ASSIGNMENT 3: Christmas, Concurrent Data Structures and Databases

CSC3021 Concurrent Programming 2018-2019

Submit by 5:59 pm on Friday 14th December 2018

Question 1: Christmas Tree Shop

A Christmas tree merchant is repeatedly selling Christmas trees to customers and loggers are bringing in more trees. The merchant is constantly busy in the shop while customers and loggers repeatedly "enter" and "exit" the shop, following these rules:

- There can be any number of customers in the shop at any time.
- At most 3 loggers can be in the shop at any time.
- The merchant must meet with a customer to make a sale. Each customer buys one tree on a visit to the shop. The merchant can sell a tree only if there is at least one tree in the shop.
- A logger brings two trees into the shop per visit.
- Customers do not enter the shop if loggers are present in the shop.
- Initially, the shop stores 20 Christmas trees.

Write process definitions for the merchant, the customers, and the loggers such that the above constraints are satisfied. You may assume the following process structure:

```
Semaphore and variable declarations…
```

```
Process Merchant

  while(true){
    …
    //need 1
    //tree and
    //1 customer
    …
}
End Merchant
```

```
Process
Customer[1..C]
  while(true){
    …
    //has entered
    …
    //has left
    …
  }
End Customer
```

```
Process Logger[1..L]

  while(true){
    …
    //has entered
    …
    //has left
    …
  }
End Logger
```

You may use semaphore variables and integer variables. You do not need to consider starvation issues but your code must be free of deadlock.

This question is a paper-based programming exercise and your hand-in should contain pseudo-code that is clear to read and unambiguous.

Question 2: Lock-Free Hash Tables

In this question you will explore the pros and cons of blocking vs lock-free concurrent data structures. You will base this exploration on maps. Maps associate a key of type Key with a value of type Value. They support following operations:

```
interface Map<Key, Value> {

    // insert value for key and return true if key was not present yet
    public boolean add(Key key, Value value);

    // remove the value associated to key and return true if key was found
    public boolean remove(Key key);

    // return the value associated to key
    public Value get(Key key);

    // check for presence
    public boolean contains(Key key);

    // count the number of elements, need not be "thread-safe"
    public int debuggingCountElements() { … }

}
```

All classes implementing the interface Map are furthermore expected to provide a constructor with one integer argument that determines the initial size of the map.

Implement the following concurrent data structures that implement the Map interface:

(i) Implement a class with signature

```
class CoarseGrainHashMap<Key, Value> implements Map<Key, Value> {

        private Map<Key, Value> map;

        CoarseGrainHashMap(int capacity) { … }

        public boolean add(Key key, Value value) { … }

        public boolean remove(Key key) { … }

        public Value get(Key key) { … }

        public boolean contains(Key key) { … }

        public int debuggingCountElements() { … } // need not be thread-safe

}
```

using java.util.HashMap to store the map. This class is not synchronised; synchronisation needs to be handled in your class. You need to ensure that your class is synchronised correctly by ensuring its add, remove and get methods are executed under mutual exclusion.

(ii) Implement a class SegmentedHashMap that spreads the data across multiple segments. Each segment operates as an independent hash map. Keys are directed to a segment by means of a hash function, such that a key is mapped to the same segment on every invocation of

get, remove or add. Each segment has its own lock and uses a hash map of type java.util.HashMap to store its data.

(iii) Implement a class LockFreeHashMap<Key, Value> using the algorithm for lock-free hash sets in Section 13.3 of the book "The Art of Multiprocessor Programming" by Herlihy and Shavit. This algorithm consists of two classes: LockFreeHashMap, which is available on QOL, and a class BucketListMap, which implements the lock-free linked-list-based set with some minor extensions. The class BucketListMap implements the functionality of a linked list, but contains no synchronisation. You need to implement the scheme of the lock-free linked list that under-pins the lock-free hash map. Follow the scheme of Chapter 9 in the book, slides 10_linkedlist_lockfree.ppt.

Note that the book contains virtually all of the source code you need, but you will need to extend it from a hash set to a hash map. A copy of the relevant chapters is provided on QOL.

You will analyse the relative performance of several map implementations by varying the ratio of add, remove and get operations. To this end, you are provided with a micro-benchmarking framework that executes a mix of add, remove and get operations at a very high rate using a configurable number of threads. You can use this micro-benchmark for evaluating the performance of concurrent hash tables, but also for debugging. Details of the micro-benchmark are provided in below.

