Research Group for Parallel Computing
Faculty of Informatics
TU Wien

Advanced Multiprocessor Programming

Summer term 2023
Theory exercise 2 of 2

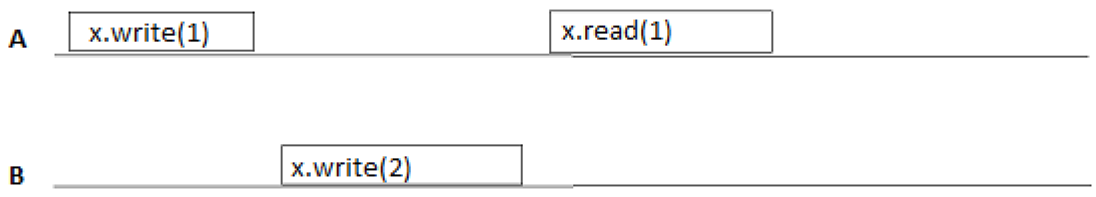
Issue date:	2023-04-23
Due date:	2023-05-08 (23:59)

Total points: 42

1 Consistency Criteria

Ex 1.1 (2 points)

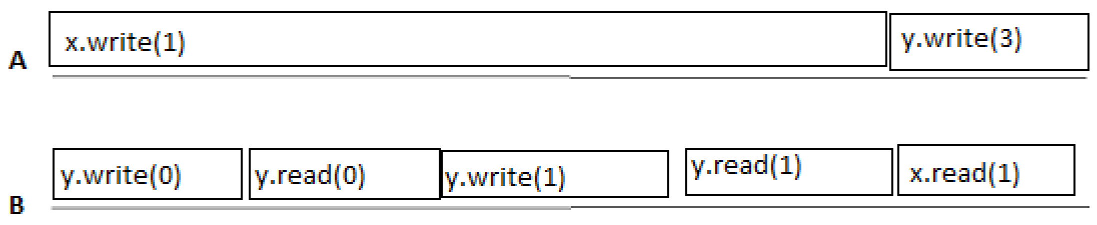
Give an example of a sequentially-consistent execution that is not safe.
Answer:



The above execution is sequentially consistent (as by reordering x.read(1) and x.write(2)) but not safe, since x.read(1) does not return the most recently written value.

Ex 1.2 (2 points)

Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory. Does the converse hold? If the memory is quiescently consistent, are the individual registers quiescently consistent? Outline a proof, or give a counterexample.
Answer:
Consider this memory (the registers are initially 0):



It is quiescently consistent; however, B itself is not quiescently consistent.

Ex 1.3 (4 points)

Suppose x, y, z are registers with initial values 0. Is the history in Figure 1 quiescently consistent, sequentially consistent, linearizable? For each consistency criterion, either provide a consistent execution or a proof sketch.

Does there exist an initial valuation for the registers x, y, z such that your above result w.r.t. linearizability changes? (Meaning if you found the history to be linearizable, can you find initial values for x, y, z s.t. it no longer is linearizable; conversely if you found the history to be not linearizable, can you find initial values s.t. it becomes linearizable?) If so, provide the initial values and either a proof sketch or an example execution.

