EE 382C/EE 361C Multicore Computing Fall 2018 Assignment 3

Grace Zhuang (gpz68) and David Chun (dc37875)

1. (10 points) For each of the histories below, state whether it is (a) sequentially consistent, (b) linearizable. Justify your answer. All variables are initially zero.

Concurrent History H1

P1 [read(x) returns 1] P2 [write(x,1)] [read(x) returns 2] P3 [write(x,2)]

It is sequentially consistent and linearizable. A possible history would be P2[write(x, 1)], P1[read(x) returns 1], P3[write(x, 2)], P2[read(x) returns 2]. Since it is linearizable, it is also sequentially consistent.

Concurrent History H2

P1 [read(x) returns 1] P2 [write(x,1)] [read(x) returns 1] P3 [write(x,2)]

It is sequentially consistent and linearizable. A possible history would be P3[write(x, 2)], P2[write(x, 1)], P1[read(x) returns 1], P2[read(x) returns 1]. Since it is linearizable, it is also sequentially consistent.

Concurrent History H3

P1 [read(x) returns 1] P2 [write(x,1)] [read(x) returns 1] P3 [write(x,2)]

It is sequentially consistent and not linearizable. A possible sequential history would be P3[write(x, 2)], P2[write(x, 1)], P1[read(x) returns 1], P2[read(x) returns 1]. However, it is not linearizable, because in real-time, we see that P3[write(x, 2)] happens before P2[read(x) returns 1] (invocation of P2 happens after response of P3), and there is no other action to write the value 1 to x, so P2 cannot possibly have read(x) return 1.

2. (5 points) Consider the following concurrent program.

Initially a, b and c are 0. P1: a:=1 ; print(b) ; print(c); P2: b:=1 ; print(a) ; print(c); P3: c:=1 ; print(a) ; print(b);

Which of the outputs are sequentially consistent. Justify your answer.

(a) P1 outputs 11, P2 outputs 01 and P3 outputs 11.

Yes it is sequentially consistent. A valid history would be P2(b:=1), P2[print(a)], P3(c:=1), P2[print(c)], P1(a:=1), P3[print(a)], P3[print(b)], P1[print(b)], P1[print(c)].

(b) P1 outputs 00, P2 outputs 11 and P3 outputs 01.

No it is not sequentially consistent. There is a cycle from P1 to P3. P1(a:=1), P1[print(b)] and P1[print(c)] must occur before P3(c:=1) and P3[print(a)]. However, P3[print(a)] must come before P1(a:=1), creating a breakpoint.

3. (5 points) Suppose that an array does not have all elements that are all distinct. Show how you can use any algorithm that assumes distinct elements for computing the maximum to solve the problem when elements are not distinct.

All we would need to do here is implement some sort of "tie-breaking" system by index. Initially, for each element in the array, we store the element AND the index in the array where it resides. We modify the original algorithm that compares values by adding an extra check that if 2 values are the same, then the one with the smaller index is "bigger". By storing a pair of values (element value and element index) in each location in the array, we make non-distinct elements distinct, since no 2 elements can have the same index. Therefore, the algorithm that worked for distinct elements will now work for non-distinct elements.

4. (20 points) Give a parallel algorithm on a CREW PRAM with time complexity O(log n) and work complexity O(n) to compute the inclusive parallel prefix sum of an array of size n by combining two algorithms: sequential prefix sum that takes O(n) time and O(n) work and a non-optimal parallel prefix algorithm that takes O(logn) time and O(nlogn) work.

Essentially what we will do is divide the array into logn chunks, each of n/logn size, and run the sequential algorithm on the individual chunks, and then the parallel algorithm on all the chunks to combine them for the total parallel prefix sum. The details are as follows:

1) Compute the parallel prefix of each individual chunk using the sequential algorithm. Chunks are of size logn, so this will take O(logn) time and O(logn) work on each chunk.
2) Now, you have the parallel prefix sum calculated for each of the n/logn chunks. We will run the parallel algorithm on this whole thing, treating each "chunk" as one entity so that whenever an entity is updated, all that means is that all the values in that "chunk" have a value added to them in parallel (so that it takes O(1) time to update the entire "chunk"). Parallel algorithm in this situation, with n/logn entities/"chunks", will have time complexity O(log(n/logn)) = O(logn - loglogn) = O(logn). Work complexity will be O((n/logn)log(n/logn)) = O((n/logn)(logn-loglogn)) = O(n).
3) All together, the time complexity is O(logn + logn) = O(logn) and the work complexity is O(logn + n) = O(n), hereby fulfilling the time and work complexity requirements by merging the 2 algorithms.

5. (30 points, programming) The goal is to develop a concurrent implementation of a sorted linked list of integers. that uses n threads, where n = 1,2,4,8. The linked list provides the following methods: boolean add(int x), boolean remove(int x), boolean contains(int x). The method add returns true if x was not in the list and was added to the list; otherwise, it returns false (and does not add it multiple times). The method remove returns true if x was in the list and was removed from the list; otherwise, it returns false. The method contains returns true if x is in the list.

Implement the following schemes: (a, 5 points) Coarse-grained Locking (b, 10 points) Fine-grained Locking (c, 15 points) Lock-Free Synchronization.

6. (30 points, programming) (a, 20 points) Implement Lock-based and Lock-Free unbounded queues of Integers. For the lock based implementation, use different locks for enq and deq operations. For the variable count use AtomicInteger. For the lock-free implementation, use Michael and Scott's algorithm as explained in the class. The deq operation should return null if the queue is empty.

(b, 10 points) Implement Lock-Free stack of Integer. You should provide push(Integer x) and Integer pop(). The pop operation should throw an exception called EmptyStack if the stack is empty. For both the data structures use a list based implementation (rather than an array based implementation).