Writeup

Answer the following questions in the writeup:

Q2 (i) Identify the linearization point(s) for the add, remove and get methods in the LockFreeHashMap. Motivate your answer. Copy-paste code snippets of the relevant methods in your writeup. Format the code using a pretty printer and display line numbers on the code in order to facilitate your discussion. Note that the linearization points are not necessarily located in the file LockFreeHashMap.java.

Q2 (ii) Evaluate the performance of your lock-free hash map implementation using the supplied micro-benchmarking framework (see Appendix I). You are asked to perform three experiments, each summarized in a graph and supplemented with an explanation of your results:

(a) Analyse the impact of the number of segments in the SegmentedHashMap on throughput. You may set `<rounds>` to at least 10, select 90% get operations (corresponding to `<gets>`=18) and set `<dly>` to 1 second. Select as many threads as you have hardware threads on your computer, a hash table size of 8192 elements, and vary the number of segments from 1 to at least 4 times the number of hardware threads or 16 (whichever is higher).

(b) Vary the number of threads from 1 to about 32 to measure the number of operations per cycle. Select 90% get operations. Select a hash table size of 8192 elements.

(c) Vary the fraction of get operations between 33% and 99% while keeping the number of threads fixed at a value that maximizes

throughput according to your measurement in (b). Measure during 1 second. Select a hash table size of 8192 elements.

The graphs should show the average operations per second. The graph for (a) should contain one curve while the graphs for (b) and (c) should contain three curves, one for each of your solutions. In each of cases (a)-(c), dscuss the results that you have obtained and explain the main trends that can be observed. In particular, focus on (i) particular conditions (e.g., particular values for the number of threads or the number of contains operations) under which one algorithm outperforms another, (ii) for each algorithm an explanation why it provides higher or lower throughput than others.

Q2 (iii) [BONUS] The LockFreeHashMap uses a fixed-size bucket list. As a result, linked lists will become quite long as the number of elements in the hash map increases, which causes increased search times. Resizing the bucket list, however, requires care as threads may concurrently attempt to modify elements of the bucket list. Migrating the contents of the bucket list from the old array to a new and longer array is clearly not an atomic operation.
Implement your strategy to make the bucket lists resizable while retaining the lock-free property of the data structure and retaining correctness. Briefly describe your solution in the write-up and identify where your code can be found in your git repo.

Information: Hash Map Micro-Benchmark

The micro-benchmarking framework is implemented in Driver.java and may be called with following arguments:

```
$ java Driver <threads> <init_size> <segments> <set_size> \
              <gets> <dly> <rounds> <type>
```

The parameters have the following meaning:

- `<threads>` specifies the number of threads concurrently accessing the hash map
- `<init_size>` is the initial size of the hash map
- `<segments>` records the number of segments used by the `SegmentedHashMap`
- `<set_size>` is the number of elements in the hash table. If the implementation is free of races, then the size of the hash map will remain at that size throughout the computation
- `<gets>` is the number of contain operations performed per pair of add and remove operations. As such, the contains ratio is given by `<gets>` / (`<gets>`+2). The driver repeatedly applies a sequence of one successful add, `<gets>` successful get operations and one successful remove operation to the hash table
- This sequence is repeated during `<dly>` **milliseconds** by each thread
- `<rounds>` specifies how many times this experiment is repeated in order to average out performance variations

- `<type>` is a string specifying the type of the hash map and may be one of `coarsegrain`, `segmented` or `lockfree`.

The driver reports the average number of operations per second as well as the standard deviation. Experiment with the settings such that on your system the average and standard deviation become stable (do not vary much between subsequent executions) and such that the standard deviation is small compared to the average. These conditions ensure that your measurements are reliable, which will help you to explain the results.

Question 3: Locks Galore

In this question, you will implement two-phase locking in a small, mock-up database application. Databases provide the ACID properties for transactions:

- Atomicity: a transaction either executes as a whole or not at all. Our application will simply fail if a transaction cannot complete.
- Consistency: a transaction is only allowed to bring the database from one valid state to another valid state, maintaining database invariants.
- Isolation: a transaction observes only the committed state that existed at the start of the transaction, such that transactions leave the database in the same state as if they executed sequentially.
- Durability: A transaction is complete only when the data has been written to persistent storage. Our mock-up implementation does not provide durability.

