```
#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 2-47. A solution to the dining philosophers problem.