1

The "Dining Philosophers Problem" is a classic problem in concurrency management first stated by Edsger Dijkstra. Suppose n philosophers (parallel processes) are seated around a circular table. Each philosopher has one chopstick to his or her left and right, thus making a total of n chopsticks. In order to eat from the bowl of noodles at the center of the table, a philosopher must pick up the two adjacent chopsticks, one at a time. Time not spent eating or picking up and putting down chopsticks is spent thinking, and a philosopher can spend an arbitrary amount of time either eating or thinking. A philosopher can thus be summarized by the pseudocode:

```
def philosopher():
    while TRUE:
        think()
        pick_up_chopsticks()
        eat()
        put_down_chopsticks()
```

The issue arises since at most floor($n/2$) philosophers can eat at the same time. If a philosopher attempts to pick up a chopstick already held by a neighbor, the attempt fails; if a philosopher attempts to eat without holding two chopsticks, the attempt also fails. A philosopher who never (or rarely) gets to eat is said to starve, and the goal of the problem is to provide a solution that efficiently ensures that all philosophers eat sometimes. Efficiency means that few excess resources (memory or time) are used in management, and that as little time is spent waiting (e.g. for a chopstick to be released) as possible.

Solutions can be generated using essentially any concurrency management mechanism (semaphores, messages, etc.), either with or without a separate management process (in addition to the philosophers themselves), and either with a single identical program for each philosopher or with differing programs.

1. Provide a working Python implementation that solves the Dining Philosophers Problem using semaphores and the threading library. Pretend the global interpreter lock doesn't exist.

2. Provide a pseudocode implementation that solves the Dining Philosophers Problem using message passing in MPI.

3. Provide a pseudocode implementation that attempts to solve the Dining Philosophers Problem using barriers, but fails in an interesting / nonobvious way. Explain how and why.

Problem 1 continued on next page...

2

One of the trickiest problems in concurrent implementations of data structures is handling simultaneous read and write access. For example, if you're traversing elements in a tree, what happens if another thread changes the tree midway?

Provide a Python implementation of a binary tree class, with an iterator implemented using `__iter__` (and, as a result, `next`). It should work correctly with multiple reader threads, as well as handling multiple reader and writer threads gracefully. That is, if the tree changes during iteration, proceed to the next best location and continue traversal. Again, pretend the GIL doesn't exist.