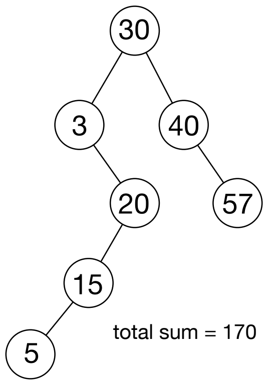
CMU 15-418/618: Parallel Computer Architecture and Programming
Practice Exercise 5 SOLUTIONS

Problem 1: Transactions on Trees

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) {
    if (n->left == NULL)
       n->left = createNode(value);
    else
       insertNode(n->left, value);
  } else if (value > n) {
    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. (2 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.)

There are only two. 20 –> 25 –> 24 or 20 –> 24 –> 25

B. (2 pts) Please list all possible values that may be returned by `sum()`.

Solution: 170, 180, 194, 195, 204, 205, 219, 229

C. (2 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?

Solution: Definitely not! The choice of how to ***implement*** *a transaction cannot change the* ***semantics*** *of the transactional abstraction.*

D. (2 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.**

Solution: Only `sum` is aborted since the write set of the committing transaction (which is node 5) conflicts with the read set of `sum`. Note that there is no conflict with the other insertions since they read no data written by `insert(10)`.

E. (2 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.**

Solution: In this case, `insert(10)` is aborted since 20 is in its read set, and it was modified by the committing transaction. `insert(24)` is also aborted since it's read and write sets conflict with the write set of the committing transaction. `sum` is not aborted since it hasn't progressed enough to reach node 20.

F. (2 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.

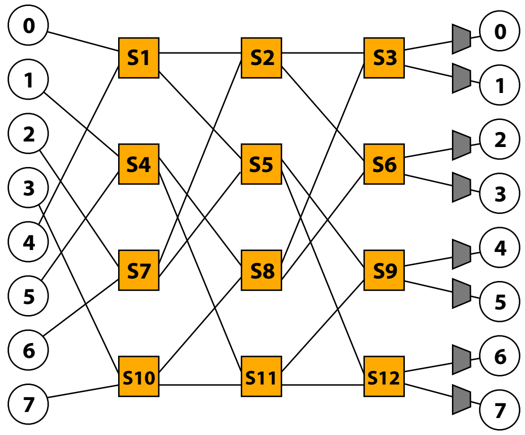
Solution: The problem is that `insert(24)` can write to `n->right` of node 20, which conflicts with `insert(25)`'s read/write of the same node. `insert(25)` will abort and restart, and then it's own update of `n->right` on the retry will cause `insert(24)` to abort if that transaction did not have time to commit.

G. (2 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*.

Solution: Any standard answer from the implementation of locks lecture would work, but in this context the solutions are used for implementing transactions, not locks: you could try random backoff, give priority to a transaction that's been aborted too many times in the past, put transactions that have aborted in a list and process the list serially (like a ticket lock), etc.

Problem 2: Interconnects

Consider sending two 256-bit packets in the Omega network below. Packet A is sent from node 4 to node 0, and packet B from node 6 to node 1. The network uses worm-hole routing with flits of size 32 bits. All network links can transmit 32 bits in a clock. **The first flit of both packets leaves the respective sending node at the same time. If flits from A and B must contend for a link, flits from packet A always get priority.**



A. (2 pts) What is the latency of transmitting packet A to its destination?

Solution: 11 cycles. 4 cycles for first flit to arrive. 7 additional cycles for the rest of the packet.

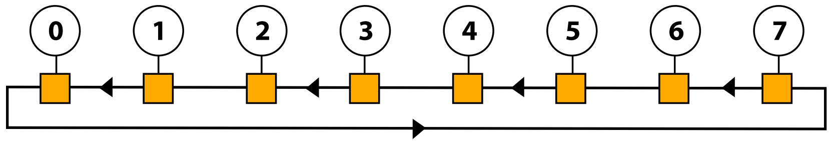
B. (2 pts) What is the latency of transmitting packet B to its destination? Please describe your calculation–switches are numbered to help your explanation.)

