习题答疑

2015.11.12

第三章第29、30题

29. The question is not well-phrased, however, since it is not clear about which steps appear in the history. The intent is that the history includes steps of x's implementation, not just top-level method calls.

30. YES

第四章第36题

- **False**? The construction doesn't works with regular instead of atomic SRSW registers.
- A counter-example is created by letting the same reader perform two consecutive reads while the writer is performing a write (to the slot of this reader). With regular registers, the first read can return the newly written value while the second read returns the old value of the register.

第四章第36题

- Atomic registers satisfy the following conditions, regular registers only the first two:
  - It is never the case that Ri → Wi
  - It is never the case that for some j: Wi →Wj →Ri
  - If Ri →Rj then i≤j
- The construction which we arrive at by substituting the array of atomic registers a_table with an array of regular registers r_table **yields** an atomic MRSW register.
- Regular and atomic registers differ only in their behavior for overlapping reads and writes. Since the second property concerns non-overlapping writes, and the original construction from Fig. 4.12 is an atomic MRSW register, the altered construction also satisfies the second property.
- It remains to show the third condition, which specifies the behavior of non-overlapping reads. To violate this condition, a read must return an earlier value than a preceding, non-overlapping read. Since every read writes its value into an entire row, this is impossible. The later read will pick up the latest value read by any earlier read by fetching the maximum StampedValue over a column and so this condition is also satisfied.

第四章第42题

- **Solution I**: read() constructs a local copy of b, reading from back to front. It then returns a reconstruction of the integer from a specific subsection of the local copy, depending on whether the first two sections are equal or not.
- If a read does not overlap a write, this register returns the latest value.
- Since this register is write-once, there may only be at most one $i \in [1, 3N-1]$ such that $c[i] = b[i]_{old}$ and $c[i-1] = b[i-1]_{new}$. If $n < 2N$, the old value $c[2N..3N-1]$ is returned. As soon as $n \geq 2N$, only the new value can be returned.

第四章第42题

- **Solution II**: Think of the register as having three copies of the value, which we call left (low-order), middle, and right (high). Think of the writer as writing one bit at a time from left to right, and the reader as reading one bit at a time from right to left. Because the reader and writer move in opposite directions, they overlap on only one of the three copies.
- If the left and middle copies are the same, then the overlap occurred on the right copy, so take the left copy.
- If the right and middle copies are the same, then the overlap occurred on the left copy, so take the right copy.
- if all three copies are different, then the overlap occurred on the middle copy, so take the right copy.

第四章第42题

- **Solution III**: Another idea is that b[N..2N-1] and b[0..N-1] are repeatedly read, until the outcomes coincide. Because the read can only have clashed with the write while reading either b[N..2N-1] or b[0..N-1]. So if the outcomes coincide, we can be certain the outcome represents either the original or the written integer value in the register.

第5章第53题

- We need to show that the Stack class can solve consensus for two threads, but not three.
- Here is a two-thread consensus protocol.
  - Both threads announcing their values in an array.
  - The stack is initialized with two items: first we push the value LOSE, and then the value WIN.
  - In the decide() method, each thread pops an item from the stack.
  - The thread that pops WIN decides its own value, and the thread that pops LOSE decides the other thread's proposed value.
- The proof that consensus is impossible with three threads is a valence argument like the one used for the Queue class. By trying all the combinations of pop() and push(), one can show that the third thread could not tell which method call came first.

第5章第53题

- Suppose towards a contradiction that there is a wait-free consensus protocol for three threads. Given threads A, B and C. Consider a critical state s. Let a move from A lead to decision 0, and a move from B to decision 1.
  - Let A and B perform methods on different stacks. ……
  - Let A and B do pops on the same stack. ……
  - Let A push a onto and B pop from the same stack. …..
  - Let A pop and B pushes b onto the same stack. …….
  - Let A push a and B push b onto the same stack. ……
- All cases contradict the fact that s is critical.

第5章第54题

- Here is a simple consensus protocol. As usual, threads share a public announce[] array. In the propose() method, the each thread with its index A writes its proposed value to announce[A]. In the decide() method, each thread first calls enq(A), and then calls peek(), which returns B. The thread then decides the value in announce[B]. This is correct because if B is the first index to be enqueued, later enq() calls will not change B's position, and B stored its value in announce[B] before calling enq(B).

第5章第68题

- The problem is that the snapshot does not tell you the queue's actual state. The operation of making room for the head and then inserting a value into it is not atomic.
- Start with an empty queue.
- The first thread does an enqueue and increments head to make room for its item, but does not write its value, leaving that spot null.
- A second thread then runs and completely finishes enqueuing its value. It then takes a snapshot and calls dequeue on the snapshot. It goes through the for-loop and finds that the space that the first thread had made is still null, so it thinks that the second enqueue's value is at the head of the queue and decides on this value.
- Then the first thread wakes up, finishes writing its value and calls dequeue on a snapshot to see its value is first and decides on it, a different value. So consensus is not reached.

第5章第69题

```
public T newCompareAndSet(T expect, T update) {
        if (compareAndSet(expect, update))
                return expect;
        else
                return get ();
}
```

第5章第69题

```
public T newCompareAndSet(T expect, T update) {
        if (compareAndSet(expect, update))
                return expect;
        else {
                curr = get();
                while (!compareAndSet(curr, curr))
                        curr = get ();
                return curr;
        }
}
```

第6章第78题

- 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.

第6章第80题

- In this alternative approach, the construction would **no longer** be wait-free.
- Namely, threads might continuously succeed to append their own method call, thereby increasing the next available sequence number by one.
- As a result, the method call of an unfortunate thread could be locked out forever.
  - Namely, it would no longer be guaranteed that a method call is appended successfully when the next available sequence number equals the index of the responsible thread modulo n.

第6章第80题

- 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.

第7章第85题

- Suppose that

1. after releasing the lock,
2. a thread A wants to immediately claim the lock again,
3. before the next thread B detects that the locked field of A's node *v* has become false.
4. A sets the locked field of *v* to true, and applies getAndSet(*v*) to tail.
5. Then A finds B as its predecessor, and forever spins on B's locked field, which remains true.
6. Likewise, B forever spins on A's locked field, which remains true.

- **What if without thread B?**