Stanford CS149: Parallel Computing
Written Assignment 4
Due Nov 10th

Problem 1: A Crazy Linked List

In class we discussed fine-grained locking on a sorted linked list. In this problem we want you to consider a refinement to the sorted linked list example from class: now, a node can only be inserted into the linked list only if the insert preserves the following property: **the value of each node in the linked list is greater than the sum of the values of the previous two nodes**. Pseudocode for insertion into this structure is given on the next page, along with an illustration of an example list that adheres to the stated requirement. In the example, note that $8 > 2 + 5$, $25 > 5 + 8$, etc. Also note that `insert()` will not insert an element if insertion would break the stated invariant.

Please insert locks into the code to ensure thread-safe execution. Full credit will only be given for correct solutions that obtain maximum concurrency (i.e., do not hold locks longer than needed for correctness, and do not hold more locks than are needed for correct execution).

You may assume:

- There is a datatype `Lock` and methods `lock(Lock& l)` and `unlock(Lock& l)`.
- **THE ONLY OPERATION ON THE DATA STRUCTURE IS `insert()`. YOU DO NOT NEED TO REASON ABOUT CONCURRENT DELETES!**.
- The list is at least four elements in size, and that inserts always come after the first two elements in the list. (In other words, don't worry about edge cases of insertions near the beginning and end of the list.)

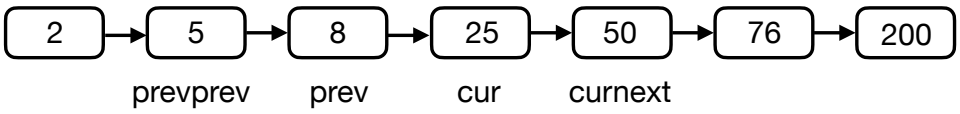
Hint: in your answer consider the case if inserting 14 into the example list on the next page. (yes, its a valid insert). In addition to your code changes, please also briefly summarize why certain per-node locks need to be held (or not held).

Question continues on the next page...

```
void insert(List* list, int value) {
   Node* n = new Node;
   n->value = value;


   Node* prevprev = list->head;

   Node* prev = prevprev->next;

   Node* cur = prev->next;

   Node* curnext = cur->next;


   while (curnext) {


     if (cur->value > value) {

         can_insert = (prevprev->value + prev->value < value) &&
                      (prev->value + value < cur->value) &&
                      (value + cur->value < curnext->value);


         break;
     }

     prevprev = prev;

     prev = cur;


     cur = curnext;


     curnext = curnext->next;


   }


   if (can_insert) {


     prev->next = n;

     n->next = cur;


   } else {

      delete n;

   }

}
```

```
struct Node {
   int value;
   Node* next;

};
```

```
struct List {
  Node* head;

};
```



Problem 2: Cache Coherence and False Sharing

Consider the following program.

