CSCI 4430 Programming Languages

Homework 9: Java Concurrency

Due: Monday May 9th 2016 @ 3:59pm

This assignment is to be done in pairs. If completed individually, there will be a penalty of 5%.

Submission Instructions

Submit the following:

hw09/README // **Important:** Include your full names and RCS Ids!

hw09/MultithreadedServer.java // Turn sequential starter code into parallel one

hw09/Account.java // You are not permitted to modify this class

hw09/constants.java // You are not permitted to modify this class

hw09/test/MutlithreadedServerTests.java // Add JUnit tests to starter ones

hw09/data/datafiles // Add data files for testing, in addition to increment and rotate

You must keep the above directory structure (as introduced by the starter code). From the parent directory of hw09, do **zip hw09.zip –r hw09**. Submit hw09.zip to the homework server for autograding.

You must use Java 7 on this assignment!

Transaction Server

By Prof. Michael Scott

For this assignment you will construct a parallel transaction server. To keep the assignment manageable we will build a simple, stylized server that captures the concurrency aspects of the problem domain while avoiding the complexity of Internet communication, database access, logging, fault tolerance, etc.

Problem 1

Download starter code from hw09.zip. The bulk of this homework requires that you modify file MultithreadedServer.java to turn the sequential server into a parallel one. In files MultithreadedServer.java and Account.java you will find the code for the **sequential** server. Read carefully through MultithreadedServer.java as it contains TODO notes and additional explanation. The server operates on an array of 26 “accounts”, named ‘A’ through ‘Z’, which it modifies according to a sequence of *transactions*, specified one per line. These transactions have the following form:

input	--> *transaction input*
	-->
transaction	--> *command more_commands* **`\n`**
more_commands	--> **`;`** *command more_commands*
command	--> *account* **`=`** *val val_tail*
account	--> **`capital_letter`** *indirects*
indirects	--> **`*`** *indirects*
	-->
val	--> *account*
	--> `integer`
val_tail	--> **`+`** *val*
	--> **`-`** *val*

Tokens other than semicolon and newline (**`\n`**) must be surrounded by white space.

Each command sets the account specified by the left-hand side to the value specified by the right-hand side. For example, the following transaction transfers ten units from account B to account A, atomically:

A = A + 10; B = B - 10

If the letter name of an account is followed by a star, then the value in the account, modulo 26, is interpreted as a reference to another account—kind of a weird variety of pointer. The significance of indirection for this assignment is that transactions are *dynamic*: they can't tell up front what accounts they are going to need; they may have to access one or more accounts before figuring out what others will need to be accessed. Suppose, for example, that account Q contains the value 37, account L contains the value 5, and account F contains the value 9. Then the transaction

A = Q**; B = A + A*

will work as follows: Q* has the value 5 (because 37 == 11 mod 26, and L is the 11th letter of the alphabet, counting from 0), and then Q** has the value 9 (because F is the 5th letter of the alphabet, counting from 0). So we assign 9 into A. Then since J is the 9th letter of the alphabet, counting from 0, we take the value in J, add 9 to it, and put the result in B.

Note that the commands within a transaction have to be executed in order, but the transactions in the input do not. This is perfectly normal, and happens all the time in real life. If there is only one concert ticket left and I buy it, then you can't, and you may choose to do something else on Saturday night. Moreover if we both try to buy that last ticket at about the same time, which one of us "wins" may depend on such artificial factors as Internet delay or temporary conflicts with other transactions.

To get you started, here are two very simple inputs, both with only one command per transaction. The first, hw09/data/increment, has no conflicts among transactions. A parallel server should execute it very quickly. The second, hw09/data/rotate, has a great many conflicts; it presents more of a challenge. You can write a solution that still executes it faster than the sequential version does. Note that the code we are giving you prints a message for each commit or abort operation, so you can figure out whether your program is correct.

The Account class, *which you are not permitted to modify*, provides several public methods:

Account (constructor)

takes an initial value as argument. You'll discover that the main program initializes the *n*th account to $26 - n$.

peek

returns the value in an account.

verify

checks to see whether an account contains an expected value. You'll use this to make sure that no other transaction has updated an account since you peeked at it.

update

sets an account to a new value.

open

secures the right to verify (read) or update (write) an account. Takes a boolean argument indicating which mode is desired.

close

relinquishes rights to verify and update the account.

print

prints the value of the account to standard output, in a wide (11-column) field.

print

prints the value of the account mod 26 to standard output, in a two-column zero-padded field. These last two methods are used by the dumpAccounts method of the main (Server) class to print debugging output.

You will find that each `open`, `close`, `peek`, `verify`, and `update` operation incurs a delay of 100ms. This is intended to simulate communication with a remote database. Given these delays, server performance will depend critically on the ability to work on more than one transaction at a time, so the delays can overlap. Use the Java 5 `Executor` mechanism to manage this concurrency. Each transaction should be represented by a task. With the `newFixedThreadPool` factory you can specify the number of threads in the worker pool, to balance the performance advantage of concurrency against the performance overhead of extra context switches. Alternatively, you can simply use the `newCachedThreadPool` factory, which will create as many threads as are able to proceed concurrently.

