Stanford CS149: Parallel Computing
Written Assignment 4 SOLUTIONS

Warming Up... A Few Conceptual Questions

Problem 1. (30 points):

A. (10 pts) In class we talked about a basic lock implementation like this:

```
void lock(int* lock) {
   while (CAS(lock, 0, 1) == 1) {}
}

void unlock(int* lock) {
   *lock = 0;
}

// As a reminder CAS() performs this logic atomically
int CAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

Consider a situation where many threads, each running on a different core in a system that **implements cache coherence using the MSI protocol**, are attempting to acquire the lock. Please describe why it can be the case that the processor that is executing the thread that **is holding the lock IS NOT the processor whose cache holds the cache line containing the variable `lock` in the M state.** Please keep in mind that a `CAS()` is always a write operation from the perspective of cache coherence.

Solution: "holding the lock" and having exclusive access to the cache line containing the lock variable are two different things. Thread T0 might have acquired the lock and be executing in a critical section. While T0 is in its critical section, other threads (e.g., T1) may attempt to acquire the lock using `CAS()`. The `CAS()` by T1 will fail (since the value in memory is 1), but the result will be the line is loaded into the M state in T1's cache, and invalidated in T0's cache.

B. (10 pts) Consider the following code, where a lock, implemented using compare and swap (CAS) is used to make the operation of incrementing the variable x atomic.

```
void lock(int* l) {
  while (CAS(l, 0, 1) == 1);
}

void unlock(int*) {
  *l = 0;
}

int x;  // shared counter variable
int l;  // lock variable

// per-thread code
lock(&l);
x = x + 1;
unlock(&l);
```

Imagine this code running on a system that relaxes both WRITE AFTER WRITE and READ AFTER WRITE memory orderings. Consider the case where x is initialized to 0, and both thread 1 and thread 2 attempt to atomically increment x using the code above. Assume thread 1 acquires the lock first. Why is it possible for thread 2 to observe that x=0 when it later acquires the lock and enters the critical section. (You may assume that each invocation of CAS is treated as a write by the coherence protocol.)

Solution: Thread 1 writes to L, reads from X, writes to X, then writes to L. Because of relaxed consistency, thread 2 may observe thread 1's write to L (unlocking the lock), before observing thread 1's write to X. As a result, after thread 2 writes to L to take the lock, its read of X in the critical section yields 0, so it's subsequent store to X stores the value 1. Uh oh!

C. (10 pts) (One more question about Spark.) Consider the following program written using Spark RDDs, in a C-like syntax. Assume that `readRDDFromFile()` generates an RDD with elements of type `int` by reading numbers from a file, and that the functions `addOne()` and `addTwo()` are defined as given below. **You may also assume that `map()`, `readRDDFromFile()`, and `writeRDDToFile()` are THE ONLY transformations allowed on RDDs.**

```
int addOne(int x) { return x+1; }
int addTwo(int x) { return x+2; }

RDD r1 = readRDDFromFile();
RDD r2 = r1.map(addOne);
RDD r3 = r2.map(addTwo);
writeRDDToFile(r3);
```

Assume that there are N numbers in the file, and consider two potential implementations of this program. In the code below, `readIntFromFile()` and `writeIntToFile()` read/write exactly one integer to/from the file.

```
// IMPLEMENTATION 1

int array1[N];
int array2[N];
int array3[N];

for (int i=0; i<N; i++)
   array1[i] = readIntFromFile();
for (int i=0; i<N; i++)
   array2[i] = addOne(array1[i]);
for (int i=0; i<N; i++)
   array3[i] = addTwo(array2[i]);
for (int i=0; i<N; i++)
   writeIntToFile(array3[i]);