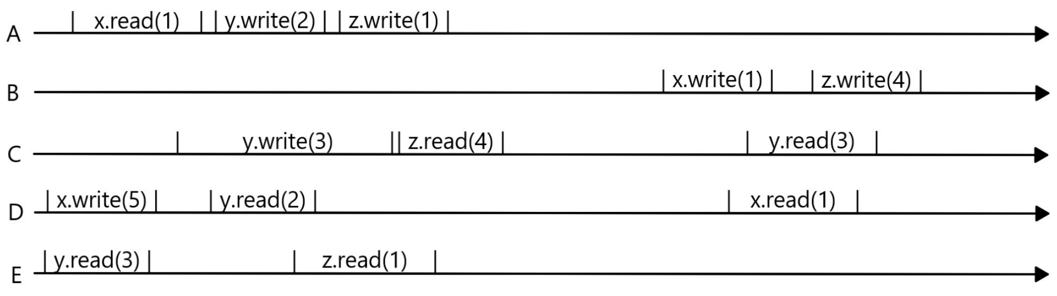


Figure 1: History with five threads and three registers

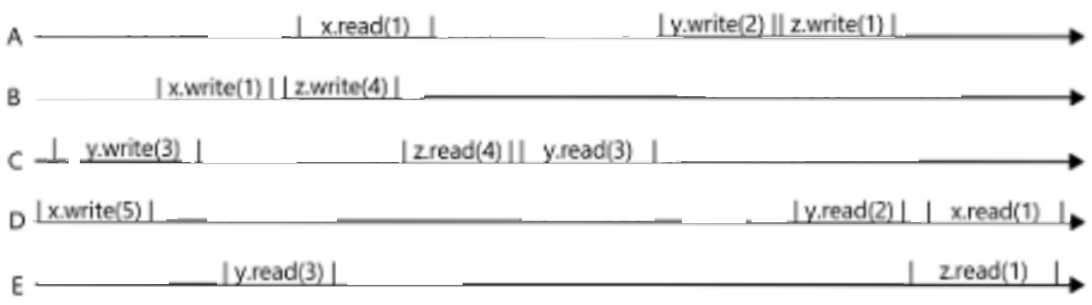
Answer:

Ex 1.3.1 Quiescent consistence

The history is not quiescently consistent, as `x.read(1), A` $\rightarrow$ `x.write(1),B` (inbetween we have quiescence) and all registers are initialized with 0.

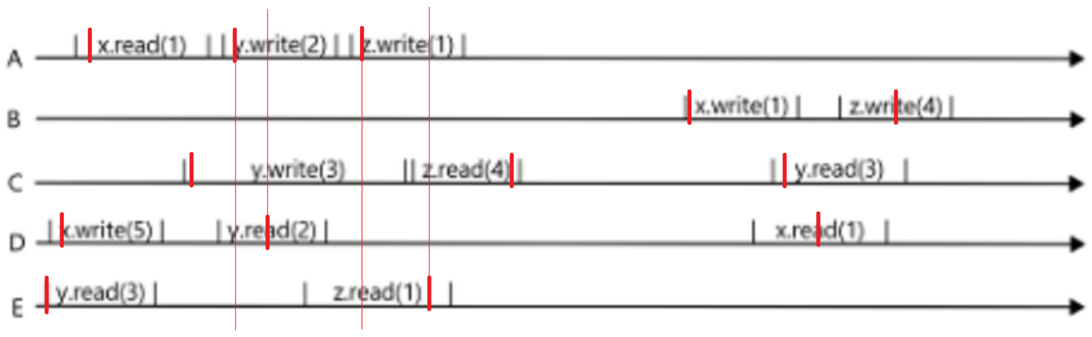
Ex 1.3.2 Sequential consistence

The history is sequentially consistent:



Ex 1.3.3 Linearizability

We cannot linearize the history (as we have already seen above, since linearizability is a stronger condition than quiescent consistence). However, by appropriate choice of linearization points and initial values $x = 1, \quad y = 3, \quad z = 4$, the history becomes linearizable:



Ex 1.4 (4 points)

Suppose x, y, z are stacks and w is a FIFO queue. Initially the stacks and the queue are all empty. Is the history in Figure 2 quiescently consistent, sequentially consistent, linearizable? For each consistency criterion, either provide a consistent execution or a proof sketch.

