VU Advanced Multiprocessor Programming
SS 2013
Exercises Batch 2

Jakob Gruber, 0203440

October 30, 2014

Contents

1 Specifications

Select any 10 of $\{21, 22, 23, 24, 27, 32, 51, 52, 53, 54, 58, 62, 65, 68\}$ from *Maurice Herlihy, Nir Shavit: The Art of Multiprocessor Programming. Morgan Kaufmann, 2008. Revised 1st Edition, 2012.*

2 Solutions

2.1 Exercise 21

Explain why quiescent consistency is compositional.

An object is quiescent if it has no pending method calls. A method call is pending if its call event has occurred, but not its response event.

Let's say an object A is composed of two quiescently consistent objects B and C. By definition, method calls to B and C appear to take effect in real-time order when separated by a period of quiescence. Since B and C are only accessed during method calls of A, any period of quiescence for A is also a period of quiescence for B and C. Therefore, the property of quiescent consistency is preserved for object A as well.

2.2 Exercise 32

This exercise examines a queue implementation (Fig. 1) whose `enq()` *method does not have a linearization point.*

The queue stores its items in an items array, which for simplicity we will assume is unbounded. The tail field is an `AtomicInteger`*, initially zero. The* `enq()` *method reserves a slot by incrementing tail, and then stores the item at that location. Note that these two steps are not atomic: there is an interval after tail has been incremented but before the item has been stored in the array.*

The `deq()` *method reads the value of tail, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps null with the current contents, returning the first non-null item it finds. If all slots are null, the procedure is restarted.*

Give an example execution showing that the linearization point for enq() cannot occur at Line 13. Hint: give an execution where two `enq()` *calls are not linearized in the order they execute Line 13.*

A: 13, B: 13, B: 14, C: `deq() == B`.

Give another example execution showing that the linearization point for `enq()` *cannot occur at Line 14.*

A: 13, B: 13, B: 14, A: 14, C: `deq() == A`.

Since these are the only two memory accesses in `enq()`*, we must conclude that enq() has no single linearization point. Does this mean* `enq()` *is not linearizable?*

No. The linearization point occurs as soon as `getAndSet()` in line 20 returns a non-`null` value.

```
1  public class HWQueue<T> {
2      AtomicReference<T>[] items;
3      AtomicInteger tail;
4      static final int CAPACITY = 1024;
5      public HWQueue() {
6          items =(AtomicReference<T>[])Array.newInstance
               (AtomicReference.class, CAPACITY);
7          for (int i = 0; i < items.length; i++) {
8              items[i] = new AtomicReference<T>(null);
9          }
10         tail = new AtomicInteger(0);
11     }
12     public void enq(T x) {
13         int i = tail.getAndIncrement();
14         items[i].set(x);
15     }
16     public T deq() {
17         while (true) {
18             int range = tail.get();
19             for (int i = 0; i < range; i++) {
20                 T value = items[i].getAndSet(null);
21                 if (value != null) {
22                     return value;
23                 }
24             }
25         }
26     }
27 }
```

Figure 1: The Herlihy/Wing queue used in exercise 32.

2.3 Exercise 51

Show that if binary consensus using atomic registers is impossible for n threads, then so is consensus over k values, where $k > 2$.

Suppose consensus over $k > 2$ values is possible, while binary consensus is impossible. We could now construct a binary consensus protocol simply by forming the k-consensus and mapping $[0, \lceil \frac{k}{2} \rceil [$ to 0 and $[\lceil \frac{k}{2} \rceil, k]$ to 1. This contradicts the fact that binary consensus is impossible.

Therefore, k consensus is not possible if binary consensus is impossible.

2.4 Exercise 52

Show that with sufficiently many n-thread binary consensus objects and atomic registers one can implement n-thread consensus over n values.

The construction is equivalent to the `StickyBit` consensus protocol from exercise 58. `decide()` first writes its value into an array of atomic registers: `a_values[threadID] = value`. It then marks itself as participating in the decision: `a_participating[threadID] = 1`. Then, for each bit of the value, it calls `decide()` on the corresponding binary consensus object. If the decision matches the bit, we continue with the next bit. Otherwise, we search all other participating threads for a value with matching bits up to the current one and try to assist that thread in writing its value.

