

## Programming Assignment #3 – Due Mon, 5 Nov

### Problem Statement:

This programming assignment gives you the opportunity to do some simple concurrent object-oriented programming using *threads*. I recommend using Java, but if you prefer, you can choose to do it in C++ using POSIX threads (e.g., use the `Sync.h` class on the course programming assignment page). Extra credit will be given for doing this assignment in BOTH Java and C++ (with POSIX threads).

### Dining Philosophers

A common problem in concurrent programming is coordinating access to a shared resource by enforcing a *synchronization condition* between multiple concurrent processes. A classic example is the **Dining Philosophers** problem, which consists of  $n$  philosophers seated at a round table thinking and eating concurrently. For this version of the Dining Philosophers problem, they are eating from a shared bowl in the middle of the table using a pair of chopsticks. Each philosopher has a left chopstick and a right chopstick, so there are  $n$  chopsticks in total. In order to eat, a philosopher must obtain both left and right chopsticks exclusively with the order determined by a coin-toss. If heads, the philosopher picks up the left then the right chopstick; if tails, pick up the right then the left chopstick. If a chopstick is not available a philosopher must wait for it to be released by his/her neighbor before being allowed to eat. A philosopher is allowed to eat for no more than 1 second before relinquishing both chopsticks to think again, so as not to starve the other philosophers. After a random number of milliseconds of thinking, a philosopher may try to eat again but must first re-acquire both chopsticks. Below is a fragment of a `Philosopher` class that inherits the `java.lang.Thread` class and shows the order in which to think, acquire chopsticks, eat, and release chopsticks:

```
public class Philosopher extends Thread {
    private String name;
    private Chopstick left, right;

    // each Philosopher is assigned an integer id and two chopsticks
    public Philosopher (int id, Chopstick c1, Chopstick c2) {
        name = "Philosopher-" + id;
        left = c1; right = c2;
    }

    // the run method is invoked by calling the start() method on
    // an instance of the Philosopher class, e.g., phil.start()
    public void run () {
        while (true) { // loop forever
            System.out.println(name + " thinking");
            // think for random number of milliseconds <= 1 sec
            // coin-toss & acquire left-right/right-left chopsticks
            System.out.println (name + " dining");
            // eat for random number of milliseconds <= 1 sec
            // coin-toss & release left-right/right-left chopsticks
        }
    }
}
```

In addition to the `Philosopher` class, you will need a `Chopstick` class for enforcing *synchronized* access to each chopstick. A philosopher acquires a chopstick when it is available, or waits. At no time may two philosophers hold the same chopstick. When finished eating, a philosopher releases both chopsticks in the order determined by another coin-toss.

Your main program must create the `Chopstick` objects and the `Philosopher` objects and assign to each `Philosopher` a pair of `Chopsticks` (modulo  $n$ ), one shared with the `Philosopher` on the left and one with the `Philosopher` on the right. Create each `Philosopher` thread and start it running in your main

procedure. Run the program for at least 3 minutes making sure to let each execution run long enough so that each Philosopher gets to eat several times before terminating the program (e.g., using ctrl-C on Unix/Linux, ctrl-Z on Windows). On termination, the program must report statistics on a) how many times each philosopher got to eat and b) how long (in milliseconds) each philosopher spent eating and thinking. Use these statistics to verify that neither *starvation* nor *deadlock* nor *unfairness* occurs for any Philosopher. Starvation means some philosopher doesn't get to eat and unfairness means some philosopher is denied a similar amount of eating time to his/her colleagues. If starvation or deadlock or unfairness occurs you must fix your program so that they do not occur.

### **Implementation Notes:**

1. In Java, a thread sleeps by calling the **Thread.sleep()** method passing an integer value representing the number of *milliseconds* to sleep. The easiest thing to do is write a `randomSleep()` function as follows, which also handles an exception that could be thrown by the `Thread.sleep()` function. **Math.random()** returns a double value between 0.0 and 1.0, so it must be multiplied by 1000 and converted to an integer value.

```
int randomSleep () {
    int ms = (int)(1000 * Math.random());
    try { Thread.sleep(ms); }
    catch (InterruptedException e) { /* ignore it */ }
    return ms;
}
```

2. Since the program is terminated on a ctrl-C interrupt from the command line, in order to get statistics printed, it is necessary to create a "statistics" class as a subclass of the class Thread, and register it to be run as a "shutdown hook" with the Java virtual machine. To do this in Java, you first create an object of the statistics class then register it with Java Virtual Machine's run-time:

```
Stats s = new Stats();
Runtime.getRuntime().addShutdownHook(s);
```

### **Submission Requirements:**

You are to hand in at the start of class on the due date a source code listing of your program printed with Enscript and with your name and comments included. **We will test your program for n = 2, 5, and some random number >10 and <10,000 philosophers so you should test that many as well.** You are also to submit an electronic copy of your source program using the **turnin** program on CS Department machines **BEFORE THE START OF CLASS ON THE DUE DATE**. Late assignments will not be accepted.

### **Honor Policy:**

This assignment is a personal learning opportunity that will be evaluated based on your ability to think independently, work through a problem in a logical manner and implement a software program on your own. You may however discuss verbally or via email the general nature of the conceptual problem to be solved with your classmates, the course TA or the course instructor, but you are to complete the actual programming for this assignment without resorting to help from any other person or other sources that are not authorized as part of this course. If in doubt, ask the course instructor. **Any indication of improper cooperation or the use of non-approved course materials (e.g., solutions of this program from a previous semester or from the web) will result in a grade of zero being assigned for this project and referral to the College for possible disciplinary action with respect to your continued participation in this course. By submitting a solution to this assignment under your name, you are asserting that the solution was done individually by you under this class honor policy.**