Stanford CS149: Parallel Computing
Written Assignment 4

Problem 1: Cache Coherence (30 pts)

Consider the following program which uses two threads of control. Note: the code is not meant to compute anything useful (in fact it is just moving zeros around!), however it is designed to get you to think about the behavior of a cache coherent system.

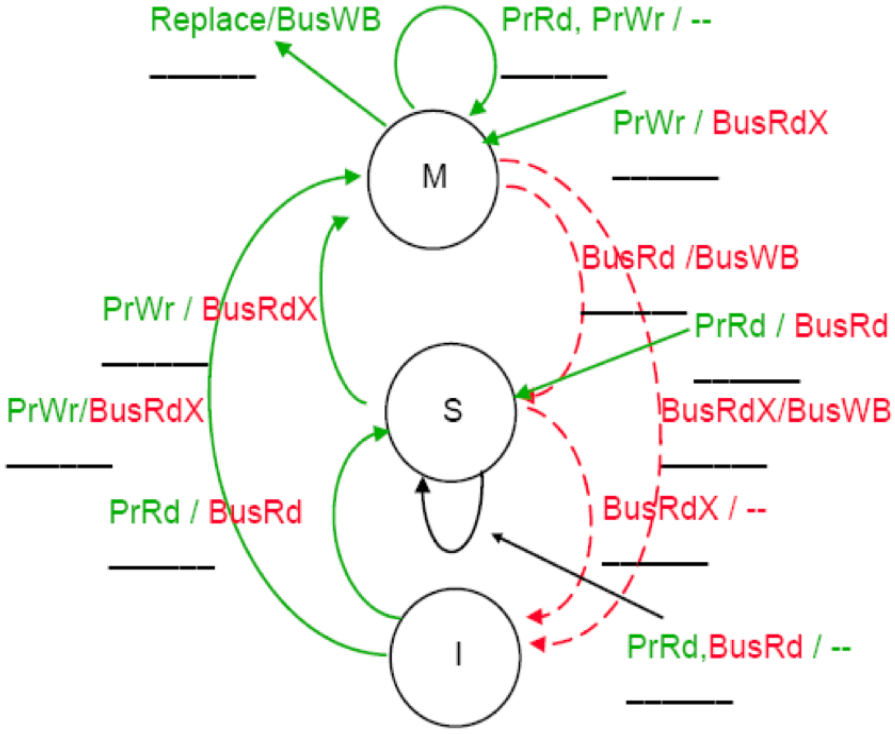
```
int array[8];   // this array is globally visible to all threads

void do_work(int id) {

   barrier();
   // <--- assume cache is invalidated here --->

   if (id == 0) {
     for (int i=0; i<8; i++)
       if (array[i] != 0)
         exit(1);
   }

   barrier();

   for (int i=0; i<2; i++) {
     array[4*i + 2*id] = array[4*i + 0 + 2*id] +
                         array[4*i + 1 + 2*id];
   }
}

int main() {
  // init array
  for (int i=0; i<8; i++)
    array[i] = 0;

  // spawn one thread
        std::thread t1 = std::thread(do_work, 0);
  do_work(1);
}
```

Assuming that code is run on a dual-core processor where:
- Each core has its own private cache
- Caches are fully-associative with four, 8-byte lines (32 byte caches)
- Caches implement the MSI protocol to implement coherence.

In your answer, assume that:
- You should only analyze references to the `array` variable
- No other operations manipulate the state of caches
- Only consider operations that occur after the line `<invalidate caches>`.

Question on next page...

Label the following MSI state diagram with the total transitions taken in the system after `<invalidate caches>`. To help you fill in the diagram, please write down the sequence of memory references and coherence protocol transitions that each core makes when running the code.



Problem 2: Those Pesky Locks (40 pts)

Consider a basic lock implementation that uses atomic compare and swap. Implementations of both the lock and the `atomicCAS` are given below.

```
// recall: this entire operation is carried out atomically
int atomicCAS(int* addr, int compare, int val) {
  int old = *addr;
  *addr = (old == compare) ? val : old;
  return old;
}

void lock(int* lk) {
  while (atomicCAS(lk, 0, 1) == 1);
}

void unlock(int* lk) {
  *lk = 0;
}
```

Consider the following OpenMP code where threads cooperatively compute the sum of a large array of N numbers. **In this code assume that `foo()` is a very expensive function that involves only arithmetic instructions.**

```
int mylock = 0;
int sum = 0;
int input[N];  // assume input is appropriately initialized

#pragma omp parallel for schedule dynamic
for (int i=0; i<N; i++) {
  lock(&mylock);
  sum += foo(input[i]);   // assume foo() is a function of only its input argument
  unlock(&mylock);
}
```

Imagine the code is run on a CPU with 16 cores that implements invalidation-based cache coherence. Each core has its own cache with 64-byte cache lines. Executing an `atomicCAS` is treated by the cache coherence protocol as a *write to the address storing the lock variable*.