// IMPLEMENTATION 2

for (int i=0; i<N; i++) {
  writeIntToFile(addTwo(addOne(readIntFromFile())));
}
```

The second implementation computes elements of the three RDDs in a different order than the first implementation. It also clearly uses far less memory than the first. Are both implementations correct implementations of the Spark RDD abstraction? (In other words do they both compute the expected result?) If your answer is yes, please describe WHAT properties of RDDs and RDD transformations allow for both of these two different implementations. If your answer is no, please describe why. **(Please ignore robustness to node failure in this problem.)**

Solution: Yes, both implementations are correct. The first implementation computes (and stores in memory) all elements of one RDD before moving on to the next RDD. The second implementation reordered the computation to immediately consume the value for one RDD right after it is produced. The Spark program defines HOW to compute each element of each RDD, but it does not specify WHEN those elements must be computed. Since the only operations on RDDs are `map()` *and file I/O, the Spark program has the ability to execute processing of RDD elements in different orders (provided the ith element of r3 is computed after the ith element of r2, etc.)*

Hash Table Parallelization

Problem 2. (30 points):

A. (10 pts) Consider the following sequence of locking/unlocking operations by two threads.

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

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

// critical section           // critical section

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

Assuming that both threads must acquire all three locks prior to entering the critical section, please describe the **correctness problem** that can occur when running these two threads. Please also describe a modification to the code that fixes the problem, while preserving mutual exclusion (protects the critical section).

Solution: Deadlock. To fix this, each thread should acquire locks in the same order.

B. (20 pts) Below you will find an implementation of a hash table (a linked list per bin). The hash table has a function called `tableInsert` that takes two strings, and inserts both strings into the table **only if neither string already exists in the table**. Please implement `tableInsert` below in a manner that enables maximum concurrency. You may add locks wherever you wish. (Update the structs as needed.) To keep things simple, your implementation SHOULD NOT attempt to achieve concurrency within an individual list (notice we didn't give you implementations for `findInList` and `insertInList`). **Careful, things are a little more complex than they seem. You should assume nothing about `hashFunction` other than it distributes strings uniformly across the 0 to `NUM_BINS` domain. (HINT: deadlock!)**

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

struct HashTable {
   Node* bins[NUM_BINS];      // each bin is a singly-linked list
   Lock  binLocks[NUM_BINS];  // lock per bin
};

int   hashFunction(string str);           // maps strings uniformly to [0-NUM_BINS]
bool  findInList(Node* n, string str);    // return true if 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 onlyOne = false;
   bool result = false;

   // be careful to avoid deadlock due to (1) creating a circular wait or
   // (2) due to the same thread taking the same lock twice
   if (idx1 < idx2) {
     lock(binLocks[idx1];
     lock(binLocks[idx2];
   } else if (idx1 > idx2) {
     lock(binLocks[idx2];
     lock(binLocks[idx1];
   } else {
     lock(binlocks[idx1];
     onlyOne = true;
   }

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

   unlock(binLocks[idx1]);
   if (!onlyOne)
     unlock(binLocks[idx2];
   return result;
}
```

A Concurrent Binary Search Tree

Problem 3. (40 points):

In this problem you'll work with a version of a binary search tree (BST where **locks are associated with the edges of the tree, rather than the nodes**. Edges are represented as a C++ class `Edge`, declared as follows:

```
class Edge {
 private:
    Node *n;           // Pointer to BST node reachable along edge (or NULL)
    Lock plock;        // Lock associated with arc
 public:
    Node *get();       // Retrieve node pointer
    void set(Node *n); // Set node pointer
    void lock();       // Acquire lock
    void unlock();     // Release lock
};
```

The node data structure has a per-node value, plus edges to its two children

```
class Node {           // Nodes in BST
  public:
    Edge left, right;  // Edges to subtrees
    int value;         // Node value

    Node(int v) {      // constructor
        value = v;
        left.set(NULL);
        right.set(NULL);
    }
};
```

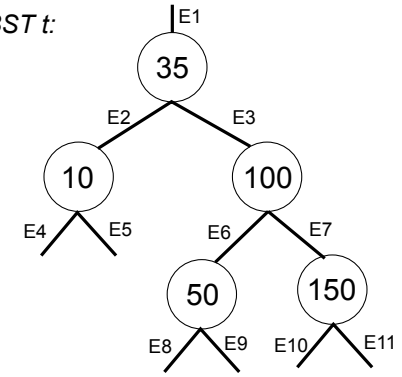
and the tree contains an "edge" to the root: (For an empty tree, the `n` field of the root edge is `NULL`.)

```
class BST {     // BST representation
 private:
    Edge root;
    public:
    // Insert value into BST
    bool insert(int val);

    // Remove maximum value node from BST, and assign its value to *val.
    // Return false if empty.
    bool remove_max(int *val);
};
```