Once this is done, we reconstruct the value from the decided bits and return it.

2.5 Exercise 53

The `Stack` class provides two methods: `push(x)` pushes a value onto the top of the stack, and `pop()` removes and returns the most recently pushed value. Prove that the `Stack` class has consensus number exactly two.

Consensus object: provides a method `T decide(T value)` which is called by each thread at most once. It is *consistent* (all threads decide the same value) and *valid* (the common decision value is some thread's input).

Consensus protocol: a solution to the consensus problem that is wait-free (and therefore also lock-free).

Consensus number: A class C solves n-thread consensus if there exists a consensus protocol using any number of objects of class C and any number of atomic registers. The consensus number is the largest n for which class C solves n-thread consensus.

This proof is similar to the FIFO Queue proof in the book, pages 108 to 110. To show that the consensus number of `Stack` is at least two, we construct a consensus protocol as in Figure 2.

This protocol is wait-free since `Stack` is wait-free and `decide()` contains no loops. If each thread returns its own input, both must have popped `WIN`, violating the `Stack` protocol. Likewise, both threads returning the other's value also violates the protocol. Additionally, the protocol must return one of the proposed values because the winning value is written before `WIN` is popped.

```
1  class StackConsensus<T> {
2      Stack s;
3      T[] proposed;
4      StackConsensus() {
5          s.push(LOSE);
6          s.push(WIN);
7      }
8      T decide(T value) {
9          proposed[threadID] = value;
10         if (s.pop() == WIN) {
11             return proposed[threadID];
12         } else {
13             return proposed[1 - threadID];
14         }
15     }
16 }
```

Figure 2: A stack consensus protocol.

We now need to show that `Stack` has a consensus number of exactly two. Assume we have a consensus protocol for threads A, B, and C. According to Lemma 5.1.3, there must be a critical state s. Without loss of generality, we assume that A's next move takes the protocol to a 0-valent state, and B's next move leads to a 1-valent state. We also know that these calls must be non-commutative; this implies that they need to be calls on the same object. Next, we know that these calls cannot be made to registers since registers have a consensus number of 1. Therefore, these calls must be made to the same stack object. We can now distinguish between three cases: either both A and B call `push()`, both call `pop()`, or A calls `push()` while B calls `pop()`.

Suppose both A and B call `pop()`. Let s' be the state if A pops, followed by B; and s'' if the pops occur in the opposite order. Since s' is 0-valent while s'' is 1-valent, and C cannot distinguish between both states, it is impossible for C to decide the correct value in both states.

Suppose both A and B call `push()`. Let s' equal the state after A pushes a, B pushes b, A pops b, and B pops a. Likewise, let s'' equal the state after B pushes b, A pushes a, A pops a, and B pops b. The pops must occur because it is the only way to observe the state of the stack. Again, states s' and s'' are indistinguishable for C, contradicting the fact that s' is 0-valent while s'' is 1-valent.

Suppose A calls `push()` while B calls `pop()`. Let s' be the state after A pushes a, B pops a, and A pops the uppermost value of the stack (if it exists). Let s'' be the state after B pops the uppermost value (if it exists), A pushes a, and A pops a. C cannot distinguish between both states. We do not care what happens if an empty stack is popped, since that does not affect the state visible to C.

```
1 class FIFOConsensus<T> {
2     FIFO f;
3     T[] proposed;
4     T decide(T value) {
5         proposed[threadID] = value;
6         f.enq(threadID);
7         return proposed[f.peek()];
8     }
9 }
```

Figure 3: A consensus protocol using FIFO queues.

2.6 Exercise 54

Suppose we augment the FIFO Queue class with a `peek()` *method that returns but does not remove the first element in the queue. Show that the augmented queue has infinite consensus number.*

We show this by constructing a decision protocol using a FIFO Queue with an infinite consensus number as shown in Figure 3.

