CSCI 315 Lab 6 Exercise

March 8, 2010

Objectives: In this lab you will work again with semaphores and monitors. In particular, you will use semaphores to develop two solutions to the dining philosopher's problem and thus gain a better understanding of resource allocation problems such as deadlock, starvation, etc.

Prelab: It is important that you read Section 6.6.3 of your text about the Dining Philosophers Problem **before** you come to lab. In this problem, five philosophers are sitting around a circular table. They spend their time alternating between thinking and eating. The problem comes in how to allocate the eating utensils to the philosophers (there are 5 utensils and each philosopher needs two to eat).

For the prelab assignment, you are to write a Java program that simulates the unconstrained version of this problem, i.e., a situation in which the philosophers use no utensils as in a pie-eating contest. You need to write two classes:

- A `Philosopher` class. `Philosopher` objects will be run as threads. Each object has one integer data member that keeps track of which philosopher it is; this data member stores the number of the `Philosopher`. The value is passed as an argument to the constructor. The class should contain a method similar to the `napping()` method used in parts of Lab 5 (e.g., in the BoundedBuffer class of the producer-consumer exercise). This method will take an integer parameter `t`, and cause the thread to sleep for a random amount of time between 0 and `t` seconds. The `run()` method of the `Philosopher` class should consist of a loop that repeats the following actions:
  - Prints a message that Philosopher i is thinking.
  - Calls the `napping()` function with argument 2.
  - Prints a message that Philosopher i is hungry.
  - Prints a message that Philosopher i is starting to eat.
  - Calls the `napping()` function with argument 1.
  - Prints a message that Philosopher i is done eating.
- A `Dining` class. This class will declare and start five `Philosopher` threads numbered 0 through 4. Note that if you declare an array of `Philosopher` objects, you can simplify your code a bit by using a loop to start each `Philosopher` thread in turn.

Write the Java code for this problem, and run it briefly to test it. While I am not going to require you to hand this code in at the start of lab, you will find that you may have difficulty in completing the exercises during the lab period if you do not write this code prior to lab.

Problems:

1. **Dining Philosophers - A Resource Restriction:** Before you start, here is some advice: For each of the exercises below, you should create a new directory for the code of that solution. This will make it possible to reference code from a previous solution.

   The Dining Philosophers problem in its standard form is a resource allocation problem. As in the version you implemented in the prelab, it consists of five philosophers sitting around a table, thinking and eating. However, to make it more interesting, our philosophers are going to be eating with chopsticks. Being philosophers, they are not nearly as well-paid as computer scientists, and so can only afford five chopsticks. One of these chopsticks is placed between each pair of philosophers around the table. To eat, a philosopher must have two chopsticks: the one to his or her left, and the one to his or her right. Note that this means that it is not possible for two neighboring philosophers to eat at the same time.

   Implement a solution using the `Semaphore` class from last lab. In particular, implement each chopstick as a mutual exclusion semaphore; the `Philosopher` thread that successfully passes through any such semaphore is assumed to have picked up the corresponding chopstick. A `Philosopher` must pick up both chopsticks in order to be able to eat. This means that you can think of the "eating" section (printing the message indicating that eating has begun, the call to `napping()` that simulates the eating time, and the message indicating that eating is finished) as the critical section of the `Philosopher` code.

   Your implementation should follow the following guidelines, which are consistent with the solution in the text:

   - You should declare an array `chopsticks[]` of five `Semaphore` objects in the `Dining` class. Initialize each entry of `chopsticks[]` with a mutual exclusion (mutex) `Semaphore`.
   - Modify the `Philosopher` class so that it has a data member that holds an array of `Semaphore` objects. The `chopsticks[]` array declared in `Dining` will be passed in as an argument to the constructor and assigned to this data member.
   - Include `acquire()` and `release()` calls in the `Philosopher run()` method to simulate picking up and putting down chopsticks. You should use the following scheme for numbering the chopsticks: the chopstick to the left of Philosopher i is numbered i, while the chopstick to the right is numbered $(i+1) \pmod 5$ (remember that Philosopher 0 is to the right of Philosopher 4).

   Compile and run the resulting program. You should observe that no two consecutive philosophers are eating at the same time. **Hand in your revised code, a test run of this program, and your answers to the following questions** (be sure to number your answers here and in later problems with the number of the exercise): Do you observe any problems when your program runs? Based on the code, what problems could occur?

2. **Dining Philosophers with Deadlock:** The standard solution can potentially deadlock. You will try to simulate the deadlock problem in this exercise. Look closely at your code. Where can a deadlock problem occur? Remember that deadlock involves an actual sequence of events (as

opposed to some static sequence of statements in the code). As in last lab, you will insert calls to `napping()` in your `Philosopher` code to encourage deadlock to occur. Experiment with where to insert the `napping()` call and reasonable nap times until you get test runs that deadlock. Add a message after each `acquire()` call saying "Philosopher i picking up chopstick j", and one after each `release()` saying "Philosopher i putting down chopstick j" to help you see what is happening.