Our database implements a relational data model, i.e., the data model consists of tables where rows represent data records and columns represent different properties of these records. Internally, such a table is typically stored using two components: a sequence of records that are persisted to disk and an in-memory index that allows fast access to the records. Two types of indices are considered: A primary index corresponds to a primary key of the relation, which implies that each record in the table has a distinct value in the column that corresponds to the primary key. A secondary key is a secondary index where the values need not be unique. In our application, records are not persisted to disk but simply live on the Java heap, and only a single primary key is supported for each table.

The application models the operation of a simple bank. It consists of one table that holds accounts. The table has two properties: a unique identifier (AID) for the account and the amount of currency held in the account (AMOUNT). The table has a primary index on AID. The application may issue following SQL[1] queries to the database:

To transfer an amount between two accounts, the TRANSFER transaction retrieves current amounts held in two accounts, and writes back updated amounts to reflect the transfer (the % signs indicate values inserted by the application):

---

[1] If a reference is required on SQL, consult the MySQL documentation at https://dev.mysql.com/doc/refman/8.0/en/sql-syntax.html.

```
START TRANSACTION;
SELECT * FROM account WHERE aid=%0 OR aid=%1 FOR UPDATE;
SELECT aid,amount FROM account WHERE aid=%0 OR aid=%1;
UPDATE account SET amount=%2 WHERE aid=%0;
UPDATE account SET amount=%3 WHERE aid=%1;
COMMIT;
```

The CREATE transaction creates a new account:

```
START TRANSACTION;
INSERT INTO account VALUES (%0,%1);
COMMIT;
```

To detect embezzlement of funds, the SUM transaction checks that the total amount held in all accounts remains the same:

```
START TRANSACTION;
SELECT * FROM account FOR UPDATE;
SELECT SUM(amount) FROM account;
COMMIT;
```

Note that the CREATE transaction requires a lock on the table because the underlying data structures do not support concurrent reads and writes. The SELECT FOR UPDATE statements must acquire the table-level lock in addition to the row-level locks.

The client application presents multiple transactions concurrently to the database. As such, proper concurrency control must be in place. Concurrency errors can happen at two levels:

- Race conditions may occur when accessing the index associated to a table, which may result in an inconsistent index, or null pointer exceptions when accessing the index. To avoid this, databases associate locks to tables.
- Complex operations may need to be performed atomically, e.g., transferring funds from one account to another consists of multiple read and write operations that need to be executed in a way that respect the ACI properties. To this end databases associate locks to records (cf SELECT FOR UPDATE statement).

In the source code provided, the class QueryEngine implements the query execution engine. It contains methods that execute each type of SQL statement. You will need to implement appropriate functionality for the following methods:

- execute(SQLStartTransactionStatement stmt)
  This statement has little effect, it merely prepares to collect a set of locks
- execute(SQLCommitStatement stmt)
  This statement needs to release all locks that have been acquired by any statement in the transaction. To help you, a class LockSet is available to track locks that need to be released.
- execute(SQLInsertStatement stmt)
  Acquires a table-level lock. No records are locked.

- execute(SQLSelectForUpdateStatement stmt)
  Acquire a lock on all records that meet the conditions of the WHERE clause as well as a lock on the table.

Task You will extend the database to implement two-phase locking. The two-phase locking protocol operates as follows:

1. Acquire all the locks that are needed by the transaction, as indicated by SELECT FOR UPDATE and INSERT statements.
2. Execute the reads and writes of the transaction
3. Release all the locks that have been acquired in the current transaction. In our implementation, this is performed by the COMMIT statement.

You will need to associate locks to tables (class Table), and to records (class Record). SELECT queries use a visitor pattern over the index. You will need to complete the RecordVisitorLock class to collect record-level locks. Hint: readers/writers locks have enormous importance for databases.

In general, databases have elaborate deadlock detection mechanisms and will abort some transactions when deadlock occurs to ensure progress. You will need to resolve deadlock also in the mock-up application. In this application, the easiest approach is to prevent deadlock in your two-phase locking implementation by determining a total order among all locks and acquiring the requested locks in said order.

Answer these questions in your writeup:

1. What are the read/write accesses made by the SELECT, UPDATE and INSERT queries to the index and to records?
2. Describe the race conditions and deadlock potential when executing a SUM transaction concurrently with a TRANSFER transaction.
3. Describe the race conditions and deadlock potential when executing a TRANSFER transaction concurrently with a CREATE transaction.
4. Describe the race conditions and deadlock potential when executing a SUM transaction concurrently with a CREATE transaction.
5. Are there race conditions or deadlock that may occur that have not been covered yet above?
6. Given that each table and each record has a lock associated to it, what is an appropriate total order among locks that will allow deadlock-avoidance? It is sufficient if your solution is correct only for the set of queries used in this application.

