```
// Dining-Philosophers Solution Using Monitors
monitor DP
{
    status state[5];
    condition self[5];

    // Pickup chopsticks
    Pickup(int i)
    {
        // indicate that I'm hungry
        state[i] = hungry;

        // set state to eating in test()
        // only if my left and right neighbors
        // are not eating
        test(i);

        // if unable to eat, wait to be signaled
        if (state[i] != eating)
            self[i].wait;
    }

    // Put down chopsticks
    Putdown(int i)
    {

        // indicate that I'm thinking
        state[i] = thinking;

        // if right neighbor R=(i+1)%5 is hungry and
        // both of R's neighbors are not eating,
        // set R's state to eating and wake it up by
        // signaling R's CV
        test((i + 1) % 5);
        test((i + 4) % 5);
    }

    test(int i)
    {

        if (state[(i + 1) % 5] != eating
            && state[(i + 4) % 5] != eating
            && state[i] == hungry) {

            // indicate that I'm eating
            state[i] = eating;

            // signal() has no effect during Pickup(),
            // but is important to wake up waiting
            // hungry philosophers during Putdown()
            self[i].signal();
        }
    }

    init()
    {

        // Execution of Pickup(), Putdown() and test()
```

```
        // are all mutually exclusive,
        // i.e. only one at a time can be executing
for i = 0 to 4

        // Verify that this monitor-based solution is
        // deadlock free and mutually exclusive in that
        // no 2 neighbors can eat simultaneously
        state[i] = thinking;
    }
} // end of monitor
```