CSE 332 Data Abstractions, Spring 2016, Homework 7

Due**: Wednesday, May 25, 2016** at the BEGINNING of lecture.

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: Graph Representations

Suppose a **directed** graph has a million nodes, most nodes have only a few edges, but a few nodes have hundreds of thousands of edges:

a) In what way(s) would an adjacency-matrix representation of this graph lead to inefficiencies?

b) In what way(s) would an adjacency-list representation of this graph lead to inefficiencies?

c) Design a representation for this sort of graph that avoids all the inefficiencies in your answers to parts (a) and (b).

Problem 4: Topological Sort

Weiss, problem 9.1 (the problem is the same in the 2nd and 3rd editions of the textbook): "Find a topological ordering for the graph in figure 9.79 (2nd ed.)/9.81 (3rd ed.)." **For each step**, show the in-degree array and the queue in the table format shown below:

In-degree Array:

	s	A	B	C	D	E	F	G	H	I	t	Queue (Front ->End)
Step 1												
Step 2												
Etc.												