Solution: 19 cycles. The first flit gets to its destination 4 + 8 cycles (it must wait for 8 cycles for the shared link at switch S2). Then the rest of the packet arrives over the next 7 cycles.

C. (2 pts) If the network used store-and-forward routing, what would be the minimum latency transmitting one packet through the network? (Assume this packet is the only traffic on the network.)

Solution: 32 cycles. 8 cycles to transmit the packet over a link $\times$ four links between any two nodes on the network.

D. (2 pts) Now consider sending packet A from node 2 to node 0 and packet B from node 5 to 3 on the unidirectional ring interconnect shown below. Assuming the conditions from part A (32-bits send over a link per clock, worm-hole routing, same packet and flit sizes, both messages sent at the same time, packet A prioritized over packet B), what is the minimum latency for message A to arrive at its destination? Message B?



Solution: 9 cycles (2 cycles for the first flit to arrive, 7 more for the remainder of the packet.

E. (2 pts) **THIS QUESTION IS UNLRELATED TO THE PREVIOUS ONES.** Consider a parallel version of the 2D grid solver problem from class. The implementation uses a 2D tiled assignment of grid cells to processors. (Recall the grid solver updates all the red cells of the grid based on the value of each cell's neighbors, then all the black cells). Since the grid solver requires communication between processors, you choose to buy a computer with a crossbar interconnect. Your friend observes your purchase and suggests there there is another network topology that would have provided the same performance at a lower cost. What is it? (Why is the performance the same?)

Solution: In this application, each processor need only communicate with the processor that responsible for its left, right, top, bottom grid neighbors. The mesh topology provides all of these connections as point-to-point links, so the app will run just as well on a mesh as on a fully-connected crossbar. The mesh has $O(N)$ cost, where the crossbar has $O(N^2)$, where N is the number of processors. A 2D torus would also be a valid answer.

Problem 3: Concurrency Using Fine-Grained Locks

NOTE: THIS QUESTION IS FOR EXTRA PRACTICE ONLY–IT WILL NOT BE GRADED BY THE STAFF ALTHOUGH SOLUTIONS WILL BE PROVIDED.

NOTE: this problem uses the same setup as Problem 1, but does not depend on its answer. See Problem 1 for the setup, but don't worry if you weren't able to answer Problem 1.

A. Consider an alternative implementation of the thread safe binary search tree from the previous problem that uses fine-grained (per-node) locking to synchronize the `insert` operations. The code below is a correct implementation of `insert`. Notice that it does not use hand-over-hand locking. **Assuming that `insert` is the only operation allowed on the tree**, describe why it's okay for the implementation to be in the state where it holds no locks at certain points during traversal.

```
struct Node {
   Node *left, *right;
   Lock* lock;
   int value;
};

Node* root;  // root of tree, assume not null

void insertNode(Node* n, int value) {

  lock(n->lock);

  // assume n is not null
  if (value < n->value) {
    if (n->left == NULL) {
       n->left = createNode(value);
       unlock(n->lock);
    } else {
       unlock(n->lock);
       insertNode(n->left, value);
  } else if (value > n->value)
    if (n->right == NULL) {
       n->right = createNode(value);
       unlock(n->lock);
    } else {
      unlock(n->lock);
      insertNode(n->right, value);
    }
  }  // insert won't be called with a duplicate element,
     // so there's no else case

void insert(int value) {
  insertNode(root, value);
}
```

B. (10 pts) Consider a system that processes a queue of tree manipulation commands.

```
insert(10);
insert(25);
print sum();
insert(24);
insert(100);
print sum();
insert(1);
...
```

The system can execute the commands concurrently as desired, provided that **the results of all sum operations** are consistent with the commands being executed in SERIAL ORDER. That is, in the example above, the first print must reflect the insertion of 10 and 25, but it must not reflect the insertion of 24, 100, and 1. Give an implementation that respects these semantics and enables as much concurrency as possible.

