Stanford CS149: Parallel Computing
Written Assignment 5

Comparing and Swapping

Problem 1. (50 points):

The logic of atomic compare-and-swap is given below (Keep in mind that atomic CAS is an operation that carries this sequence of logic **atomically**.)

```
int CAS(int* addr, int compare, int val) {
   int old = *addr;
   *addr = (old == compare) ? val : old;
   return old;
}
```

Consider a program where multiple threads cooperate to compute the sum of a large list of **SIGNED INTEGERS**.

```
// global variables shared by all threads

int values[N];  // assume is very large
int sum = 0;

//////////////////////////////////////////////////////

// per thread logic (assume thread_id, num_threads are defined as expected)

for (int i=thread_id; i<N; i+=num_threads) {
  sum += values[i];
}
```

A. (15 pts) There is a correctness problem with the above C code when `num_threads > 1`. Using CAS, please provide a fix so that the code computes a correct parallel sum. To make this problem a little trickier, there are two rules: **(1) You must provide a non-blocking (lock-free) solution. (2) Each iteration of the loop should update the global variable `sum`. You are not allowed to accumulate a partial sum locally and reduce the partial sums after the loop.**

```
for (int i=thread_id; i<N; i+=num_threads) {




}
```

B. (10 pts) In the original code above, `sum += values[i]` is a read-modify-write operation. (The code reads the value of `sum`, computes a new value, then updates the value of `sum`. One way to fix the code above is to make this sequence of operations atomic (i.e., no other writes to `sum` occur in between the read and the write by one thread. **Does your solution in Part A maintain atomicity of the read-modify-write of `sum`?** (why or why not?) If it does not, why are you confident your solution is correct? **Keep in mind the numbers to be summed are signed integers.**

C. (15 pts) Now imagine the problem is changed slightly to include a new shared variable `count`, representing the number of inputs that have been summed together to yield the value in `sum`. `count` and `sum` must always stay in sync. **In other words, it should not be possible to ever observe a value of `count` and `sum` such that the sum of the first `count` elements of `vals` don't add up to `sum`.**

```
// global variables shared by all threads
int values[N];
int sum = 0;
int count = 0;

////////////////////////////////////////////////////

// per thread logic
for (int i=thread_id; i<N; i+=num_threads) {
  sum += values[i];
  count++;
}
```

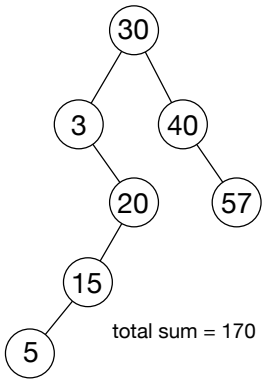
Using only CAS, please provide a correct, thread safe version of the code. **This problem is independent of parts A and B.** However like part A, you must update the shared variables each iteration of the loop (no local partial sums). **Unlike part A, you can take any approach to synchronization, even if it IS NOT LOCK FREE.**

D. (10 pts) Imagine you had a double-word CAS (operating on 64-bit values in memory), implement a lock-free solution to part C. **Also answer the following question: is the shared variable update now guaranteed to be atomic? (why or why not?)**. For simplicity, assume double-word CAS returns true on success, false otherwise.

Transactions on Trees

Problem 2. (50 points):

Consider the binary search tree illustrated below.



The operations `insert` (insert value into tree, assuming no duplicates) and `sum` (return the sum of all elements in the tree) are implemented as transactional operations on the tree as shown below.

```
struct Node {
   Node *left, *right;
   int value;
};
Node* root;  // root of tree, assume non-null

void insertNode(Node* n, int value) {
  if (value < n->value) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n->value) {
    if (n->right == NULL)
       n->right = createNode(value);
    else
       insertNode(n->right, value);
  }  // insert won't be called with a duplicate element, so there's no else case
}

int sumNode(Node* n) {
   if (n == null) return 0;
   int total = n->value;
   total += sumNode(n->left);
   total += sumNode(n->right);
   return total;
}

void insert(int value) {  atomic { insertNode(root, value); }   }
int sum()              {  atomic { return sumNode(root); )      }
```

Consider the following four operations are executed against the tree in parallel by different threads.

```
insert(10);
insert(25);
insert(24);
int x = sum();
```

A. (8 pts) Consider different orderings of how these four operations could be evaluated. Please draw all possible trees that may result from execution of these four transactions. (Note: it's fine to draw only subtrees rooted at node 20 since that's the only part of the tree that's effected.)

B. (7 pts) Please list all possible values that may be returned by `sum()`.

C. (7 pts) Do your answers to parts A or B change depending on whether the implementation of transactions is optimistic or pessimistic? Why or why not?

D. (7 pts) Consider an implementation of **lazy, optimistic** transactional memory that manages transactions at the granularity of tree nodes (the read and writes sets are lists of nodes). Assume that the transaction `insert(10)` commits when `insert(24)` and `insert(25)` are currently at node 20, and `sum()` is at node 40. Which of the four transactions (if any) are aborted? **Please describe why.**

E. (7 pts) Assume that the transaction `insert(25)` commits when `insert(10)` is at node 15, `insert(24)` has already modified the tree but not yet committed , and `sum()` is at node 3. Which transactions (if any) are aborted? **Again, please describe why.**

F. (7 pts) Now consider a transactional implementation that is **pessimistic** with respect to writes (check for conflict on write) and **optimistic** with respect to reads. The implementation also employs a "writer wins" conflict management scheme – meaning that the transaction issuing a conflicting write will not be aborted (the other conflicting transaction will). Describe how a **livelock problem** could occur in this code.

