The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 6

July 22, 2009

Exercise 76. Give an example showing how the universal construction can fail for objects with nondeterministic sequential specifications.

Solution A non-deterministic sequential specification means that a method applied to the object in a particular state may produce different responses and/or leave the object in any of a number of different states. Representing the object as a log of method calls does not work for non-deterministic sequential objects because the object state is not well-defined.

Exercise 77. Propose a way to fix the universal construction to work for objects with nondeterministic sequential specifications.

Solution There are many ways to fix this problem. The simplest is to choose a deterministic implementation that satisfies the non-deterministic specification.

Exercise 78. In both the lock-free and wait-free universal constructions, the sequence number of the sentinel node at the tail of the list is initially set to 1. Which of these algorithms, if any, would cease to work correctly if the sentinel node's sequence number was initially set to 0?

Solution Setting the sequence number of the sentinel node to 0 would cause both constructions to fail because they both test this value to determine whether the node has been added to the head of the log.

Exercise 79. Suppose, instead of a universal construction, you simply want to use consensus to implement a wait-free linearizable register with read() and compareAndSet() methods. Show how you would adapt this algorithm to do so.

Solution The simplest approach would be to remove the invocation object from the Node<> class and replace it with an integer that would hold the contents of the register. The read() function would return the integer in the Node<> returned by Node.max() while the compareAndSet() method would append a Node to the log just as Universal.apply() does.

Exercise 80. In the construction shown here, each thread first looks for another thread to help, and then tries to to append its own node.

Suppose instead, each thread first tries to append its own node, and then tries to help the other thread. Explain whether this alternative approach works. Justify your answer.

```
1    ...
2    static public Node last(Node tail) {
3      Node prev = tail;
4      Node current = prev.next;
5      while (true) {
6        if (current == null || current.seq == 0)
7              return prev;
8              prev = current;
9              current = current.next;
10       }
11   }
12   ...
```

Figure 1: Finding the last node without a head

Solution It should be possible to reverse the order, for a thread to first append its own Node and then help another thread. The key is to be able to guarantee that no thread starves. In the lock-free construction, it is possible for a thread to starve while other threads make progress. If each thread that makes progress by successfully appending its own Node then proceeds to help less fortunate threads, it should still be possible to guarantee progress and thus prove that the algorithm is wait-free.

Exercise 81. In the construction in Fig. **??** we use a "distributed" implementation of a "head" reference (to the node whose decideNext field it will try to modify) to avoid having to create an object that allows repeated consensus. Replace this implementation with one that has no head reference at all, and finds the next "head" by traversing down the log from the start until it reaches a node with a sequence number of 0 or with the highest non-zero sequence.

Solution A head-less implementation would replace Node.max() with a call that traverses the list beginning from the tail (see Fig 1).

Exercise 82. A small addition we made to the lock-free protocol was to have a thread add its newly appended node to the *head* array in Line **??** even though it may have already added it in Line **??**. This is necessary because, unlike in the lock-free protocol, it could be that the thread's node was added by another thread in Line **??**, and that "helping" thread stopped at Line **??** right after updating the node's sequence number but before updating the *head* array.

1. Explain how removing Line **??** would violate Lemma **??**.

2. Would the algorithm still work correctly?

Solution Suppose helping thread B adds A's node to the list, sets its sequence number, and then halts. A notices when the sequence number becomes non-zero and returns, without updating the announce[] array. The next time A executes another call,

$$\mathsf{head}[A].seq < start(A),$$

violating the Lemma.
[[[[Nir?]]]]

Exercise 83. Propose a way to fix the universal construction to work with a bounded amount of memory, that is, a bounded number of consensus objects and a bounded number of read–write registers.

Hint: add a before field to the nodes and build a memory recycling scheme into the code.

Solution [[[[Nir?]]]]

Exercise 84. Implement a consensus object that is accessed more than once by each thread using read() and compareAndSet() methods, creating a "multiple access" consensus object. Do not use the universal construction.

Solution [[[[Nir?]]]]