Solution to Dining Philosophers

```
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 left and right neighbors
             test((i + 4) % 5);
             test((i + 1) % 5);
      }
```

Solution to Dining Philosophers (cont)

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

initialization_code() {
       for (int i = 0; i < 5; i++)
          state[i] = THINKING;
}
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations in the following sequence:

```
Each philosopher invokes the operations

//Thinking

DP.pickup(i);

//Eating

DP.putdown(i);
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
P(mutex);
    …
          // body of F
    …
if (next_count > 0)
  V(next)
else
  V(mutex);
```

Monitor Implementation Using Semaphores

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait() can be implemented as:

```
x_count++;
if (next_count > 0)
    V(next);
else
    V(mutex);
P(x_sem);
x_count--;
```

Monitor Implementation Using Semaphores

- The operation x.signal() can be implemented as:

```
if (x_count > 0) {
    next_count++;
    V(x_sem);
    P(next);
    next_count--;
}
```