Assignment on semaphores:Dining philosopher_problem

Write a C Program to solve Dining philosophers problem using monitors as a solution and implememt this problem using threads.

Dining philosophers problem is a classic synchronization problem.A problem introduced by Dijkstra concerning resource allocation between processes. Five silent philosophers sit around table with a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat.
Eating is not limited by the amount of spaghetti left: assume an infinite supply.
However, a philosopher can only eat while holding both the fork to the left and the fork to the right
(an alternative problem formulation uses rice and chopsticks instead of spaghetti and forks).

Each philosopher can pick up an adjacent fork, when available, and put it down, when holding it.
These are separate actions: forks must be picked up and put down one by one.
The problem is how to design a discipline of behavior (a concurrent algorithm) such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking.

Monitor-based Solution to Dining Philosophers

- Full code for monitor solution (continued on next slide):

```
monitor DP {
  status state[5];
  condition self[5];

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

  Putdown(int i) {
    state[i] = thinking;
    test((i+1)%5);
    test((i-1)%5);
  }
  ... monitor code continued next slide ...
```

- Pickup chopsticks
  - indicate that I'm hungry
  - set state to eating in test() only if my left and right neighbors are not eating
  - if unable to eat, wait to be signalled

- Put down chopsticks
  - 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 signalling R's CV

Monitor-based Solution to Dining Philosophers

```
... monitor code continued from previous
    slide...
  ...
  test(int i) {
    if (state[(i+1)%5] != eating &&
      state[(i-1)%5] != eating &&
      state[i] == hungry) {

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

  init() {
    for i = 0 to 4
      state[i] = thinking;
  }

}  // end of monitor
```

- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

- Execution of Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
  - deadlock-free
  - mutually exclusive in that no 2 neighbors can eat simultaneously