G. (7 pts) Give one livelock avoidance technique that an implementation of a pessimistic transactional memory system might use. You only need to summarize a basic approach, but make sure your answer is clear enough to refer to how you'd schedule the *transactions*.

A Concurrent Binary Search Tree (EXTRA CREDIT)

Problem 3. (10 points):

Consider a representation for binary search trees (BSTs), in which locks are associated with the arcs in the tree, rather than the nodes. These arcs are represented as a C++ class `Arc`, declared as follows:

```
class Arc {
 private:
    Node *np;          // Pointer to BST node (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 value, plus arcs to its two children

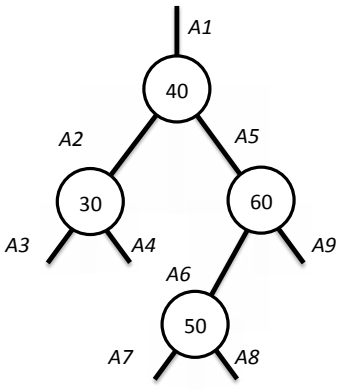
```
class Node {           // Nodes in BST
  public:
    Arc left, right;   // Arcs to subtrees
    int value;         // Node value

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

and the tree contains an arc to the root: (For an empty tree, the `np` field of the root arc is `NULL`.)

```
class BST {     // BST representation
 private:
    Arc root;
 public:
    bool insert(int val);       // Insert value into BST
    // Remove maximum from BST, and assign its value to *valp.
    // Return false if empty.
    bool remove_max(int *valp);
};
```

The following BST, which we will call "`t`," includes labels for all of its arcs:



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

```
bool BST::insert_synch(Arc *p, int val) {
    Node *np = p->get();
    if (np == NULL) {
        p->set(new Node(val));
        p->unlock();
        return true;
    }
    p->unlock();
    if (np->value == val) {
        return false;
    }
    Arc *nextp = val < np->value ? &np->left : &np->right;
    nextp->lock();
    return insert_synch(nextp, val);
}
```

The following is a function for removing the maximum element in the tree. It is intended to be thread safe (but may or may not be).

```
bool BST::remove_max_synch(Arc *p, int *valp) {
    Node *np = p->get();
    if (np == NULL) {
        p->unlock();
        return false;
    }
    Arc *nextp = &np->right;
    nextp->lock();

    bool found = remove_max_synch(nextp, valp);

    if (!found) {
        // Current node holds the maximum value
        *valp = np->value;
        // Replace this node with its left subtree
        Arc *leftp = &np->left;
        leftp->lock();
        p->set(leftp->get());
        leftp->unlock();
        delete np;
    }
    p->unlock();
    return true;
}
```

A. (1 pt) Fill in the following code for insertion into the BST by calling function `insert_synch`:

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




    return result;
}
```

B. (1 pt) For BST `t`, assume a thread executes the call `t.insert(70)`. What sequence of lock acquisitions and releases would it cause to occur? (Use the notation `L1` to indicate locking of arc A1, `U2` to indicate unlocking of arc A2, etc.)

C. (1 pt) For BST `t` (without any additional insertions), assume a thread executes the call `t.remove_max()`. What sequence of lock acquisitions and releases would occur?

D. (3 pts) Starting with BST `t`, suppose two threads execute the following:

Thread 1: `t.insert(70);`

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

Assume that Thread 1 acquires the lock on arc A1 first. Identify sequences of actions by the two threads that could cause the resulting tree to contain only 3 nodes, and then answer the following:

(a) (1 pt) Describe the specific locking, unlocking, and update operations:

(b) (1 pt) Draw the resulting tree.

(c) (1 pt) What value was returned by `remove_max`?

E. (1 pt) Starting with BST `t`, suppose two threads execute the following:

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

Thread 2: `t.insert(70);`

Assume Thread 1 acquires the lock on arc A1 first. List all possible value(s) that could be assigned to `v`. Explain why this is the complete set of possibilities.

F. (1 pt) Modify the insertion code below to eliminate the problem you identified earlier, **while still allowing fine-grained concurrency.**

```
bool BST::insert_synch(Arc *p, int val) {
    Node *np = p->get();
    if (np == NULL) {
        p->set(new Node(val));
        p->unlock();
        return true;
    }
    p->unlock();
    if (np->value == val) {
        return false;
    }
    Arc *nextp = val < np->value ? &np->left : &np->right;
    nextp->lock();
    return insert_synch(nextp, val);
}
```

G. (2 pts) Modify the removal code below to **allow greater concurrency**, while avoiding any races with preceding or following operations.

```
bool BST::remove_max_synch(Arc *p, int *valp) {
    Node *np = p->get();
    if (np == NULL) {
        p->unlock();
        return false;
    }
    Arc *nextp = &np->right;
    nextp->lock();

    bool found = remove_max_synch(nextp, valp);

    if (!found) {
        *valp = np->value;
        Arc *leftp = &np->left;
        leftp->lock();
        p->set(leftp->get());
        leftp->unlock();
        delete np;
    }
    p->unlock();
    return true;
}
```