This is a tricky problem. First try and find a solution that allows inserts() that come before sum() in the queue to execute concurrently with the sum(). We'll give good partial credit for that. For full credit (tricky) we're looking for a solution that allows sum() to execute concurrently with inserts() before and after it in the queue. Note your implementation will likely need to modify BOTH the implementation of insert() and sum().

Starter code is given on the next page to give you room.

```
struct Node {
   Node *left, *right;
   int value;
   Lock lock;
};

Node* root;  // root of tree

void insertNode(Node* n, int value) {

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

void insert(int value) {
  lock(root->lock);
  insertNode(root, value);
}

int sumNode(Node* n) {
  if (n == null)

     return 0;

  Node* left;
  Node* right;
  left = n->left;
  right = n->right

  if (n->left != NULL) {
    lock(n->left->lock);
  }
  if (n->right != NULL) {
    lock(n->right->lock);
  }
  unlock(n->lock);

  int leftSum;
  int rightSum;

  if (left)
    leftSum = sumNode(n->left);
  else
    leftSum = 0;
  if (right)
    leftSum = sumNode(n->right);
  else
    rightSum = 0;

  return n->value + leftSum + rightSum;
}

int sum() {
   lock(root->lock);
   return sumNode(root);
}
```

Explanation of the above solution: Full credit solutions prevents sum's from passing prior `insert` operations, and enabled subsequent inserts to operate on the tree concurrently with prior sums. While the solution provided here does not provide maximal concurrency between `sum` and subsequent `insert` operations, it was considered a full credit solution. Three key things to note:

(a) It was necessary to have hand-over-hand locking in `insert` to prevent sums from jumping ahead of previous inserts. (Getting this right was 1/2 credit.)

(b) In `sumNode`, we must have a lock on both left and right children if they are not null before releasing the parent. This is to prevent inserts behind us from passing the sum and inserting somewhere deeper in the tree.

(c) We must store pointers to the left and right child prior to releasing the parent. This is because in the edge case where either left or right is NULL, and we don't take a lock on that child, it could be the case that after releasing the parent, an insert comes in and inserts a new node as the previously NULL child. This means that when sum then calls sum() on that node, it will return a non zero value, which is incorrect. So we check if left and right were NULL prior to releasing the parent to determine whether or not to call sum on that child. (This was subtle so only one point was deducted if missed.)

Some other nice solutions that we saw used some form of versioning to allow more concurrency. These solutions added an additional field inside the `Node` struct that was basically a counter to keep track of the insertion at which that node was inserted into the tree. When insert gets called, they atomically incremented a global version counter and passed this into the `insertNode()` function. Inside `insertNode` they used hand over hand locking and when creating the new node, they put the version counter into this new node.

When `sum()` gets called, they take a lock around the global counter, read its value, and then release the lock. This value gets passed into the following calls to `sumNode()` and tells them that they should only sum the values of nodes with a version number less than or equal to that of the global counter. All nodes with a greater version counter we inserted after this call to `sum()`.

Inside `sumNode`, they locked the parent, then locked the left child, and then released the parent, and computed `leftSum`. Then they locked the right child and computed the `rightSum`. Note that in this implementation the point of locking the left and right nodes before proceeding down those respective paths is to ensure that `sumNode` does not jump ahead of any prior inserts but it DOES NOT ensure that inserts can't jump ahead of sums. We are ensured that sums can't jump ahead of inserts because `insertNode` does hand over hand locking so since sum must take a lock on a node before computing its sum, you know that sum can't do so until the inserts before it have released their locks on the nodes that sum is about to read. To make sure that `sumNode` does not sum up values from nodes inserted after the call to sum, the `sumNode` function will check the version number of a node before adding its value into the sum. If the version number is greater than the version number passed down through the calls to `sumNode`, then we don't add in the value and return 0, else we call `sumNode` on that node.