monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = thinking;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != eating) &&
            (state[i] == hungry) &&
            (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }

    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}

Figure 7.22  A monitor solution to the dining-philosopher problem.

    dp.pickup(i);
    ...
    eat
    ...
    dp.putdown(i);

It is easy to show that this solution ensures that no two neighbors are eating simultaneously, and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We shall not present a
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];

    void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = thinking;
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test(int i) {
        if ((state[(i + 4) % 5] != eating) &&
            (state[i] == hungry) &&
            (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
    }

    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}

Figure 7.22 A monitor solution to the dining-philosopher problem.

dp.pickup(i);
...
cat
...
dp.putdown(i);

It is easy to show that this solution ensures that no two neighbors are eating simultaneously, and that no deadlocks will occur. We note, however, that it is possible for a philosopher to starve to death. We shall not present a
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
    ... 
    eat
    ...
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
    ... 
    think
    ...
} while(true);

Figure 7.17 The structure of philosopher i.

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation-free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure 7.17.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next. In Section 7.7, we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).