# CS149 Writing Assigenment 4

Name: Yi Hu(yeehu), Sean Hsu(khhsu)

## Problem 1

1. Assume processor 1 has the lock. Then the variable lock has a value of 1. Another processor 2 tries to acquire the lock and a BusRdX happens. Now processor 2 has the cache line of the variable lock and is in the M state but processor 1 has the lock. Processor 2 is in the while loop to wait for the lock.
2. Since READ AFTER WRITE is relaxed, it is possible that when thread 2 acquires the lock and loads x from the memory before the thread 1 writes back to the memory. Thread 2 would observe x=0 in this case.
3. Yes. RDDs are partitioned and these transformations have narrow dependencies.

## Problem 2

1. The sequence of locking and unlocking operation by the 2 threads can run into a deadlock problem. For example, if 2 threads run concurrently and thread 1 obtains lock(l1) and lock(l2) and thread 2 obtains lock(l3), then there is no way that thread 1 to obtain lock(l3) and no way for thread 2 to obtain lock(l2). Since both threads cannot proceed to critical section and release the locks, this turns into a deadlock situation.

   In order to avoid this problem, we can order the sequence like below:

   ```
           T0                                T1
   ==================                ==================

   lock(l1);                           lock(l1);
   lock(l2);                           lock(l2);
   lock(l3);                           lock(l3);

   // critical section                 // critical section

   unlock(l3);                         unlock(l3);
   unlock(l2);                         unlock(l2);
   unlock(l1);                         unlock(l1);
   ```

   In this sequence, whichever lock obtains lock(l1) first will exclusively obtain lock(l2) and lock(l3) as well. Since lock(l1) is released last, it ensures that no other threads can obtain the locks and proceed to the critical section while other threads still owns any lock. This preserves the mutual exclusion and ensures the correctness problem.

2. The implementation is provided below. Deadlocking occurs when hash function hashes to the same index, and the threads try to acquire the lock for the same lock before unlocking, resulting in a deadlock. Thus checking this condition is important before performing insertion into the hash table.

   ```
   struct Node {
     string value;
     Node* next;
     Lock* lock;
   };

   struct HashTable {
      Node* bins[NUM_BINS];   // each bin is a singly-linked list

   };

   int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
   ```

```
bool  findInList(Node* n, string str);    // return true is str is in the list
void  insertInList(Node* n, string str);  // insert str into the list

bool tableInsert(HashTable* table, string s1, string s2) {
   int idx1 = hashFunction(s1);
   int idx2 = hashFunction(s2);
   bool result = false;
   if(idx1 == idx2){
        lock(table[idx1]->lock);
   }
   else{
        lock(table[idx1]->lock);
        lock(table[idx2]->lock);
   }
   if (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {

     insertToList(table->bins[idx1], s1);
     if(idx1 != idx2){
        unlock(table[idx1]->lock);
     }
     insertToList(table->bins[idx2], s2);
     if(idx1 != idx2){
        unlock(table[idx1]->lock);
     }
     else{
        unlock(table[idx2]->lock);
     }
     result = true;
   }

   return result;
}
```

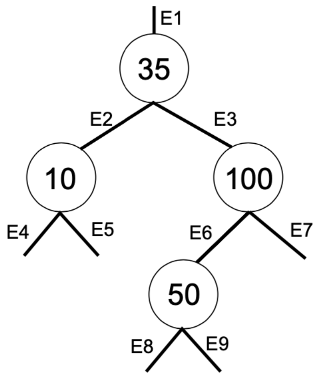
## Problem 3

1. L1 → U1 → L3 → U3 → L6 → U6 → L8 →U8

2. L1 →L3 → L7 →L11 →U11 →L10 →U10 →U7 →U3 →U1

3. (a) A tree can be left with 4 nodes if the remove max remove E7 when 140 is attached to E10. During insertion(140) the sequence of locking is: L1 → U1 → L3 → U3 → L7 → U7 → L10 →U10, during removal of max (150) the sequence is: L1 →L3 → L7 →L11 →U11 →L10 →U10 →U7 →U3 →U1. This it is possible that during insertion of 140, when thread 1 has a lock on edge 10, thread 2 first obtains lock on edge 10 and assigns the node of E7 to NULL. Then thread 2 unlocks E10 and E10, E11 are not in the BST tree. Then thread 1 does L10, and sets the val to 140 and unlocks E10. Then thread 2 proceeds to remove the node and unlocks E7, E3, E1. This leaves the tree with only 4 nodes.

   (b) The tree will look like the following image:



4. The sequence for remove max is in part 3.b. Since it is fine grain locking, thread 1 will hold Lock 1 during removal of node 150 while thread 2 has to wait to obtain the lock. During removal, val will be assigned 150. When the lock is released, then thread 2 will be able to obtain lock 1 and insert the value into the BST. Thus the only possible value is 150.

5. To solve the problem in part 3.C, we need to ensure that the left edge of the rightmost node should be first acquired by this thread doing insertion.

```
bool BST::insert_sync(Edge *e, int val) {
    Node *n = e->get();
    if (n == NULL) {
        e->set(new Node(val));
        e->unlock();
        return true;
    }
    if (n->value == val) {
        e->unlock();
        return false;
    }
    Edge *next = val < n->value ? &n->left : &n->right;
    next->lock();
    e->unlock();
    return insert_sync(next, val);
}
```