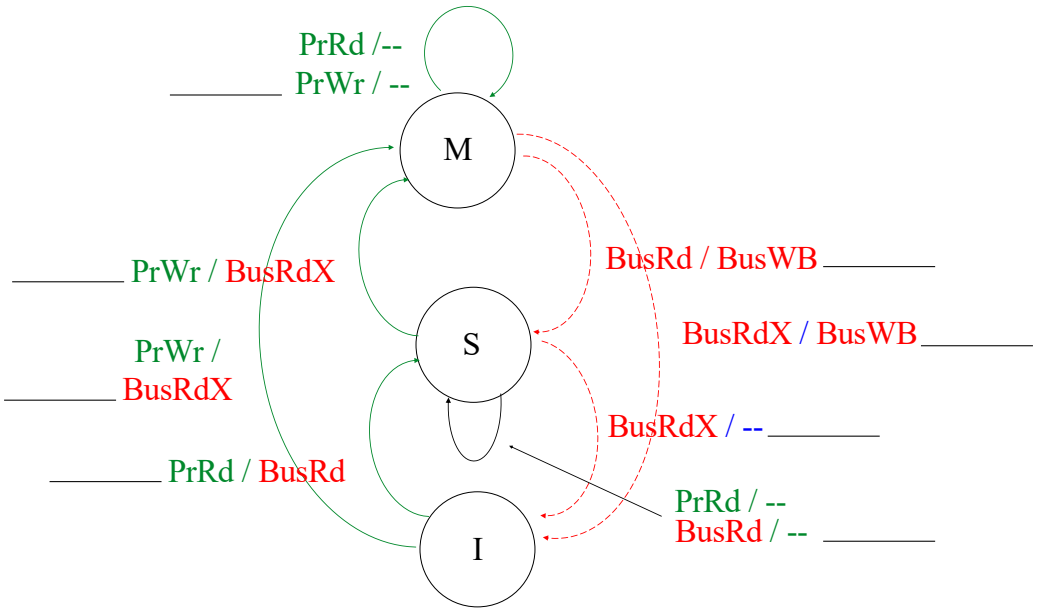
```
void worker(int* counter) {  // each thread runs this function

  barrier();

  // <--- invalidate caches: assume all lines in all caches are
  //      invalidated here after leaving barrier --->

  for (int i=0; i<NUM_ITERS; i++)
    (*counter)++;   // work here (load + incr + store)
}

void test(int num_threads) {
  std::thread threads[MAX_THREADS];
  int         counter[MAX_THREADS];  // Aligned on a cache line boundary

  for (int i=0; i<num_threads; i++)
    threads[i] = std::thread(worker, &counter[i]); // spawn thread
  for (int i=0; i<num_threads; i++)
    threads[i].join(); // wait for thread to terminate
}
```

Consider calling `test()` with `num_threads`=2.

Assuming that code is run on a dual-core processor where:

- Each core has its own private cache
- Caches use the MSI protocol to implement coherence. For reference the MSI diagram is given on the next page.

PROBLEM CONTINUES ON NEXT PAGE...



In your answer to the following questions:

- You should only analyze references to the `counter` array
- Assume all arithmetic is free, you only need to think about memory operations in this problem.
- **Assume that the bus access protocol is such that one core executes the whole C statement involving both the memory read and then the memory write (see the comment "// work here") before allowing bus transactions from the other core.**
- **Assume that between iterations of the `for` loop by one core, the other core is able to execute one iteration of its for loop as well.**
- Each processor action (PrRd, PrWr shown in green) takes 1 cycle. (e.g., A cache hit on a PrWr takes 1 cycle)
- Each bus transaction (BusRd, BusRdX, BusWB shown in red) takes 10 cycles. (Specifically: a PrWr that generates BuxRdX takes a total of 1+10=11 cycles. A PrWr that generates a BusRdX that triggers a remote cache BusWB (write back) requires 1+10+10=21 cycles.)
- No other operations manipulate the state of caches
- Only consider operations that occur after the line <invalidate caches>

PROBLEM CONTINUES ON NEXT PAGE...

A. How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. Please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 4 byte cache line size

B. How many cycles does the memory system take to execute the worker for-loop on one of the threads with the following cache organization. Please list how many times the various coherence protocol transitions occur.
   - 16 byte cache size
   - Fully associative cache (any line can go anywhere, no conflict misses)
   - 8 byte cache line size

Problem 3: Memory Consistency and Locks

Consider the following execution, where T1 and T2 occur in parallel. Assume `x = y = a = b = 0` at the start of the program.

	T1		T2
(1)	x = 1	(3)	y = 2
(2)	a = y + 1	(4)	b = 3 + x
		(5)	a = x + y

Assume the threads run under a *sequentially consistent* memory model. For each of the questions below write TRUE if the given output is possible after all statements have executed, or FALSE otherwise. If you answer TRUE, **give an ordering which produces the output.**

A. x = 1, y = 2, a = 3, b = 4

B. x = 1, y = 2, a = 3, b = 3

C. How many different instruction orderings are possible under a sequentially consistent memory model?

D. You are allowed to modify the program only by inserting one or more calls to lock() and unlock() in each thread. Is there a way to insert the locks to ensure that you get a deterministic output for this program? If so, state where you would insert the lock and unlock instructions (e.g., "lock() before (1) and unlock() after (2)"). If not, what is the minimum number of outputs this program can produce, and where would you insert the lock() and unlock() instructions so that only these orderings were possible?