Programming Assignment:
Dining Java Philosophers

CS 377: Parallel Programming
Spring 2019

March 25, 2019

1 Administrative Details

Due: Wed, April 3, 2019

To be handed in: Your printed lab report, and printed source code for both versions.

Comments: Be sure to update the comments to indicate your name and the location of your source file, as well as which version is implemented.

Report: Your lab report will discuss your overall experience solving problems, any problems encountered, how solved, lessons learned, etc.

Starting Code: The starting code is the same code we worked with during the live demo in class on March 25. Copy this starting code from my java-phils directory. There are three Java classes to copy:
`/home/mlsmith/cs377-examples/java-phils/Main.java`
`/home/mlsmith/cs377-examples/java-phils/Philosopher.java`
`/home/mlsmith/cs377-examples/java-phils/TableMon.java`

2 Description

One of the difficult and frustrating things about solving this problem with UPC is that the language provides low level locks and barriers as synchronization primitives. When we attempted to solve the dining philosophers problem using a waiter that seats at most four philosophers at a time, things got a little messy. It would have been nice to have monitors available, which would have simplified things considerably.

Java provides a limited monitor synchronization mechanism. All Java objects have a hidden lock, and the `synchronized` keyword permits us to implement critical sections, as well as a queue to wait on synchronization conditions (i.e., one implicit condition variable). The starting code gives one example of using `synchronized`, along with the `wait()` and `notifyAll()` primitives.

In this assignment, you will gain experience using the two forms of Java synchronization supported by the `synchronized` keyword: synchronized methods and synchronized code blocks. Examples of synchronized methods abound in `TableMon.java`. As a reminder, the form of a synchronized code block is as follows:

```
synchronized ( obj ) {
    // A thread executing in this block holds the hidden lock
    // of object obj.  This is a critical section!
    // Thus, two or more objects can synchronize with one
    // another by sharing a common object, and using
    // synchronized code blocks.
}
```

3 Assignment

Your mission is to implement two additional versions of the dining philosophers in Java. One version will be monitor-like, using synchronized methods; the other version will use synchronized code blocks and shared objects among the dining philosophers.

synchronized code blocks. In this version, you will use synchronized code blocks and synchronizing objects to represent the chopsticks. You will implement a Chopsticks class that contains an array of Chopstick objects to represent the chopsticks. You may make Chopstick an inner class of Chopsticks if you wish. In addition to the array of Chopstick objects, provide two methods in class Chopsticks: `getLeft(int id)` and `getRight(int id)`. Note: these methods need not be synchronized, as they merely return the reference to the appropriate chopstick for the given philosopher id. These methods will be used in the Philosopher class in the lifecycle of the philosopher to create synchronized code blocks. The basic idea is this:

```
synchronized (chopsticks.getLeft(id)) {
    // philosopher has left chopstick!
    synchronized (chopsticks.getRight(id)) {
        // philosopher has right chopstick!

        // don't just hold them---eat!
    }
}
```

Using these synchronized code blocks, implement a deadlock free solution to the dining philosophers (i.e. pick up chopsticks safely!). The three (or four) main Java classes for this solution are: Main, Philosopher, and Chopsticks (which should include the inner class Chopstick).

synchronized methods. In this version, you will implement the waiter solution to the dining philosophers. This time, you have monitor capability! Add a WaiterMon class to the starter code, and modify your Philosopher objects to include a reference to a waiter monitor, in addition to the table monitor. The WaiterMon object should have a single instance variable to keep track of the number of philosophers currently seated. Provide two synchronized methods: `sitDown(int id)` and `standUp(int id)`. These methods should increment and decrement the count, as appropriate. If there is no room at the table, `sitDown(int id)` should wait. Similarly, `standUp(int id)` should notify any waiting philosophers that there's room at the table.