Your solution to the database problem will make up the second part of the concurrent programming competition. You can consider following improvements on your solution:

- The SELECT SUM(amount) query is the most time-consuming. Is it worthwhile to execute the query in parallel using multiple threads?
- The SELECT FOR UPDATE query in the SUM transaction acquires all record locks in the account table. Lock escalation is a technique that minimises locking overhead: if too many record-level locks are

required, then the database will fall back to acquiring a table-level lock instead. Carefully consider the concurrency that is required among different types of queries. Is the concurrency required for the escalated lock identical to the table lock that is already present?

- Is it worthwhile to utilise your lock-free concurrent hash map of question 2 in the database? What locks could it avoid?
- Snapshot isolation is a technique to allow concurrency between read-only transactions (especially long-running ones) and transactions that update the database. Read-only transactions do not acquire any locks while transactions that perform updates acquire locks as before. For each transaction, a snapshot of the database contents are determined that include the updates made by all transactions that have committed, but contain no updates made by transactions that are in progress or that have not started yet. The database stores multiple versions of each record, corresponding to updates made by different transactions. The key steps required are: (i) assign each transaction a unique sequence number; (ii) when a transaction starts, determine the set of committed transactions; (iii) record multiple versions of each record, e.g., by maintaining a linked list of records in the index instead of just one record; and (iv) cleaning up stale versions, i.e., versions that will never be part of the snapshot for any new transaction.

  Further information on snapshot isolation can be found in the document db_isolation.pdf on QOL.

Describe your approach to enhancing concurrency in the write-up.

You are provided with a functionally correct database implementation and a driver program (Bank.java) that issues SQL queries to the database. The provided source code does not support concurrent database accesses.

The driver program accepts a number of command line flags. These are:

- --nthreads N: N concurrent threads may access the database. Set N to 1 to test for baseline correctness of a sequential implementation.
- --fsum F: Each thread executes a query to sum up all accounts once every F queries. Set F to 0 to disable these queries.
- --fcreate F: Each thread executes a query to create a new account once every F queries. Set F to 0 to disable these queries.
- --naccounts N: The initial number of accounts held by the bank.
- --msecs M: The driver runs for periods of M milliseconds. It is best to run for 5 seconds per round.
- --sum_needs_lock (true|false): true when SUM transactions must perform a SELECT FOR UPDATE query, false if this should be omitted (as in the case of snapshot isolation).

Your implementation will determine the degree of concurrency in the database, but also the correctness. If race conditions are detected or exceptions are thrown, the program will terminate with an error message.

You should not modify the driver program (Bank.java), nor the classes that represent SQL statements (SQL*.java).

Correctness checking and competition

Question 3 is part of the concurrent programming competition. The goal of this question is to implement locking in a correct way such that concurrency is maximised as indicated by the average operations per second. A £100 prize will be awarded for the student that performs best across both the concurrent PageRank and the database locking problems. Upload your code to the DOMjudge server (hvan.eeecs.qub.ac.uk/domjudge) using the contest '1819_db'.

Marking Question 3

You will receive 6 marks on your write-up and implementation choices and 8 marks on the performance of your code. The performance of a reference implementation of two-phase locking will count for 40% of the marks on this component while achieving the performance of a reference implementation of snapshot isolation counts as 100%.

You must upload your code to the server to support efficiency of marking. You are allowed to submit new solutions as frequently as you want; however only the final submission will be taken into account for marking purposes.

Submission

Your submission will involve providing access to a **git repository** which contains regular commits of your work, submission of at least one version of your code for question 3 to the **domjudge** server and a write-up of your experimental measurements and conclusion submitted to **TurnItIn**.

The write-up will be a PDF document submitted through TurnItIn. It will be at most 4 sides of an A4 page. The minimum font size should be 11 point and margins must be at least 2cm in all directions. Only a standard 'Arial' or other 'Sans Serif' font should be used. Excess pages may be ignored during marking.

Guidance: Marking scheme:

Q1: 6 marks, Q2: 9 marks + 5 bonus marks; Q3: 6 marks write-up + 8 marks competition. Total 29. Peer review: 1 mark.

This assignment contributes to 30% of the marks for CSC3021. Excess bonus marks (marks above 29, not including peer review) are added up to a previous assessed component in this module.