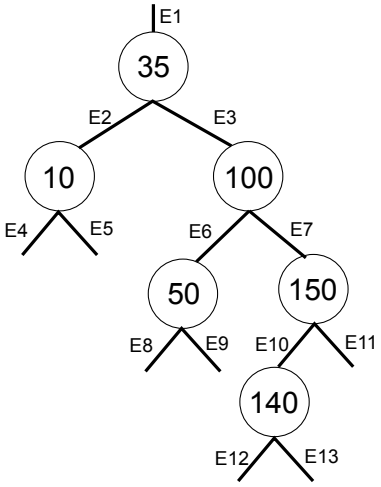
The following BST, which we will call `t` includes labels for all of its arcs. Notice that in a binary search tree, the left subtree of a node n contains nodes with values LESS than N, and the right subtree of a node n contains nodes with values GREATER than n.

As a reference, a correct insertion of the value 140 into t would yield the tree on the bottom-left. A correct removal of the maximum value would result in the tree on the bottom-right. We also show the result of removing the maximum element twice.

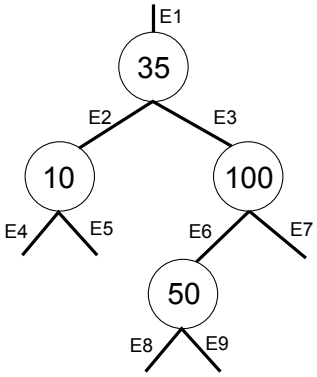
original BST t:



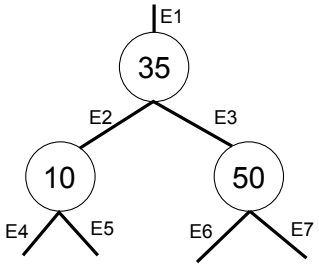
after inserting 140 to t:



after removing max from t:



after removing max from t, and then removing max again:



The following is a function for inserting elements into the tree. **It is intended to be thread safe (but may or may not be).**

```
// Top level insertion code (insert val into BST)
bool BST::insert(int val) {
    bool result = false;

    root.lock();
    result = insert_sync(&root, val);

    return result;
}

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

The following is a function for removing the maximum element in the tree. (Notice it always traverses to the right child until there are no more right children.) **It is intended to be thread safe (but may or may not be).**

```
// Top level remove code. Returns true if a node exists, and fills in val
bool BST::remove_max(int* val) {
  root.lock();
  return remove_max_sync(root, *val);
}

// removal subroutine
bool BST::remove_max_sync(Edge *e, int *val) {
    Node *n = e->get();
    if (n == NULL) {
        e->unlock();
        return false;
    }
    Edge *next = &n->right;
    next->lock();

    bool found = remove_max_sync(next, val);

    if (!found) {
        // Current node holds the maximum value since there
        // is no right child

        *val = n->value;

        // Replace this node with its left subtree
        Edge *left = &n->left;
        left->lock();
        e->set(left->get());
        left->unlock();
        delete n;
    }
    e->unlock();
    return true;
}
```

A. (5 pts) For BST `t`, assume a thread executes the call `t.insert(40)`. What sequence of lock acquisitions and releases would it cause to occur? (Use the notation `L1` to indicate locking of edge E1, `U2` to indicate unlocking of edge E2, etc.)

Solution: L1, U1, L3, U3, L6, U6, L8, U8

B. (5 pts) For the original BST `t` (without any additional insertions), assume a thread executes the call `t.remove_max()`. What sequence of lock acquisitions and releases would occur?

Solution: L1, L3, L7, L11, U11, L10, U10, U7, U3, U1

C. (10 pts) Starting with BST `t`, suppose two threads execute the following:

Thread 1: `t.insert(140);`

Thread 2: `int v; t.remove_max(&v);`

Assume that Thread 1 acquires the lock on edge E1 first. Identify sequences of actions by the two threads that could cause the resulting tree to contain only four nodes, and then answer the following:

(a) (5 pts) Describe the specific locking, unlocking, and update operations: *Solution:*

- *Thread 1: L1, U1, L3, U3, L7, L7 Now have e = E7, next = E10*
- *Thread 2: L1, L3, L7, L11, U11, L10 Set E7 to point to NULL. U10, U7, U3, L1*
- *Thread 1: L10, Assign value 140. U10.*

As a result we've "lost" the addition of value 140.

(b) (5 pts) Draw (or describe in text) the resulting tree.

Solution: The tree would contain nodes 35 (root), 10 (left child of root) and 100 (right child of root), and 50 (left child of root's right child)

D. (5 pts) Starting with BST `t`, suppose two threads execute the following:

Thread 1: `int v; t.remove_max(&v);`

Thread 2: `t.insert(200);`

Assume Thread 1 acquires the lock on edge E1 first. List all possible value(s) that could be assigned to `v`. Explain why this is the complete set of possibilities.

Solution: Only value 150. Once Thread 1 acquires the lock on E1, it will not release it until the operation is completed.

E. (15 pts) Modify the insertion code below to eliminate the problem you identified earlier, **while still allowing fine-grained concurrency.**

Solution:

