2.1 Lecture Summary

2.1 Critical Sections

Lecture Summary: In this lecture, we learned how *critical sections* and the *isolated* construct can help concurrent threads manage their accesses to shared resources, at a higher level than just using locks. When programming with threads, it is well known that the following situation is defined to be a *data race* error — when two accesses on the same shared location can potentially execute in parallel, with least one access being a write. However, there are many cases in practice when two tasks may legitimately need to perform concurrent accesses to shared locations, as in the bank transfer example.

With critical sections, two blocks of code that are marked as isolated, say A and B, are guaranteed to be executed in mutual exclusion with A executing before B or vice versa. With the use of isolated constructs, it is impossible for the bank transfer example to end up in an inconsistent state because all the reads and writes for one isolated section must complete before the start of another isolated construct. Thus, the parallel program will see the effect of one isolated section completely before another isolated section can start.

Optional Reading:

1. Wikipedia article on Critical Sections.

2. Wikipedia article on Atomicity.

2.2 Lecture Summary

2.2 Object-Based Isolation

Lecture Summary: In this lecture, we studied *object-based isolation*, which generalizes the isolated construct and relates to the classical concept of *monitors*. The fundamental idea behind object-based isolation is that an isolated construct can be extended with a set of objects that indicate the scope of isolation, by using the following rules: if two isolated constructs have an empty intersection in their object sets they can execute in parallel, otherwise they must execute in mutual exclusion. We observed that implementing this capability can be very challenging with locks because a correct implementation must enforce the correct levels of mutual exclusion without entering into deadlock or livelock states. The linked-list example showed how the object set for a delete() method can be defined as consisting of three objects — the current, previous, and next objects in the list, and that this object set is sufficient to safely enable parallelism across multiple calls to delete(). The Java code sketch to achieve this object-based isolation using the PCDP library is as follows:

```
isolated(cur, cur.prev, cur.next, () -> {
```

. . . // Body of object-based isolated construct

});

The relationship between object-based isolation and monitors is that all methods in a monitor object, M1, are executed as object-based isolated constructs with a singleton object set, {M1}. Similarly, all methods in a monitor object, M2, are executed as object-based isolated constructs with a singleton object set, {M2} which has an empty intersection with {M1}.

Optional Reading:

1. Wikipedia article on Monitors

2.3 Lecture Summary

2.3 Spanning Tree Example

Lecture Summary: In this lecture, we learned how to use object-based isolation to create a parallel algorithm to compute *spanning trees* for an undirected graph. Recall that a spanning tree specifies a subset of edges in the graph that form a tree (no cycles), and connect all vertices in the graph. A standard recursive method for creating a spanning tree is to perform a depth-first traversal of the graph (the *Compute(v)* function in our example), making the current vertex a parent of all its neighbors that don't already have a parent assigned in the tree (the *MakeParent(v, c)* function in the example).

The approach described in this lecture to parallelize the spanning tree computation executes recursive *Compute(c)* method calls in parallel for all neighbors, *c*, of the current vertex, *v*. Object-based isolation helps avoid a data race in the *MakeParent(v,c)* method, when two parallel threads might attempt to call *MakeParent(v1, c)* and *MakeParent(v2, c)* on the same vertex *c* at the same time. In this example, the role of object-based isolation is to ensure that all calls to *MakeParent(v,c)* with the same *c* value must execute the object-based isolated statement in mutual exclusion, whereas calls with different values of *c* can proceed in parallel.

Optional Reading:

1. Wikipedia article on Spanning Trees

2.4 Lecture Summary

2.4 Atomic Variables

Lecture Summary: In this lecture, we studied *Atomic Variables*, an important special case of object-based isolation which can be very efficiently implemented on modern computer systems. In the

example given in the lecture, we have multiple threads processing an array, each using object-based isolation to safely increment a shared object, *cur*, to compute an index *j* which can then be used by the thread to access a thread-specific element of the array.

However, instead of using object-based isolation, we can declare the index *cur* to be an *Atomic Integer* variable and use an atomic operation called *getAndAdd()* to atomically read the current value of *cur* and increment its value by 1. Thus, j=cur.getAndAdd(1) has the same semantics as isolated (cur) { j=cur;cur=cur+1; } but is implemented much more efficiently using hardware support on today's machines.

