CMSC433: Project 1
Grading explanations

1. Grading Breakdown:

70pts Public/Secret Tests
15pts Concurrency Control
10pts Design Choices
5pts Design Document
110pts Total

2. Design Document:

- Did not submit one [-5]
- Missing information [-2]
    - Failed to explain which situations ImperativeGraph performs better than PersistentGraph and vice-versa
- Longer than one page [-1]

3. Design Choices:

Efficiency of your design

3.1 Canonical Design

- ImperativeGraph uses one or two Hash structures for storing nodes and edges, which is guarded by a single lock on *this* for each method.
    - Bonus points if your implementation allowed for higher parallelism [+5]
        - Using two data structures and synchronizing on them individually. However, this made you more likely to lose points for deadlocks. (see 4.2)
        - Implementing ImperativeGraph using PersistentGraph or ImmutableLists, which allows you to release locks earlier when iterating by first copying the list you want to iterate.
- PersistentGraph uses ImmutableLists to store nodes and edges. No synchronization is needed here because there are no writes to the graph.
- The HashMap in the Edge class needs to be synchronized during create() (see 4.1)

3.2 Poor Design

Correct, but inefficient

- Putting data structures in the Graph class. [-2]
    - Doing this requires you to synchronize the isEmpty(), outDegree(), and inDegree() methods in order to guarantee thread-safety for a ImperativeGraph, but this is inefficient for

PersistentGraph. A more efficient way to implement these three functions is by using the iterative functions of each graph type with a node counting function. As long as the iterative functions are thread-safe, you should not need to synchronize in the Graph class.

- Poor algorithmic choices [-1 each]
    - Counting both nodes and edges for isEmpty() when you should only need to check nodes
    - For equals(), doing containment checks both ways as opposed to doing a one-way check and a size check
- Unnecessarily creating separate objects for locking instead of using intrinsic locks [-1]
- Unnecessary deep copies [-3]
    - Deep copying when iterating over a PersistentGraph
- Mild Oversynchronization [-2 each conceptual occurrence]
    - Synchronizing on thread-local data structures
- Severe Oversynchronization [-5]
    - Using locks in PersistentGraph
    - Using a static global lock in ImperativeGraph instead of per-instance locks

4. Concurrency Control

Thread-Safety

4.1 Undersynchronization [-2 each, max -10]

- Failing to synchronize the Edge class HashMap in create(). There is a data race when executing the if-then check since threads can still switch context in between the *if* and *then* statements. Using a ConcurrentHashMap does not fix this problem because it only synchronizes for the duration of an individual method call.
- Using the Guarded-by pattern incorrectly or not at all
    - Using separate node/edge list locks when adding/removing nodes/edges, but synchronizing on *this* when copying or clearing.
    - Synchronizing on node and edge locks when using data structures representing outgoing/incoming edges
- Failing to synchronize on the graph being compared against during ImperativeGraph equals(), since the other graph can still be modified while you are comparing against it.
- Atomicity violations
    - Releasing the node lock too early in addEdge() can allow iterators to see an invalid/incorrect graph state

4.2 Deadlocks [max -5]

- Obvious deadlocks [-5]
    - Not acquiring locks in a consistent order, i.e. synchronizing on nodeLock then edgeLock

in one method while getting the edgeLock then nodeLock in another.
- Subtle deadlocks [-2]
    - Only synchronizing on edges while iterating. Suppose you give iterEdges() a function that calls addNode(). This thread will now acquire the edge lock first before acquiring the node lock, possibly causing a deadlock with any other thread trying to acquire the node lock before the edge lock.

5. Secret Tests

5.1 PersistentGraph Tests

- testEmptyGraph():
    - gets the empty PersistentGraph and checks that isEmpty() is true
    - adds some nodes and checks that the result is not empty
- testEquals():
    - tests the equals() methods for PersistentGraphs
    - empty graph should equal itself
    - simple graph with some nodes and edges should equal itself
    - removing a node from a graph should make it not equal to the original graph
    - no graph should equal an object that is not a graph
    - graph "A -> B, A -> C" should not equal graph "A -> B, C -> A"
    - graph "A -> B, A -> C" should not equal graph "A -> B, A -> D"
- testEmptyAction():
    - tests various actions that should do nothing such as:
    - removing nodes and edges from the empty graph
    - count of nodes and edges on the empty graph should be 0
    - removing nonexistent nodes and edges from a simple graph
    - count of successors and predecessors of a nonexistent node should be 0
- testImmutable():
    - tests that addNode, addEdge, removeNode, removeEdge, iterPredecessors, iterSuccessors, iterNodes, iterEdges do not mutate the graph
- testMT1():
    - four threads are handed the same PersistentGraph reference and perform some operations to create their own local copies, make sure the four graphs come out as expected
- testMT2():
    - three threads perform operations on the same graph reference, make sure the graph is not mutated

5.2 ImperativeGraph Tests

Graph used for most of these tests:

Nodes: [0, 1, 2, 3, 4, 5, 6, 7]

Edges: [0->1, 1->2, 2->3, 3->4, 4->5, 5->6, 6->0, 6->1, 6->2, 6->3, 6->5]

- testSSTEdges1():
    - removing nodes/edges from an empty graph
    - removing non-existent nodes/edges
    - removing nodes also removes edges
- testSSTEdges2():
    - edges are only added once
    - bidirectional edges
    - adding an edge with exactly one existing node
- testSSTEquals():
    - empty graphs
    - adding the same edge to equal graphs
- testSSTCopyClear():
    - modifying a copy does not modify the original
    - cleared graph equals a new graph
- testSSTIter():
    - checks iterSuccessors() and iterPredecessors()
    - iterating an empty graph
- testSSTEmpty():
    - removing all nodes/edges results in an empty graph
- testSSTDegree():
    - checks inDegree() and outDegree() of all nodes
- testSMTPC():
    - checks for data races between addEdge() and removeNode(): edges must always have valid nodes
- testSMTDR():
    - checks for addEdge() atomicity: make sure nodes are not added twice