```
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);
}
```

PRACTICE PROBLEM 1: Understanding Instruction Interleavings (Some of Which are Relaxed)

Assume that `x` and `y` are memory locations and `r1` and `r2` are per-thread local registers, `M` is a lock (a mutex), and `T0` and `T1` are threads. For each of the following program fragments we want you to compute the number of possible final states of the system. (due to different interleavings) **For each unique final state give the values stored in memory (X,Y) and the registers (T0.r1, T0.r2, T1.r1, T1.r2).**

Assume all fragments start with the initial conditions:
`T0.r1=0, T0.r2=0, T1.r1=0, T1.r2=0, x=0, y=0`

You may assume sequential consistency at all times except for the final part, where we explicitly mention a relaxed consistency model.

Hint: We recommend that you number the instructions, then work out all possible interleavings of the instructions, and then determine the outcomes of those interleavings.

A.

Thread T0	Thread T1
`lock(M)`	`lock(M)`
`T0.r1 = x`	`x = 1`
`unlock(M)`	`y = 1`
	`unlock(M)`

Solution: Two possible outcomes:
1. T0.r1=0, T0.r2=0, T1.r1=0, T1.r2=0, x=1, y=1
2. T0.r1=1, T0.r2=0, T1.r1=0, T1.r2=0, x=1, y=1

B.

Thread T0	Thread T1
`T0.r1 = x`	`lock(M)`
	`x = 1`
	`y = 1`
	`unlock(M)`

Solution: Two possible outcomes:
1. T0.r1=0, T0.r2=0, T1.r1=0, T1.r2=0, x=1, y=1
2. T0.r1=1, T0.r2=0, T1.r1=0, T1.r2=0, x=1, y=1

C.

Thread T0	Thread T1

`T0.r1 = x`
`T0.r2 = y`
	`y = 1`
	`x = 1`
	`y = 2`

Solution: Five possible outcomes:
1. T0.r1=0, T0.r2=0, T1.r1=0, T1.r2=0, x=1, y=2
2. T0.r1=0, T0.r2=1, T1.r1=0, T1.r2=0, x=1, y=2
3. T0.r1=0, T0.r2=2, T1.r1=0, T1.r2=0, x=1, y=2
4. T0.r1=1, T0.r2=1, T1.r1=0, T1.r2=0, x=1, y=2
5. T0.r1=1, T0.r2=2, T1.r1=0, T1.r2=0, x=1, y=2

D. Assume total store ordering (TSO) relaxed consistency. **TSO relaxes read after write order.** Specifically: A processor running a thread can proceed with a read from address Y THAT IS AFTER a write to address X in program order before the write to X is complete and visible to all processors.

Thread T0	Thread T1
`y = 5`	`T1.r2 = y`
`T0.r1 = x`	`T1.r1 = x`
`T0.r2 = T0.r1 + 1`	`T1.r2 = T1.r1 + T1.r2`
`x = T0.r2`	`x = T1.r2`

Solution: T0 "goes first":
1. T0.r1=0, T0.r2=1, T1.r1=1, T1.r2=6, x=6, y=5 (T1 sees write to Y)
2. T0.r1=0, T0.r2=1, T1.r1=1, T1.r2=1, x=1, y=5 (T1 sees write to Y late)

T1 "goes first":
3. T0.r1=0, T0.r2=1, T1.r1=0, T1.r2=0, x=1, y=5

Others:
4. T0: T0.r1=5, T0.r2=6, T1.r1=0, T1.r2=5, x=6, y=5
5. T0: T0.r1=0, T0.r2=1, T1.r1=0, T1.r2=5, x=5, y=5

PRACTICE PROBLEM 2: Load Linked / Store Conditional

A common set of instructions that enable atomic execution is load linked-store conditional (LL-SC). The idea is that when a processor loads from an address using a `load_linked` operation, the corresponding `store_conditional` to that address will succeed only if no other writes to that address from another processor have intervened. Note that unlike `test_and_set` or `compare_and_swap`, which are single atomic operations, load linked and store conditional are different operations and the processor may execute other instructions in between these two operations. Pseudocode for these instructions is given below.

