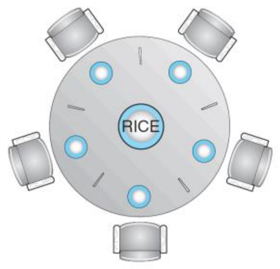
Homework 7

1. Create **n** threads, each sleeps for a random amount of time between 1 and 10 seconds, then displays its sleep time and exits. Prompt the user to input **n**. (1 point)

2. Solve a cross tunnel problem using priority mechanism. There are 10 people who wants to go through a tunnel. Each time only one person can go through the tunnel, and it takes 5 seconds (using **sleep()** to simulate). When one person has crossed, he/she prints the name out. The priority is determined alphabetically (e.g., a person named "abc" has a priority higher than "xyz"). Create 10 people with randomly generated names, and let them go through a tunnel.

3. Solve a single producer, single consumer problem using **wait()** and **notifyAll()**. The producer must not overflow the shared buffer, which can happen if the producer is faster than the consumer. If the consumer is faster than the producer, then it must not read the same data more than once. Do not assume anything about the relative speeds (i.e., random producing and consuming speeds) of the producer or consumer. (1 point)

4. The dining philosophers problem, invented by Dijkstra, is a classic synchronization problem. There are **n** philosophers around a table. These philosophers spend part of their time thinking and part of their time eating. While they are thinking, they don't need any shared resources, but they eat using chopsticks. As philosophers, they have very little money, so they can only afford **n** chopsticks (i.e., the same number of chopsticks as philosophers). These chopsticks are spaced around the table between them. See the following figure for an example with 5 philosophers.



When a philosopher wants to eat, that philosopher must pick up the chopstick to the left and the one to the right. If the philosopher on either side is using a desired chopstick, our philosopher must wait until the necessary chopsticks become available.

No two **Philosophers** can successfully **take()** the same **Chopstick** at the same time. In addition, if the **Chopstick** has already been taken by one **Philosopher**, another can **wait()** until the **Chopstick** becomes available when the current holder calls **drop()**.

In **Philosopher.run()**, each **Philosopher** just thinks and eats continuously, namely each **Philosopher** thinks for a randomized amount of time, then tries to **take()** the right and left **Chopsticks**, eats for a randomized amount of time, and then does it again.

Implement a program that simulates the dining philosophers problem and **will not deadlock**. (2 points)