT*. In other words, if i is 2, *LEFT* is 1 and *RIGHT* is 3.

The program uses an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy. Note that each process runs the procedure *philosopher* as its main code, but the other procedures, *take-forks*, *put-forks*, and *test*, are ordinary procedures and not separate processes.

2.5.2 The Readers and Writers Problem

The dining philosophers problem is useful for modeling processes that are competing for exclusive access to a limited number of resources, such as I/O devices. Another famous problem is the readers and writers problem (Courtois et al., 1971), which models access to a database. Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it. It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers. The question is how do you program the readers and the writers? One solution is shown in Fig. 2-47.

In this solution, the first reader to get access to the database does a down on the semaphore *db*. Subsequent readers merely increment a counter, *rc*. As readers leave, they decrement the counter, and the last one out does an up on the semaphore, allowing a blocked writer, if there is one, to get in.

The solution presented here implicitly contains a subtle decision worth noting. Suppose that while a reader is using the database, another reader comes along. Since having two readers at the same time is not a problem, the second reader is admitted. Additional readers can also be admitted if they come along.

Now suppose that a writer shows up. The writer may not be admitted to the database, since writers must have exclusive access, so the writer is suspended. Later, additional readers show up. As long as at least one reader is still active, subsequent readers are admitted. As a consequence of this strategy, as long as there is a steady supply of readers, they will all get in as soon as they arrive. The writer will be kept suspended until no reader is present. If a new reader arrives, say, every 2 seconds, and each reader takes 5 seconds to do its work, the writer will never get in.

To prevent this situation, the program could be written slightly differently: when a reader arrives and a writer is waiting, the reader is suspended behind the writer instead of being admitted immediately. In this way, a writer has to wait for readers that were active when it arrived to finish but does not have to wait for readers that came along after it. The disadvantage of this solution is that it achieves less concurrency and thus lower performance. Courtois et al. present a solution that gives priority to writers. For details, we refer you to the paper.