```
int load_linked(int* addr) {
  return *addr;
}

bool store_conditional(int* addr, int new_val) {
  if ( \* data in addr has not been written to by any processor *\
       \* since the last load_linked on addr                    *\ ) {
    *addr = new_val;
    return true;
  } else {
    return false;
  }
}

Example usage (atomically read value of x and replace with f(x)):

int x;
load_linked(&x);
int y = f(x);  // do stuff with x here
store_conditional(&x, y);
```

A. Implement a spin lock using LL and SC primitives. (Your implementation can assume that threads behave reasonably, and will not attempt to unlock a lock they they have not previously acquired.)

```
void Lock(int* l) {
  while (!(load_linked(l) == 0 && store_conditional(l, 1)));
}

void Unlock(int* l) {
  *l = 0;
}
```

B. Now we'd like you to implement load-linked (LL) and store conditional (SC). **Assume you have a multi-core processor that already implements cache coherence using the MSI protocol.** How would you extend the behavior of the cache to implement LL and SC instructions? Specifically describe:

- Any additional state you want to add to cache lines. (Recall that with MSI, the cache already tracks whether a line is in the M, S, or I state.)
- What does the cache do to implement an LL operation. (How does it behave differently than a normal read, which moves the line to the S state, or keeps the line in the M state?)
- What does the cache do for a SC operation? How does the system use the cache line state's to determine if the SC succeeds or fails?
- Why your solution would correctly cause an SC to fail if there has been another intervening write to the line by another processor.
- **NOTES: Your implementation of LL/SC should be able to support many pending LL's for many cache lines at the same time (e.g., as many cache lines as the cache can hold) – solutions that say "add a line to the cache that is a special line for one active LL/SC are not accepted".** Your implementation of SC should fail if there can been any intervening writes by the local or remote processors to the line since the LL. You do not need to handle the case where a line with a pending LL is evicted from the cache due to cache capacity or cache conflicts between the LL and SC.

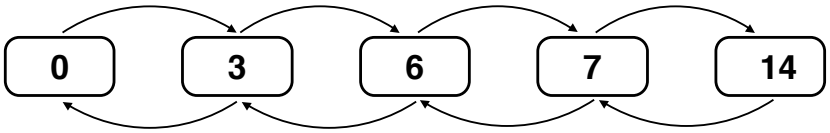
Solution: LL should load the line in the M state (exclusive) and set an extra "LL bit". (If LL is executed on a line in the M state, then LL only turns on the "LL bit".) Then SC should check to see if the LL is set to determine whether the SC succeeds or fails. A regular write (not an SC) by the local processor to a line with the LL bit set clears the bit. Invalidating a line with the LL bit set clears the bit. Therefore, any write by another processor will cause an invalidate (due to MSI protocol, causing a later SC to fail.

One might think that just implementing LL by loading the line in the M state would be sufficient, with SC just checking to make sure the line is in the M state on the SC. However, this is insufficient since it doesn't guarantee the line was in the M state for the entire duration between the LL and SC. (The line could have been dropped, and then read back in by a write by the local core. It would be impossible to determine if the line was in the M state due to no intervening writes, or if remote writes occurred, and then a local write reclaimed the line in the M state.

PRACTICE PROBLEM 3: Concurrent Linked Lists

Consider a **SORTED doubly-linked list** that supports the following operations.

- `insert_head`, which traverses the list from the head. The implementation uses hand-over-hand locking just like in class.
- `delete_head`, which deletes a node by traversing from the head, using hand-over-hand locking just like in class.
- `insert_tail`, which traverses the list **backwards from the tail** to insert a node using hand-over-hand locking in the opposite order as `insert_head`.



A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - Test 1: `insert_head(2)`, `delete_head(14)`
   - Test 2: `insert_head(12)`, `delete_head(6)`
   - Test 3: `insert_head(13)`, `insert_tail(4)`

   The first two unit tests complete without error, but the third test goes badly and it does not terminate with the right answer. Describe what behavior is observed and why the problem occurs. (All unit tests start with the list in the state shown above.)

   Solution: Deadlock occurs. The insert operation proceeding from the head holds a lock and attempts to acquire the lock for the next node in the list. The insert operation beginning at the tail holds a lock and attempts to lock the previous node in the list. This is a classic hold and wait with circular dependencies problem, and deadlock can occur. Imagine one thread holding the lock on 3 and trying to acquire the lock for 6. Imagine the other thread holding the lock on 6 and trying to acquire the lock for 3.

B. Imagine that locks in this system supported not only `lock()` and `unlock()`, but the ability to query the state of the lock via the call `trylock()` (this call takes the lock if the lock is free, but immediately returns false if the lock is currently locked – it does not block). Given this functionality, describe a fix to the problem you identified in part A? **Your answer should avoid livelock, but it is acceptable to allow for the possibility of starvation.**

Solution: The simplest solution is to replace every call to `lock()` *with* `trylock()`*, and when* `trylock()` *fails, abort and restart the operation. Of course, this will almost certainly turn deadlock into livelock, and so there are a number of ways to avoid the livelock problem.*

You could have operations from the head of the list take precedence over those from the tail (that is, only operations from the tail abort and restart). This scheme could starve operations from the tail. Instead, you could make livelock less likely with some form of random back-off scheme. Another idea is to make the data structure could operate in "phases," where in one phase it only allows operations from the head, and in a second phase allows operations from the tail. For example, in the "head phase" you could maintain a bit indicating if there were operations pending from the head, and enqueue threads wanting to operate on the tail. When the operations executing from the head complete, then the data structure flips to "tail phase" and only allows operations from the tail (enqueue operations from the head.)

PRACTICE PROBLEM 4: Tricky Little Graphs

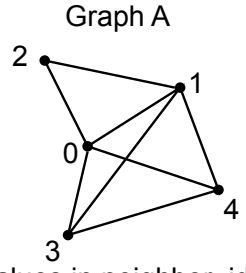
```
struct Graph_node {
   Lock   lock;
   float  value;
   int    num_edges;    // number of edges connecting to node (its degree)
   int*   neighbor_ids; // array of indices of adjacent nodes
};

