Kenta Hasui

CS377

Assignment 4 Report

In this assignment, we implemented a solution to the dining philosophers problem using Java's synchronized methods and code blocks to simulate monitors. One thing that I noticed while doing this assignment was that it was a lot less painful than trying to implement it using UPC. The "synchronized" construct had less pitfalls than the locks in UPC. I guess this is because the partial monitor implementation in Java operates at a higher level of abstraction than locks. Once I understood the syntax of both synchronized code blocks and synchronized methods, it was pretty simple to implement.

First, I copied the code from the cs377-examples directory and tried to implement the first solution, using synchronized code blocks and a Chopsticks class. The Chopsticks class and its inner Chopstick class were very simple to implement. For the Philosophers class, it took a bit of time for me to realize that I didn't need the TableMon class at all! Once I trimmed that part of the code away, I was able to see the solution a bit clearer. However, I still needed to implement a deadlock-free simulation. At first, I thought each philosopher needed to check if a chopstick was available within the synchronized code block, so they called wait() and notifyAll() on the chopsticks. I eventually realized that since the code was contained within a block that was synchronized on the individual Chopstick objects, the code only executes if the chopstick was available! So I didn't need to clutter up the solution with unnecessary calls to wait. I did have a question as to whether it was necessary to have a call to notifyAll() at the end of each synchronized block. I didn't think so, as no threads call wait().

For the next implementation, we used synchronized methods and classes representing monitors for a table and a waiter. This was a bit more involved, but still not too difficult. The synchronized methods acted as a condition variable indicating that the waiter was available, or indicating that the table can be accessed (to avoid race conditions). The sitDown() and standUp() methods were elegant and simple to implement, with no busy waiting. It was much less painful than trying to implement the same solution in UPC.