CSE 6431 Lab: Concurrent Transactions

As we learned in the class, synchronization in a multi-threaded program is in general a hard problem, and there is no generic solution. However, in practice, people have developed several paradigms you can follow. This lab uses concurrent transactions in databases as an example to illustrate two of them.

Simplified database and transaction: In this lab, we implement a very simplified database. It contains a fixed number of rows stored in memory. Each row is a single integer value. A transaction can contain a list of read or write operations (i.e., no insert, delete, range queries, etc), and one operation can read/write a single row. We assume operations of a transaction touch distinct rows and thus they can be executed in any order. We assume a transaction never aborts. In other words, you do not need to consider atomicity and durability in this lab. You only need to implement isolation/concurrency control. You need to achieve serializability.

Paradigm 1: Strict two-phase locking. We have learned this in the class. To implement this, you will need to add a read/write lock for each row and grab corresponding locks when executing a transaction. Since all rows are known before executing a transaction, you should be able to avoid the deadlock problem. gradual locking

Paradigm 2: Pipelined execution. This is another popular paradigm used in practice. In this paragraph, we divide all data into several partitions and assign one thread to be responsible for each partition. For example, assuming our database has 100 rows, we can create 10 threads, let thread 0 be responsible for rows 0-9, let thread 1 be responsible for rows 10-19, and so on. Then when executing a transaction, we let the transaction flow from thread 0 to thread 9: If thread i is responsible for some of the operations in the transaction, thread i executes those operations. If thread i is not responsible for any of the operations, it simply does nothing. Either way, thread i passes the transaction to thread i+1. In this way, all threads are chained into a pipeline with producer-consumer queues, in which thread i is the consumer of thread i-1 and the producer of thread i+1. The advantage of this paradigm is that, since a data item can only be touched by one thread, there is no need for synchronization.

Details. You are strongly encouraged to implement this lab in Java. I provide a template code to start with in Java. However, if you really prefer C/C++, that's also OK.

In the template code, I provide definitions of the database, transactions, and operations, and a serial implementation of multiple transactions. You are free to change those when necessary. For example, you probably will need to add an ID to each transaction to distinguish them. Use "javac *.java" to compile the code. Use "java Database" to execute it.

You are required to replace the serial implementation with concurrent execution. You are required to provide two implementations corresponding to the above two paradigms. To be concrete, you will need to add locks (for paradigm 1) or producer-consumer queues (for paradigm 2), create threads (for both paradigms), and let each thread do its job. For paradigm 1, you should create one thread for each transaction. For paradigm 2, you should create 10 threads, each responsible for 10 rows.

Tests: You should write a program to enumerate all possible serial executions and verify that your execution is equivalent to one serial execution. Assuming the number of transactions is not large, this should not take long. Optional: You may also implement the serialization graph approach if you are interested, but this is not required.

Expected output: Your program should output to screen a log like "Transaction 1 reads row 100 = 3; Transaction 2 writes row 99 = 6; …" and finally output "This execution is equivalent to a serial execution of Transaction 2 -> Transaction 1-> …".

Submission: Your submission should include all .java files and a readme file about how to compile and run your code. You should pack them into a zip file and submit it on Carmen. For two paradigms, you may have two Database.java like Database1.java and Database2.java. In your submission, you can include any batches of transactions that you use for testing. The TA may add more testcases. We will test your code on coelinux.coeit.osu.edu, so make sure that your code works on coelinux.

Related readings:

Java threads: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Java read/write lock:

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html

Java producer-consumer queue (You do not need to implement this by yourself):

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html