Note: Once you can cause deadlock to happen, it's a good idea to run the program a number of times to see what happens. With a good choice of when to nap and napping time, you'll see that sometimes your program deadlocks, and sometimes it will run for a long time without deadlocking. This shows that deadlock is the result of a particular series of events; having the potential for deadlock in your code is not a guarantee that it will occur. This can make debugging a real pain.

Hand in your revised code and a test run of this program showing an occurrence of deadlock along with the answer to the following question: What situation do you observe occurring that leads to the deadlock?

3. **Dining Philosophers - Breaking Symmetry:** The deadlock problem in the previous exercise occurs because all the philosophers manage to pick up their left chopstick. Then no philosopher can pick up his or her right chopstick, because some other philosopher already has it. The symmetry in the solution - each philosopher picks up a left chopstick, then a right chopstick - leads to a deadlock. More generally, symmetric behavior occurs when similar objects always make the same choices. In some cases this isn't a problem. Other cases it will cause a deadlock.

   Problems with symmetrical behavior are common in computer science. It is a standard issue that one needs to think about in dealing with parallel algorithms where multiple processes or threads act on a set of common resources, for example.

   How do we break symmetry? In parallel algorithms, a common technique is to apply randomness at the point of symmetry. For example, separate processes can each flip a coin (or the computational equivalent), then decide what to do based on the result. We could do this in the Dining Philosophers solution, and greatly decrease the probability of having deadlock occur. (In this case, each philosopher would flip a coin to decide which chopstick to pick up first.) Unfortunately, the use of randomness would still allow the possibility of choosing symmetric behavior, though with a small probability. When we are dealing with making choices to help a parallel algorithm proceed a little faster, this isn't a problem. But with the possibility of deadlock, we need to be more careful.

   One way to break symmetry in the Dining Philosophers is to let different `Philosopher` objects behave differently. There are a couple of obvious possibilities:

   - Each `Philosopher` object can check its number. If the number is odd, it picks up its right chopstick first; if it is even, it picks up its left chopstick first.
   - Each `Philosopher` object picks the lowest-numbered chopstick it needs first.

   Implement **both** of these solutions in your program (leave in the `napping()` calls that produced the deadlock, however). That is, implement both ways

of deciding which chopstick to pick up first and comment one out. Compile and run each of your solutions a couple of times to convince yourself that they run without deadlocking.

Hand in your revised code for both of these solutions and a test run of each. Also hand in answers to the following questions: Do these solutions solve all the potential problems? If not, what problem can still occur?

4. **Dining Philosophers - A monitor solution:** In the exercises so far you have used the semaphore as the protection and synchronization mechanism. Now you will fabricate a solution to the dining philosophers problem using the monitor mechanism in Java. Your solution will follow the structure described in Section 6.7.2 on pages 268-270 of the text. Note that the code in the text is not Java code. Here are some pointers:

   - Implement a class `DiningPhilosophers` along the lines of the solution on page 269. You will need to incorporate the keyword `synchronized` into some of the methods, and to make additional changes.

     The `DiningPhilosophers` class provides the synchronization and protection for the philosophers and their chopsticks via the `takeForks` and `returnForks` methods. The `test` method is a key element.

     In the code on page 269 of the text, both the `takeForks` and `test` methods involve a notion of *self*. Think carefully how you would implement this in Java. Do you need to do anything explicit? How do you make a thread *wait* if the philosopher is not in the *eating* state? Do you need an explicit representation of *self*? Also note that *signal* in the `test` method is really a notification; which form (`notify()` or `notifyAll()`) do you need here?
   - Implement a class `Philosopher` as you have done in the other problems. In this problem, though, you want to make use of the sychronization provided by the `DiningPhilosophers` object. Hence it is necessary to instantiate a `DiningPhilosophers` object in the `main` method and then pass it to each philosopher object. This will give each philosopher access to the `takeForks` and `returnForks` methods. Modify the `run` method appropriately.
   - The `Dining` class can remain as in the previous problems except that there must be a `DiningPhilosophers` object instantiated. Also, be sure that you pass an integer value to each `Philosopher` object so it can interact appropriately with the `DiningPhilosophers` object.
   - The final step is to put appropriate output statements to indicate what is happening to each philosopher. This should signal transitions from one state to another and also indicate any extended waiting – i.e., if the philosopher wakes up but still can't get a chopstick, print a message.

   Hand in the code for the three classes along with output from a sample run. Also hand in a brief explanation showing that this solution is deadlock free.

Hand in: Hand in the specified code, test runs, and answers to the questions by **Wednesday after the break** (note the extended due date).