// a graph is a list of nodes, just like in assignment 3
Graph_node graph[MAX_NODES];
```

Consider the undirected graph representation shown in the code above.

Your boss asks you to write a program that atomically updates each graph node's `value` field by setting it to the average of all the values of neighboring nodes. The program must obtain a lock on the current node and all adjacent nodes to perform the update. It does so as follows...

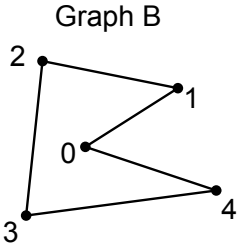
```
void update(int id) {
    Graph_node* n = &graph[id];
    LOCK(n->lock);
    for (int i=0; i<n->num_edges; i++)
      LOCK(graph[n->neighbor_ids[i]].lock);
    // now perform computation...
```

Consider running the update code in parallel on nodes 0 and 1 in the two graphs below. For each graph, determine if deadlock occurs. Please describe why or why not. (Note: we do not ask you to solve the deadlock problem, but think about you might avoid it, assuming you must still only use locks. Consider changing the order in which you take the locks.



Values in neighbor_ids[]:

0: 3 4 2 1 3: 0 4 1
1: 0 2 3 4 4: 0 3
2: 0 1



Values in neighbor_ids[]:

0: 1 4 3: 2 4
1: 0 2 4: 0 3
2: 1 3

Solution: Deadlock occurs in graph both Graph A and Graph B. The deadlock problem occurs because there is a hold-and-wait situation with circular dependencies. In graph A for example, one possible deadlock scenario is for the thread updating node 0 to acquire locks 0,3,4,2 and for the thread updating node 1 to acquire lock 1. Likewise in graph B, one possible deadlock scenario is for the thread updating node 0 to acquire a lock on 0, and the thread updating node 1 to lock on node 1. A simple solution would be, before processing a node, make a list of adjacent vertices and the current node, sort the list, and then take locks in that order. Since all threads would request locks in the same order, you've removed the circular dependency.