Where the sequential version of the transaction server performs transactions sequentially, you should arrange for your concurrent version to be *disjoint access parallel*. That is, any two transactions that access no common accounts should be able to proceed without aborting or otherwise interfering with each other.

Of course if concurrent transactions access the *same* accounts, their updates may conflict with one another. Moreover because of indirection it's impossible to tell in advance whether two transactions will conflict. I recommend (and the infrastructure you've been given supports) an optimistic strategy in which transactions proceed concurrently unless and until a conflict is discovered, at which point one of them must abort.

You are required to ensure that transactions are *serializable*; that is, that their overall effect is equivalent to that of an execution in which the transactions occur sequentially, one by one, in some order. To achieve this effect without deadlock you can implement two-phase locking, in which all a transaction's `open` operations occur before any of its `close`s. The key to deadlock freedom in this scheme is to open all accounts in some canonical order (e.g. A through Z), to avoid any circular dependences. This in turn requires that a transaction figure out what it wants to do (including all indirections) before actually opening anything; hence the rule that `peek` must be called on an *unopened* account.

To support the implementation outlined above (i.e. to detect conflicts among transactions, and to encourage you to cache peeked-at and to-be-written values), the `Account` class enforces certain access rules:

- An account must be "opened for reading" before it can be verified. It must be "opened for writing" before it can be updated.
- An account can be peeked at when it is open in another thread, but *not* when it is open in the current thread.
- An account cannot be opened for reading if it is already open for reading or writing in the current thread. It cannot be opened for writing if it is already open for writing in the current

thread.

- If a thread attempts to open an account for reading that is currently open for writing in another thread, or if it attempts to open an account for writing that is currently open for reading or writing in another thread, then the open operation will throw a `TransactionAbortException`, which indicates that the transaction must abort and retry. You will need to modify the code to close any accounts that are already open when this occurs.
- An account can be closed only by a thread that has it open.

Remember: you are not permitted to modify `Account`.

Problem 2

In addition to MultithreadedServer.java, you are required to create transaction files in hw09/data/ and add JUnit tests to hw09/test/MultithreadedServerTests.java to test your implementation. We will grade for white-box coverage expecting that your tests achieve at least 80% statement (block) coverage of Account.java and MultithreadedServer.java.

Problem 3

You are required to comment your code. Use commenting style of your choice, e.g., Javadoc or one you have learned in Data Structures, Principles of Software or another class.

In addition, you are required to annotate object allocation sites when the allocated object(s) that can be accessed by more than one thread. Use two annotations, `/* shared mutable state */` and `/* shared immutable state */` as in the examples below. (These are random examples. You do not have to have these lines in your code.)

```
/* shared mutable state */ a = new Account(10);

/* shared immutable state */ List<Account> list = new
ArrayList<Account>();
```

For our purposes, an object is `shared mutable state` when two or more concurrent threads can access that object and at least one of these accesses is a modification to the object (including transitive modification). An object is `shared immutable state` when all concurrent accesses are reads. Minimize and be extra careful with shared mutable state.

If an object creation site is left un-annotated, that means the allocated object(s) is thread-local, i.e., it is never accessed by more than one thread.

Resources

If you are not familiar with Java, you should take a look at Oracle's on-line tutorial. For answers to specific questions, consult the Language Reference Manual or your favorite Java book. Full documentation on Java 7 can be found at docs.oracle.com/javase/7/docs.

Getting started with Eclipse

Perhaps the easiest way to load the starter code in Eclipse is as follows. Create a new Project (File -> New -> Java Project), say **TransactionServer**. Open a command-line window and go to your Eclipse **workspace** directory (on a Mac, this is usually /Users/yourUsername/Documents/workspace/). In **workspace**, you'll find the project you just created: workspace/TransactionServer/src. Copy hw09.zip into workspace/TransactionServer/src, then do **unzip hw09.zip**. Return to Eclipse, and in Project Explorer click on **TransactionServer**, then choose File -> Refresh. Package **hw09** will show in Project Explorer.

To load JUnit into your project, click on TransactionServer, then Project -> Properties -> Libraries -> AddLibrary -> JUnit.

(Optional) To measure code coverage in Eclipse, install the EclEmma tool: In Eclipse Marketplace, search for EclEmma.

Important note: When loading files for reading in Eclipse, by default you'll use relative path names starting at src, e.g., src/hw09/data/increment. However, in the Homework server, src-based relative path names fail. If you remove src/, the pathname will work in the Homework server. A better solution is to change the default configuration in Eclipse to allow relative path names such as hw09/data/increment, just as in the Homework server. To do so, right-click on MultithreadedServerTests.java, then Run Configurations… In Run Configurations, choose Arguments, and in Working Directory, click Other. In Other, type ${workspace_loc:TransactionServer}/src/.

Grade Breakdown

The homework is out of 60 points. We'll override the autograded points if you have submitted sequential code (we have hidden tests that verify "multithreaded-ness").

Problem 1. Functional correctness of multithreaded server: 20 points. (autograded and reviewed by us)
Problem 2. Quality of tests: 15 points. (autograded)
Problem 3. Comments and annotations: 15 points for comments and 10 points for correct annotations (TA graded)