DBMS Design Report

Concurrency and transaction atomicity are two essential parts of any database management system. In this simplified DBMS project, we use ReadWriteLock to handle concurrency errors and tempfile to guarantee transaction atomicity.

Our team members prefer file safety to performance, so we chose serializable as this DB’s isolation level and ReadWriteLock to deal with concurrency errors. ReadWriteLock has a great advantage than synchronize, that is it has two separate locks for writing and reading. It allows multiple users to read a certain file, at a time. In other words, multiple threads can read from a shared file without causing concurrency errors. This will improve reading performance. The concurrency errors first occur when reads and writes to a shared resource occur concurrently, or if multiple writes take place concurrently○1. Since we lock the file when writing, no one else can access the file that guarantee situations like dirty read and phantom read would never happen.

Furthermore, we create two file officialFile named cs542.db and tempFile named tempFile.db so that we can make sure the original file keeps untouched when DB write data to file. Every time when new data is added or old data is removed, the updated data will always be pushed to tempFile first. Then we rename the tempfile cs542.db. If machine reboots or system is down when updating data, data in tempfile may loss but data in cs542 will keep untouched since we do not write data to cs542 directly. This design guarantees any unsuccessful transaction rolls back. In addition, tempFile can also be used to measure the length of file in case of oversize. Some more detail will be included in the function explanation.

Before we can call any methods, we must initialize HW1 class first. Then we create a HashMap to store data in cs542 and create a tempFile, a ReadLock and a WriteLock.

Put function:

Lock the HW1 object by writeLock then put the new key- value pair into HashMap. Here is the rule for putting new data that if there is any key in hashtable same as the adding key, previous value will be replaced by the new value. After putting new key-value pair into hashtable, we copy the whole hashMap to tempFile and judge tempFile by length. If the tempfile meets the length requirements, we rename tempFile as cs542.db. If not, return "The file does not have enough room for data”. Finally release the lock.

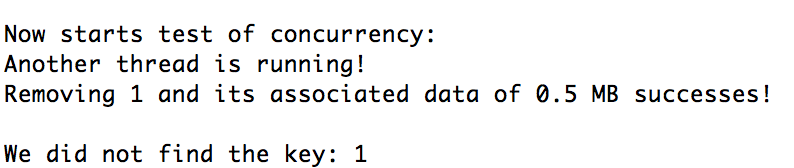
Get function:

Lock the HW1 object by readLock then retrieve the data for the given key. If we can find that key in the hashtable, then return the length of data for that key. If not, return "We did not find the key”. After that, release the lock.

Remove function:

Lock the HW1 object by writeLock then remove the data for the given key. If we can find that key in the hashtable, return “Successes”. If not, return “did not find”. After that, write new hashMap to tempFile and rename tempFile cs542 meanwhile update the length of 542.db. Then release the lock.

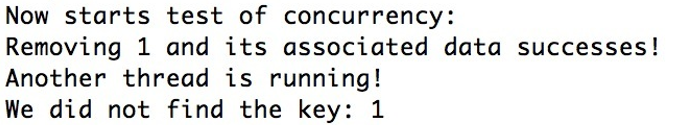
Test result:

We first put one data pair with key = 1 and length = 0.5M into file as the original officialFile cs542.db and then test concurrency. Besides the main thread, we create a new thread named t to run get method. We call remove function to remove value of which key is 1 in the main thread and t thread calls get method

Concurrency test:

We do the test: when one caller does a Remove() and another caller does a Get() with the same key a millisecond later.

In the test code, we use thread to control function remove(1) and get(1) happen with a millisecond later. And the result shows as following:



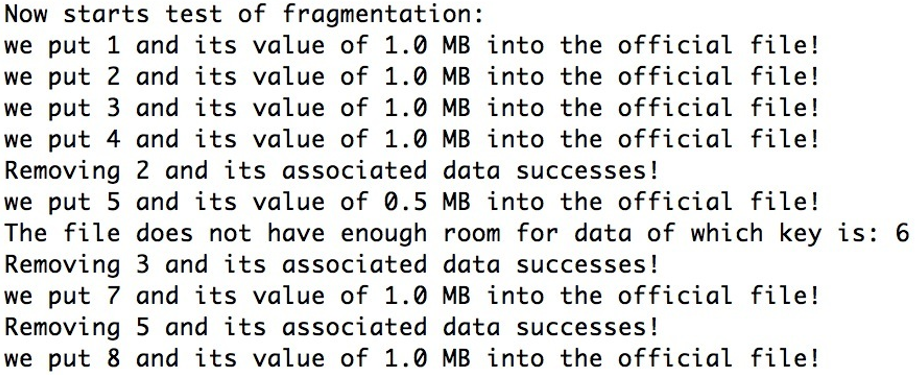
We can find that this two processor do not interrupt each other, it let the remove happen before getting value. We guarantee concurrency successfully.

Durability test:

Fragmentation test:

We do the test: Put() 4 values, byte arrays of 1 MB each, with keys A, B, C and D. Remove key B.Put() ½ MB in size for key E. Validate that a Put() 1 MB in size for key F fails. Remove C and now validate that a Put() 1 MB in size for key G succeeds. Remove E and try Put() 1 MB in size for key H. With a naive implementation, it will fail even though there is room in store.db. An extra bonus point if you can modify your code such that Put("H", …) succeeds.

The resulting is showing as following, we do every process successfully.



Reference:

1. <http://tutorials.jenkov.com/java-util-concurrent/readwritelock.html>