Another example that we studied in the lecture concerns *Atomic Reference* variables, which are reference variables that can be atomically read and modified using methods such as *compareAndSet()*. If we have an atomic reference *ref*, then the call to *ref.compareAndSet(expected, new)* will compare the value of *ref* to *expected*, and if they are the same, set the value of *ref* to *new* and return *true*. This all occurs in one atomic operation that cannot be interrupted by any other methods invoked on the *ref* object. If *ref* and *expected* have different values, *compareAndSet()* will not modify anything and will simply return *false*.

Optional Reading:

1. Tutorial on Atomic Integers in Java

2. Article in Java theory and practice series on Going atomic

3. Wikipedia article on Atomic Wrapper Classes in Java

2.5 Lecture Summary

2.5 Read-Write Isolation

Lecture Summary: In this lecture we discussed *Read-Write Isolation*, which is a refinement of object-based isolation, and is a higher-level abstraction of the *read-write locks* studied earlier as part of Unstructured Locks. The main idea behind read-write isolation is to separate read accesses to shared objects from write accesses. This approach enables two threads that only read shared objects to freely execute in parallel since they are not modifying any shared objects. The need for mutual exclusion only arises when one or more threads attempt to enter an isolated section with write access to a shared object.

This approach exposes more concurrency than object-based isolation since it allows read accesses to be executed in parallel. In the doubly-linked list example from our lecture, when deleting an object *cur* from the list by calling *delete(cur)*, we can replace object-based isolation on *cur* with read-only isolation, since deleting an object does not modify the object being deleted; only the previous and next objects in the list need to be modified.

Optional Reading:

1. Wikipedia article on Readers-writer lock

Mini Project 2: Global and Object-Based Isolation

---

miniproject_2.zip

Project Goals and Outcomes

One of the major challenges of writing concurrent applications is the correct protection of data shared by multiple, concurrently executing streams of execution. "Correct" protection of shared data generally has several properties:

1. A deadlock cannot occur (i.e. progress is guaranteed).
2. A data race cannot occur.
3. That protection does not cause excessive contention, and hence does not significantly hurt performance.

One of the concurrent programming concepts you learned about this week that helps to prevent all three of the above issues is object-based isolation. In object-based isolation, the objects being accessed are explicitly passed to the synchronization construct being used. The synchronization construct can then use this information on the specific objects being accessed by the critical section to optimize synchronization and improve safety. This is in contrast to global isolation, where a single global lock or construct ensures safety but might experience heavy contention. Two examples of object-based isolation are the Java synchronized statement and the PCDP object-based isolated API.

In this mini-project, you will gain experience writing code that makes use of object-based isolation using PCDP. To motivate this topic, we will implement a concurrent banking system that can accept transactions from many threads of execution at once. We will use object-based isolation to protect bank accounts from concurrent accesses to ensure that all bank balances are correct, and enable more transactions per second than would be possible with global isolation.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_2.zip at the top of this description, you should see the project source code file at

miniproject_2/src/main/java/edu/coursera/concurrent/BankTransactionsUsingObjectIsolation.java

and the project tests in

miniproject_2/src/test/java/edu/coursera/concurrent/BankTransactionsTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of BankTransactionsUsingObjectIsolation.java. You should not change the signatures of any public or protected methods inside of BankTransactionsUsingObjectIsolation.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of BankTransactionsTest.java in the final grading process, so do not change that file or any other file except BankTransactionsUsingObjectIsolation.java.

Your main goals for this assignment are as follows:

1. Inside BankTransactionsUsingObjectIsolation.java, implement the issueTransfer method using object-based isolation to protect against concurrent accesses to the source and destination bank accounts.

There are helpful TODOs in BankTransactionsUsingObjectIsolation.java to help guide your implementation.

Project Evaluation

Your assignment submission should consist of only the BankTransactionsUsingObjectIsolation.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera’s auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 50% - performance of object-based isolation on two cores
- 50% - performance of object-based isolation on four cores