CptS 483/580

Homework #4 – First programming assignment

Assigned: 2/9/11

Due: 2/18/11 11:59:59PM. Turn in to the course turn-in page
http://www.eecs.wsu.edu/~hauser/cs483/turnin

In this assignment you will create several classes that implement synchronizers described in the book in section 5.5. The directions for each class provide the class name and file name that you are to use for that class. When finished, put all of the source code in a zip file (that's a zip file, not tar, not bzip, not jar: a zip file) and submit it to the turn-in page. As always, you may turn in the code as many times as you like. Only the last will be graded.

The purpose of this assignment is to gain experience thinking about how threads can be synchronized with each other. Thus, you may not use any library classes that essentially implement any of the assigned classes. You may, however, use library classes that provide synchronization capabilities beyond what is provided by intrinsic synchronization, such as the `Atomic<X>` classes, lock classes, etc.

Each implementation must contain enough class-level comments to explain *how* it works as well as per-method comments explaining *what* each method does. Include additional comments at the block or line level to help me understand your implementations.

1. `Semaphore (Semaphore.java)`

Semaphore semantics are described in section 5.5.3. Implement a constructor providing the initial number of permits, and public methods for `acquire(int n)` and `release(int n)`. Overloaded parameterless methods `acquire()` and `release()` correspond to passing 1 to the parameter-ful versions.

2. `CountDownLatch (CountDownLatch.java)`

CountDownLatch is described in section 5.5.1. The constructor initializes a counter to the provided value. The `countdown()` method decrements the counter (by 1); the `await()` method returns when the counter is no longer positive.

3. `CyclicBarrier (CyclicBarrier.java)`

CyclicBarrier is described in section 5.5.4. The constructor initializes a counter. The `await()` method decrements the counter and waits until the counter reaches zero at which point all waiting threads are released, the counter is reset to its initial value, and subsequent calls to await again block until the counter reaches 0. Undergrads: you do not need to implement the handling of timeouts and interruptions described in the text; Grads should do a full implementation. Nobody needs to implement the barrier action feature described in the first line of p. 101.) Note: my Race program for assignment 1 contained a barrier implementation but it does not fully meet the re-usability requirements for this assignment. You may use it as a starting point if you wish.

Assignments will be graded on: correct functioning, correctly described in comments. Test code is not provided – you're welcome to write your own but as we've seen, testing can only tell us that code is wrong, not that it is correct, and may have limited utility for concurrent programs. For any concurrent program you must be able to construct a correct argument about how your

code correctly implements the desired functionality—and that's what I expect to see in the comments to your code.