This protocol is wait-free since `FIFO` is wait-free, and `decide()` contains no loops. If two threads return different values, `peek()` must have returned different thread ids. Since the first element is never removed, this violates the FIFO Queue protocol. Validity is ensured because each thread writes its value into `proposed` before pushing its thread id.

`FIFOConsensus` is wait-free, consistent, valid, and works for any number of threads, and as such it is a consensus protocol with an infinite consensus number.

2.7 Exercise 58

Objects of the `StickyBit` *class have three possible states $\perp$, 0, 1, initially $\perp$. A call to write(v), where v is 0 or 1, has the following effects: If the object's state is $\perp$, then it becomes v. If the object's state is 0 or 1, then it is unchanged. A call to* `read()` *returns the object's current state. 1. Show that such an object can solve wait-free binary consensus (that is, all inputs are 0 or 1) for any number of threads. 2. Show that an array of $\log_2 m$* `StickyBit` *objects with atomic registers can solve wait-free consensus for any number of threads when there are m possible inputs. (Hint: you need to give each thread one single-writer, multi-reader atomic register.)*

1. Figure 4 shows a construction of a binary consensus protocol with consensus number ∞.

Validity is given because `write()` is called before any value is read (after writing, the state of the sticky bit is equal to the first written value). Consensus is given because a sticky bit changes its value only for the first write (and each following read returns that value). Assuming `StickyBit` itself is wait-free, then so is our consensus protocol.

```
1 class StickyBinaryConsensus<T> {
2     StickyBit s;
3     T decide(T value) {
4         s.write(v);
5         return s.read();
6     }
7 }
```

Figure 4: The binary consensus protocol composed of a StickyBit object.

2. Figure 5 shows a consensus protocol implementation for any number of threads and any number of possible values.

`decide()` first indicates that the current thread has submitted a value by setting `a_reg[threadID]` to 1 and `a_reg[t + threadID]` to the proposed value. It now attempts to set all sticky bits to `v`. If it reaches a point at which the current sticky bit element is already set to another value, it looks for a thread which has proposed a value matching the already set sticky bits, and then tries to help that thread complete the sticky bit assignment. Once all sticky bits have been assigned, their value is returned.

This protocol is wait-free since `StickyBit`, the registers and `booleanArrayToInt()` are wait-free, and because all loops are bounded. It is consistent because it returns a value derived from the sticky bit array after having attempted a write to each array element (and by definition of the sticky bit, its value never changes after being written once).

It remains to show that `StickyConsensus` is valid. Let $value_i[k]$ be the k'th bit of the value proposed by thread i. Suppose there are two threads i, j and two indices $k < l$ such that $s[k] = value_i[k]$ and $s[l] = value_j[l]$ (with $value_i[k] \neq value[j[k]$ and $value_i[l] \neq value_j[l]$). This implies that thread j must have had a `v` with bits $[0..l-1]$ not matching those of s. This however is impossible since during each outer loop iteration, it is checked whether the bits match; if they don't, `v` is changed to match the value of a thread with matching bits. Therefore, all threads must return the same value, and the protocol is valid.

2.8 Exercise 62

Consider the following 2-thread ***`QuasiConsensus`*** *problem:*

Two threads, A and B, are each given a binary input. If both have input ***`v`****, then both must decide* ***`v`****. If they have mixed inputs, then either they must agree, or B may decide* 0 *and A may decide* 1 *(but not vice versa).*

Here are three possible exercises (only one of which works). (1) Give a 2-thread consensus protocol using QuasiConsensus showing it has consensus number 2, or (2) give a critical-state proof that this object's consensus number is 1, or (3) give a read–write implementation of QuasiConsensus, thereby showing it has consensus number 1.