A. (9 pts) Consider the case where the thread running on core 0 holds the lock, and is in the middle of executing the function `foo()`. (Recall `foo()` does not contains any memory instructions.) Describe what cache coherence traffic is occurring while core 0 runs `foo()`.

B. (9 pts) Give code for an improved implementation of `lock()` that significantly reduces cache coherence traffic while core 0 is executing `foo()`. Describe why this implementation is better.

C. (9 pts) You run the code above on the 16-core CPU and barely observe any speedup. Please explain why, and then suggest a fix to the code that still uses locks, but should dramatically increase speedup.

.

D. (9 pts) **(THIS QUESTION IS INDEPENDENT FROM A-C, BUT WE DO RECOMMEND YOU REVIEW THE INFORMATION ABOUT THE CPU'S CACHE.)** Now assume `foo()` is a very cheap to compute function. Your friend looks at your solution from the previous subproblem, and says, since `foo()` is so cheap, the overhead of frequent synchronization is high. I'd rather write the code without synchronization in the inner loop. Your friend comes up with the following solution.

```
int partial[16];
int sum = 0;
int input[N];  // assume input is appropriately initialized

for (int i=0; i<16; i++)
  partial[i] = 0;

#pragma omp parallel for schedule dynamic
for (int i=0; i<N; i++) {
  // omp_get_thread_num() returns the thread id of the
  // thread running the current iteration of the loop.
  // On this machine it will return values [0,15]
  int which = omp_get_thread_num();
  partial[which] += foo(input[i]);   // assume foo() is a function of only its input argument
}
for (int i=0; i<16; i++)
  sum += partial[i];
```

They run the code and are a little surprised they don't get perfect speedup. What is the reason why?

E. (4 pts) Consider the original code in the problem (without any modifications made by you to the `lock` implementation or to the body of the `for` loop). However now assume that instead of performing many arithmetic instructions, `foo()` instead performs many *memory instructions*. You run the code on the 64-core processor many times, each time using a different number of threads. **Assume that the CPU uses snooping to implement its invalidation-based coherence.** You measure the performance of the entire execution of the for loop, notice that execution time actually INCREASES with higher thread counts. Why might this be the case?

Problem 3: Hash Table Parallelization (30 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


};

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 (!findInList(table->bins[idx1], s1) &&
       !findInList(table->bins[idx2], s2)) {



     insertToList(table->bins[idx1], s1);



     insertToList(table->bins[idx2], s2);



     result = true;
   }




   return result;
}
```

The following problems are PRACTICE PROBLEMS and will not be graded.

Practice Problem 1: Memory Consistency

Consider the following program which has four threads of execution. In the figure below, the assignment to `x` and `y` should be considered stores to those memory addresses. Assignment to `r0` and `r1` are loads from memory into local processor registers. (The print statement does not involve a memory operation.)

Processor 0	Processor 1	Processor 2	Processor 3

`x = 1`	`y = 1`	`r0 = y`
		`r1 = x`
		`print (r0 & ~r1)`
			`r0 = x`
			`r1 = y`
			`print (r0 & ~r1)`

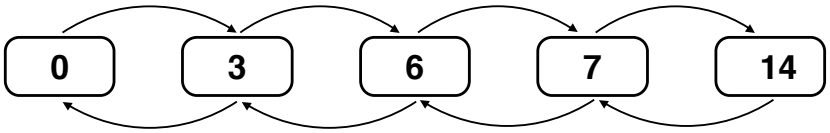
- Assume the contents of addresses `x` and `y` start out as 0.
- Hint: the expression `a & ~b` has the value 1 only when `a` is 1 and `b` is 0.

You run the program on a four-core system and observe that both processor 2 and processor 3 print the value 1. Is the system sequentially consistent? Explain why or why not?

Practice Problem 2: 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`.



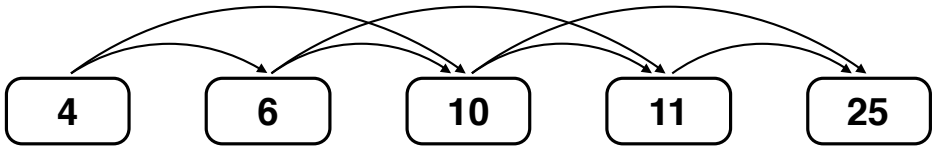
A. Your friend writes three unit tests that each execute a pair of operations concurrently on the list shown above.

   - 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.)

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.**

Practice Problem 3: More Pointers, More Problems



Consider the semi-skip list structure pictured above. Each node maintains a pointer to the next node and the next-next node in the list. **The list must be kept in sorted order.** A node struct is given below.

