CSS430  
Martin Metke  
2017/07/23

Program 3 – Dining Philosophers

# Investigating Java Synchronization

Investigate the following options for solving this problem.  For each of the following put write a brief comment about how you could use it to solve the problem, along with a short list of pros & cons for each one.

## synchronized / wait / notify / notifyAll

The Java “synchronized” keyword marks a code block as being executable by only one thread at any one time, for each instance of the object that contains it or that it points to. This essentially turns the marked object instance into a monitor with only one Condition: the marked code block itself. Wait, notify, and notifyAll are the thread control functions that cause threads which are waiting on updates to the held object instance to sleep or awaken.

The Dining Philosophers problem could be solved using five Fork objects that have synchronized takeFork() and returnFork() methods; any DiningPhilosopher could call takeFork() on the Fork objects to its right and left, but would have to sleep until it was notified that the Fork object in question was no longer held by another thread.

The benefit of using synchronized is that Java provides all of the thread-safe functionality natively, and utilizing this functionality is clean and simple. The drawbacks include that it requires synchronizing threads on multiple separate Fork instances, and that this approach will require nested calls to wait() and notify() in order to prevent starvation/deadlocks when one thread holds one Fork and has to wait for the other.

## Counting semaphores

A counting semaphore allows N threads to acquire some resource where “N” is the number of instances of that resource that are available. In the case of the DiningPhilosophers problem, each fork can be represented by a resource that two Philosophers contend for.

In essence, the single-count semaphores that would be used to implement a solution to the DiningPhilosophers problem would mimic mutexes. On the plus side, this approach can be used to prevent deadlocks and can be combined relatively simply with starvation-prevention steps. On the negative side, this solution requires a thread to gain locks on both the left and right forks before “picking up” either, so there must be as many semaphores as there are forks (and correct handling of at least two semaphores for each thread). It also makes the Philophers themselves responsible for proper synchronization, rather than providing a simple object interface to handle synchronization externally.

## Reentrant locks

A reentrantLock allows for unstructured locking with features such as timeouts, testing hold counts, immediate return when the executing thread already holds the lock, and more. Functionally it is similar to a Java *synchronized* monitor, but does not require that the exclusive execution block be prefaced by the *synchronized* keyword and has the capacity to avoid starvation innately (using the *fairness* parameter) by prioritizing long-waiting threads.

Pros for the reentrantLock approach are: clean design; deadlock and starvation avoidance. Cons include: decreased (possibly *significantly* decreased) performance; makes the Philosopher threads responsible for locking; requires one lock per fork.

# How My Solution Works

Explanation of solution and source material.

## Algorithm

The assignment requires using a monitor-like class to control thread synchronization in this implementation of the Dining Philosophers problem. To that end, I implemented DiningPhilosophers.java as a Monitor using a ReentrantLock and a set of Conditions defined on it. While this may be overkill for the requirements of this assignment, the instructions specifically required \*not\* using Semaphores, and if necessary the ReentrantLock can be made “fair” for FIFO access to the Conditions. I chose not to implement the Monitor using a “fair” ReentrantLock, however, due to the possible significant performance drawbacks of doing so.  
 The basic design of my DiningPhilosophers class is derived from the original file, extending it in compliance with the test files and instructions. Aside from the constructor and two methods to report the current state of an instance, I implemented the takeForks and returnForks methods.

TakeForks: this method uses the ReentrantLock “lock” to ensure mutual exclusion. It then:

* 1. Has a thread attempt to take the forks to the specified Philosopher’s left and right.
  2. Makes the thread wait on one or both of the “fork” Conditions if the forks are not currently available.
  3. Counts how many times the thread has awoken and waited (to prevent starvation in cases of contention).
  4. Handles interrupts by continuing to check, and if necessary wait on, the forks.
  5. Marks the desired forks as unavailable and sets the Philosopher’s state to EATING.
  6. Finally, unlocks the critical section.

So once a thread enters “takeForks”, it \*must\* acquire the forks to the specified Philosopher’s right and left before it leaves the method.

returnForks: similarly, this method uses the ReentrantLock “lock” to ensure mutual exclusion. A thread running this method then:

1. Verifies whether the forks to the left and right of the specified Philosopher were actually held, and if so, marks them as available.
2. Signals any thread waiting on the left or right fork to resume
3. Finally, unlocks.

## Sources

1. CSS430 class notes
2. *Siblerschatz, Galvin, Gagne* Operating System Concepts with Java, 8th edition. 2010
3. *Baptiste Wicht*, “Java Concurrency – Part 5 : Monitors (Locks and Conditions)”. 2010-09-06  
   <https://baptiste-wicht.com/posts/2010/09/java-concurrency-part-5-monitors-locks-and-conditions.html>  
   *Accessed 2017/07/26 1330 hours*
4. *Oracle and affiliates,* “Java Platform Standard Ed. 7 - Class ReentrantLock”. 2016  
   <https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html>  
   *Accessed 2017/07/25 2000 hours*
5. *Chinaxing*, “Why lock condition await must hold the lock”, 2015-09-25 1538 hours  
   <https://stackoverflow.com/questions/32774399/why-lock-condition-await-must-hold-the-lock>  
   *Accessed 2017/07/26 1715 hours*