CSE332 Data Abstractions, Summer 2010

Homework 7

Due Friday, August 13th, 2010 at the beginning of class.

Problem 1: Parallel Prefix and Filter

In this problem, the input is an array of strings and the output is an array of integers. The output has the length of each string in the input, but empty strings are filtered out. For example:

["", "", "cse", "rox", "", "homework", "", "7", ""]

produces output:

[3, 3, 8, 1]

A parallel algorithm can solve this problem in O(log n) span and O(n) work by 1) doing a parallel map to produce a bit vector, 2) doing a parallel prefix over the bit vector, and 3) doing a parallel map to produce the output.

Show the intermediate steps for the algorithm described above on the example above. For each step, show the tree of recursive task objects that would be created (where a node’s child is for two problems of half the size) and the fields each node needs. Do not use a sequential cut-off. Show three separate trees (for the three steps). Explain briefly what each field represents.

Note that because the input length is not a power of two, the tree will not have all its leaves at exactly the same height.

Problem 2: 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 how this approach prevents blocking forever unlike the original code.

(b) Show this approach is incorrect by giving an interleaving of two threads in which a withdrawal is forgotten.

Problem 3: 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.