CSE 332 Data Abstractions, Winter 2014
Homework 8

Due**: Wednesday, March 12, 2014** at the BEGINNING of lecture. Your work should be readable as well as correct. Please write your **section** at the top of your homework.

Problem 1: Another Wrong Bank Account

Note: The purpose of this problem is to show you something you should **not** do because it does **not** work. Consider this pseudocode for a bank account supporting concurrent access; assume that Lock is a valid locking class, although it is not in Java:

```
class BankAccount {
      private int balance = 0;
      private Lock lk = new Lock();
      int getBalance() {
            lk.acquire();
            int ans = balance;
            lk.release();
            return ans;
      }
      void setBalance(int x) {
            lk.acquire();
            balance = x;
            lk.release();
      }
      void withdraw(int amount) {
            lk.acquire();
            int b = getBalance();
            if(amount > b) {
                  lk.release();
                  throw new WithdrawTooLargeException();
            }
            setBalance(b - amount);
            lk.release();
      }
}
```

The code above is wrong if locks are not re-entrant. Consider the *absolutely horrible* idea of "fixing" this problem by rewriting the withdraw method to be:

```
void withdraw(int amount) {
      lk.acquire();
      lk.release();
      int b = getBalance();
      lk.acquire();
      if(amount > b) {
            lk.release();
            throw new WithdrawTooLargeException();
      }
      lk.release();
      setBalance(b - amount);
      lk.acquire();
      lk.release();
}
```

(a) Explain why this "fixed" code doesn't "block forever" (unlike the original code).

(b) Show this new approach is incorrect by giving an interleaving of two threads, both calling the new withdraw, in which a withdrawal is forgotten.

Problem 2. Simple Concurrency with B-Trees

Note: Real databases and file systems use very fancy fine-grained synchronization for B-Trees such as "hand-over-hand locking" (which we did not discuss), but this problem considers some relatively simpler approaches.

Suppose we have a B Tree supporting operations insert and lookup (deletion is NOT a supported operation on this particular B Tree). A simple way to synchronize threads accessing the tree would be to have one lock for the entire tree that both operations acquire/release.

(a) Suppose instead we have one lock per node in the tree. Each operation acquires the locks along the path to the leaf it needs and then at the end releases all these locks. Explain why this allegedly more fine-grained approach provides absolutely no benefit.

(b) Now suppose we have one *readers/writer lock* per node and lookup acquires a read lock for each node it encounters whereas insert acquires a write lock for each node it encounters. Assume that the same policy from part (a) is followed in that a thread will only release all of its locks when it is done with its operation. How does this provide more concurrent access than the approach in part (a)? Is it any better than having one readers/writer lock for the whole tree (explain)?

(c) Now suppose we modify the approach in part (b) so that insert acquires a write lock only for the leaf node (and read locks for other nodes it encounters). How would this approach increase concurrent access? When would this be incorrect? Explain how to fix this approach without changing the asymptotic complexity of insert by detecting when it is incorrect and in (only) those cases, starting the insert over using the approach in part (b) for that insert. Why would reverting to the approach in part (b) be fairly rare?

Problem 3: Dijkstra's Algorithm

a) Weiss, problem 9.5(a) (the problem is the same in the 2nd and 3rd editions of the textbook). Use Dijkstra's algorithm and show the results of the algorithm in the form used in lecture — a table showing for each vertex its best-known distance from the starting vertex and its predecessor vertex on the path. Also show the **order** in which the vertices are added to the "cloud" of known vertices as the algorithm progresses.

   Although the final table is all that is required, for potential partial credit I would recommend showing how your table changes over time by leaving lots of space and crossing things out as they change. Also be sure that you show the value of the path variable.

b) If there is more than one minimum cost path from v to w, will Dijkstra's algorithm always find the path with the fewest edges? If not, explain in a few sentences how to modify Dijkstra's algorithm so that if there is more than one minimum path from v to w, a path with the fewest edges is chosen. Assume no negative weight edges or negative weight cycles.

c) Give an example where Dijkstra's algorithm gives the wrong answer in the presence of a negative-cost edge but no negative-cost cycles. Explain briefly why Dijkstra's algorithm fails on your example. The example need not be complex; it is possible to demonstrate the point using as few as 3 vertices.

d) Suppose you are given a graph that has negative-cost edges but no negative-cost cycles. Consider the following strategy to find shortest paths in this graph: uniformly add a constant k to the cost

of every edge, so that all costs become non-negative, then run Dijkstra's algorithm and return that result with the edge costs reverted back to their original values (i.e., with k subtracted).

- Give an example where this technique fails (Dijkstra's would not find what is actually the shortest path) and explain why it fails.

- Also, give a general explanation as to **why** this technique does not work. Think about your example and why the original least cost path is no longer the least cost path after adding k.