CSE 332 Data Abstractions, Winter 2011

Homework 8

Due Friday, March 11, 2011 at the **beginning** of lecture. Please be sure your work is readable (either written clearly or typed). This homework has **three** problems. ***Please write your section at the top of your homework.***

Problem 1: Concurrent Queue with Two Stacks

Consider this Java implementation of a queue with two stacks. We do not show the entire stack implementation, but assume it is correct. Notice the stack has synchronized methods but the queue does not. The queue is incorrect in a concurrent setting.

```
class Stack<E> {
      ...
      synchronized boolean isEmpty() { ... }
      synchronized E pop() { ... }
      synchronized void push(E x) { ... }
}

class Queue<E> {
      Stack<E> in = new Stack<E>();
      Stack<E> out = new Stack<E>();
      void enqueue(E x){ in.push(x); }
      E dequeue() {
            if(out.isEmpty()) {
                  while(!in.isEmpty()) out.push(in.pop());
            }
            return out.pop();
      }
}
```

(a) Show the queue is incorrect by showing an interleaving that meets the following criteria:
i. Only one thread ever performs enqueue operations and that thread enqueues numbers in increasing order (1, 2, 3, ...).
ii. There is a thread that performs two dequeue operations such that its first dequeue returns a number larger than its second dequeue, which should never happen.
iii. Every dequeue succeeds (the queue is never empty). Your solution can use 1 or more additional threads that perform dequeue operations.

(b) A simple fix would make enqueue and dequeue synchronized methods. Explain why this would never allow an enqueue and dequeue to happen at the same time.

(c) To try to support allowing an enqueue and a dequeue to happen at the same time when 'out' is not empty, we could try either of the approaches below for dequeue. For each, show an interleaving with one or more other operations to demonstrate the approach is broken. Make sure your interleaving violates the FIFO order of a queue.

```
E dequeue() {
      synchronized(out) {
            if(out.isEmpty()) {
                  while(!in.isEmpty()) out.push(in.pop());
            }
            return out.pop();
      }
}

E dequeue() {
      synchronized(in) {
            if(out.isEmpty()) {
                  while(!in.isEmpty()) out.push(in.pop());
            }
      }
      return out.pop();
}
```

(d) Provide pseudo-code for a solution that correctly supports allowing an enqueue and a dequeue to happen at the same time when 'out' is not empty; only provide pseudo-code for the method(s) that change. Your solution should involve multiple locks.

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. 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. 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. Minimum Spanning Trees

(a) Weiss, problem 9.15(a). For Prim's algorithm, start with vertex A, show the resulting table (see Table 9.55 as an example), and indicate the order in which vertices are added. For Kruskal's algorithm, produce a table, similar to Table 9.56. Ties may be broken arbitrarily.
(b) Weiss, problem 9.15(b).