Suppose we are in the critical state. Since we only have the `QuasiConsensus` object, the only possible action for threads A and B is to call `decide()` on it.

```
1  class StickyConsensus<T> {
2      int l = number of bits in value;
3      int t = number of threads;
4      StickyBit[] s = new StickyBit[l];
5      MRSWRegister[] a_reg = new MRSWRegister[2 * t]; /*
           Initially 0. */
6      T decide(T value) {
7          T v = value;
8          a_reg[t + threadID] = value;
9          a_reg[threadID] = 1;
10         for (int i = 0; i < l; i++) {
11             b = bit i of v;
12             s[i].write(b);
13             if (s[i].read() != b) {
14                 for (int j = 0; j < t; j++) {
15                     if (a_reg[j] == 1 && bits [0..i]
                           match in a_reg[t + j] and s) {
16                     v = a_reg[t + j];
17                 }
18             }
19         }
20         return booleanArrayToInt(s);
21     }
22 }
```

Figure 5: The consensus protocol composed of a StickyBit object.

```
1  class TeamConsensus<T> {
2      TreeNode leafs[n];
3      T decide(T value) {
4          T v = value;
5          TreeNode node = leafs[threadID];
6
7          while (node != null) {
8              v = node.decide(v);
9              node = node.parent;
10         }
11
12         return v;
13     }
14 }
```

Figure 6: The consensus protocol composed of a tree of team consensus object.

For each combination of input mappings, we will now construct a situation in which it is impossible to reach consensus.

$A : 0, B : 0$: For identical inputs 0, `QuasiConsensus` returns 0. In this case, thread B cannot distinguish between this state and a state in which $decide_A(1) = 1$.

$A : 1, B : 1$: For identical inputs 1, `QuasiConsensus` returns 1. In this case, thread A cannot distinguish between this state and a state in which $decide_B(0) = 0$.

$A : 1, B : 0$: For mixed inputs, `QuasiConsensus` may either agree, or decide 1 for A and 0 for B. If $decide_B(0) = 0$ and $decide_A(1) = 1$, A does not know if B called $decide_B(1) = 1$.

$A : 0, B : 1$: For mixed inputs, `QuasiConsensus` may either agree, or decide 1 for A and 0 for B. If $decide_A(0) = 1$, A cannot distinguish between states a) in which B receives 1 as the decision (and A should also decide 1), and b) in which B receives 0 as the decision.

QuasiConsensus therefore has a consensus number of 1.

2.9 Exercise 65

A team consensus object provides the same propose() and decide() methods as consensus. A team consensus object solves consensus as long as no more than two distinct values are ever proposed. (If more than two are proposed, the results are undefined.)

Show how to solve n-thread consensus, with up to n distinct input values, from a supply of team consensus objects.

We simply construct a complete binary tree of team consensus objects with the smallest number of leaf nodes $l = 2^k$ such that $l \geq n$. The remaining consensus protocol is shown in figure 6. This protocol is correct since every team consensus object receives at most two distinct input values, namely either the raw inputs (at the leaf level), or the decisions of its two children.

2.10 Exercise 68

Fig. 5.17[1] *shows a FIFO queue implemented with read, write,* `getAndSet()` *(that is, swap) and* `getAndIncrement()` *methods. You may assume this queue is linearizable, and wait-free as long as* `deq()` *is never applied to an empty queue. Consider the following sequence of statements.*

- *Both* `getAndSet()` *and* `getAndIncrement()` *methods have consensus number 2.*
- *We can add a* `peek()` *simply by taking a snapshot of the queue (using the methods studied earlier in the course) and returning the item at the head of the queue.*
- *Using the protocol devised for Exercise 54, we can use the resulting queue to solve n-consensus for any n.*

We have just constructed an n-thread consensus protocol using only objects with consensus number 2. Identify the faulty step in this chain of reasoning, and explain what went wrong.

The second step is incorrect. The mechanism explained in Chapter 4.3 takes a snapshot of a number of atomic readers of disjunct registers. The mechanism required for this exercise would be an atomic snapshot of a single, multi-reader, multi-writer object, which is a different problem altogether.

If in fact we did have a way to create such a copy of the FIFO queue, the reasoning would be sound.

[1] In the book. The implementation is not relevant to this exercise.