```
struct Node {
   int value;
   Node* next;
   Node* skip;   // note that skip == next->next
};
```

Please describe a thread-safe implementation of **node deletion** from this data structure. You may assume that deletion is the only operation the data structure supports. Please write C-like pseudocode.

To keep things simple, you can ignore edge-cases near the front and end of the list (assume that you're not deleting the first two or last two nodes in the list, and the node to delete is in the list). If you define local variables like `curNode`, `prevNode`, etc. just state your assumptions about them. **However, please clearly state what per-node locks are held at the start of your process. E.g., "I start by holding locks on the first two nodes.".** Full credit will only be given for solutions that maximize concurrency.

```
// delete node containing value
  void delete_node(Node* head, int value) {
```

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];
```

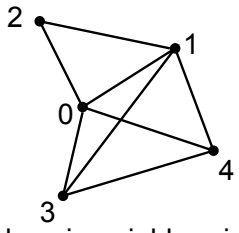
Consider the undirected graph representation shown in the code above.

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.

Graph A



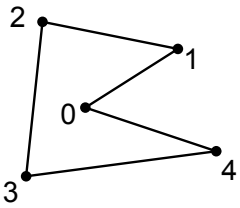
Values in neighbor_ids[]:

0: 3 4 2 1 3: 0 4 1

1: 0 2 3 4 4: 0 3

2: 0 1

Graph B



Values in neighbor_ids[]:

0: 1 4 3: 2 4

1: 0 2 4: 0 3

2: 1 3

Practice Problem 5: Deletion from a Binary Tree

The code below, and continuing on the following two pages implements node deletion from a binary search tree. We have left space in the code for you to insert locks to ensure thread-safe access and high concurrency. You may assume functions `lock(x)` and `unlock(x)` exist (where `x` has type `Lock`). **Your solution NEED ONLY consider the delete operation.**

```
// opaque definition of a lock.  You can call 'lock' and 'unlock' on
// instances of this type
struct Lock;


// definition of a binary search tree node.  You may edit this structure.
struct Node {

   int value;
   Node* left;
   Node* right;

   Lock lock;

};
```

```
// Delete node containing value given by 'value'
// Note: For simplicity, assume that the value to be deleted is not the root node!!!
void delete_node(Node* root, int value) {

   Node** prev_link = NULL;              // pointer to link to current node
   Node* cur = root;                     // current node during traversal

   while (cur) {                         // while node not found
      if (value == cur->value) {         // found node to delete!

         // Case 1: Node is leaf, so just remove it, and update the parent node's pointer to
         // this node to point to NULL
         if (cur->left == NULL && cur->right == NULL) {

            *prev_link = NULL;


            free(cur);
            return;
         } else if ( cur->left == NULL ) {
            // Case 2: Node has one child. Make parent node point to this child

            *prev_link = cur->right;

            free(cur);
            return;
         } else if ( cur->right == NULL) {
            // *** ignore this case, symmetric with previous case ***
         } else {
            // Case 3: Node has two children.  Move the next larger value in the tree into this
            // position.  The subroutine delete_helper returns this value and executes the swap
            // by removing the node containing the next larger value. SEE NEXT PAGE


            /* We need to hold cur->lock the entire time we're in the tree subtree */
            cur->value = delete_helper(cur->right, &cur->right);

            return;
         }
      } else if (value < cur->value) {
         // still searching, traverse left

         prev_link = &cur->left;
         cur = cur->left;
      } else {
         // still searching traverse right

         // *** ignore this case, symmetric with previous case ***
      }
   }
}
```

```
// The helper method removes the smallest node in the tree rooted at node n.
// It returns the value of the smallest node.
int delete_helper(Node* n, Node** prev) {

   Node** prev_link = prev;
   NODE* cur = n;

   // search for the smallest value in the tree by always traversing left
   while (cur->left) {

      prev_link = &cur->left;

      cur = cur->left;

   }

   // this is the smallest value
   int value = cur->value;

   // remove the node with the smallest value
   if (cur->right == NULL) {
      // Case 1: Node is a leaf

      *prev_link = null;

      free(cur);
   } else {
      // Case 2: Node has a right child

      *prev_link = cur->right;

      free(cur);
   }

   return value;
}
```