Worksheet 5.4: The Dining-Philosophers Problem

Consider the following monitor solution to the **Dining-Philosophers Problem**.

```
monitor DiningPhilosophers {
        enum { THINKING; HUNGRY, EATING) } state [5] ;
        condition self [5];

  void pickup (int i) {
  1             state[i] = HUNGRY;
  2             test(i);
  3             if (state[i] != EATING) self[i].wait;
  }
          void putdown (int i) {
  1             state[i] = THINKING;
  2              test((i+1)%5);
  3              test((i-1)%5);
  }

   void test (int i) {
   printf("\nTesting Phil. %d", i);
1        if ((state[(i + 1) % 5] != EATING) &&
2        (state[i] == HUNGRY) &&
3        (state[(i -1) % 5] != EATING) ) {
4              state[i] = EATING ;
5              self[i].signal ();
  printf("\nPhils. %d and %d not eating", (i+1)%5, (i-1)%5);
  printf("\nPhil. %d can eat", i);
         }
         else {
  printf("\nPhil. %d cannot eat", i);
         }
 }
```

1. Considering the print statements added to the test function, show the output printed by the above code if the following sequence of operations are executed.
   pickup (2);
   pickup (1);
   putdown (2);

   Testing Phil. 2
   Phils. 3 and 1 not eating
   Phil. 2 can eat

   Testing Phil. 1
   Phil. 1 cannot eat

   Testing Phil. 3
   Phil. 3 cannot eat

   Testing Phil. 1
   Phils. 2 and 0 not eating
   Phil. 1 can eat

2. Why does the above solution prevent deadlocks? First describe the scenario that causes a deadlock and then explain why this scenario is impossible with the above monitor solution. (**Limit: 3 lines)**

   Deadlock (DL) happens if **each** phil. picks one chopstick and waits for the other. DL cannot happen here, because a monitor function is a critical section (only one process can be in the monitor at any time). So, a phil. checks both chopsticks in a critical section and eats only if both are available.

3. Which lines in the **test()** method are not needed when **test()** is called from **pickup()**? Explain why each of these lines is not needed. (**Limit: 2 lines each)**

   Line 2 is not needed, because the state of phil. i is already set to HUNGRY in pickup()

   Line 5 is not needed, because phil. i is already active in the monitor and is not waiting on a condition variable. So, there is no need to signal it.

4. The problem solved by the above monitor models 5 processes (philosophers) sharing 5 resources (chopsticks), where each Process i shares a resource with Process (i+1) mod 5 and Process (i-1) mod5. Suppose that we modify the problem to be as follows:
   There are n processes (philosophers) sharing n resources (chopsticks), where each Process i shares k resources with k other processes (k<n). More specifically, Process i shares a resource with Process (i+1) mod n, a second resource with Process (i+2) mod n, …. and a kth resource with Process (i+k) mod n. Each Process i cannot operate (eat) unless it has exclusive access to all **k** resources. Modify the above code to solve this problem. You only need to rewrite the functions that change after identifying them. **Ignore the print statements in this part**.

```
void putdown (int i) {
      if (state[i] != EATING) exit (1);
      state[i] = THINKING;
        for (j= 1; j <= k ; j++)
             test ((i + j) % n);
}

void test (int i) {

       if (state[i] != HUNGRY) return;
      for (j=1; j<=k; j++)
           if (state[(i + j) % n] == EATING)
                  return;

       state[i] = EATING ;
       self[i].signal ();
}
```