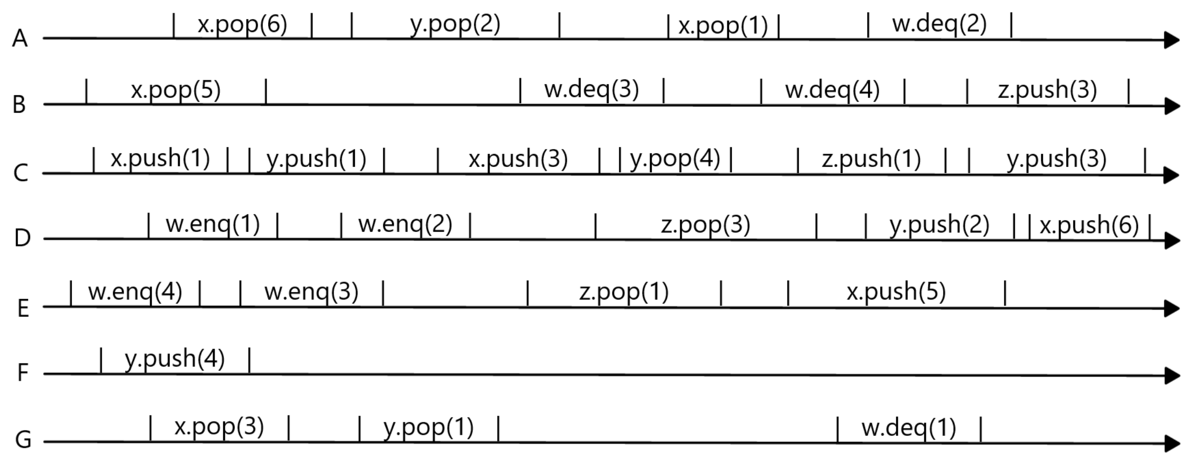


Figure 2: History with seven threads three stacks and one queue

Answer:

Ex 1.4.1 Quiescent consistence

Spanning the history is no window of quiescence. That means that when checking consistency, we can arrange the method calls in any way we like. Every `pop(i)` and `deq(i)` seems to be paired with a corresponding `push(i)` and `enq(i)`. As such the method calls appear to happen in a one at a time, sequential order. With both of these, we have asserted quiescent consistency.

Ex 1.4.2 Sequential consistence

We have `w.enq(4), E` $\rightarrow$ `w.enq(3),E` and `w.deq(3), B` $\rightarrow$ `w.deq(4),B`, and we are supposed to fulfill the First in, first out principle of queues. For the above, there is no order for which we can assert this. Hence, the history is not sequentially consistent.

Ex 1.4.3 Linearizability

We have `x.pop(6), A` $\rightarrow$ `w.push(6),D`, and 6 is not on the stack x. As these calls do not overlap, we cannot find suitable linearization points. Thus, the history is not linearizable.

Ex 1.5 (2 points)

The `AtomicInteger` class (in **java.util.concurrent.atomic** package) is a container for an integer value. One of its methods is

`boolean` `compareAndSet(`**`int`** `expect,` **`int`** `update).`

This method compares the object's current value to `expect`. If the values are equal, then it atomically replaces the object's value with `update` and returns `true`. Otherwise it leaves the object's value unchanged and returns `false`. This class also provides

`int` `get()`

which returns the object's actual value.

Consider the FIFO queue implementation shown in Figure 3. It stores its items in an array `items`, which for simplicity we will assume has unbounded size. It has two `AtomicInteger` fields:

head is the index of the next slot from which to remove an item and tail is the index of the next slot in which to place an item. Give an example execution showing that this implementation is *not* linearizable and provide a short explanation as to why.

```
 1  class IQueue<T> {
 2    AtomicInteger head = new AtomicInteger(0);
 3    AtomicInteger tail = new AtomicInteger(0);
 4    T[] items = (T[]) new Object[Integer.MAX_VALUE];
 5    public void enq(T x) {
 6      int slot;
 7      do {
 8        slot = tail.get();
 9      } while (! tail.compareAndSet(slot, slot+1));
10      items[slot] = x;
11    }
12    public T deq() throws EmptyException {
13      T value;
14      int slot;
15      do {
16        slot = head.get();
17        value = items[slot];
18        if (value == null)
19          throw new EmptyException();
20      } while (! head.compareAndSet(slot, slot+1));
21      return value;
22    }
23  }
```

Figure 3: Non-linearizable queue implementation

Answer:

1. <x.enq(0), A> up to but excluding line 10
2. <x.enq(1), B>
3. <x.deq(), B>
4. After deq() returns, A proceeds with its <x.enq(0), A> until the call returns

Before <x.deq(), B>, we have that head is 0 and items[0] is None and it will throw an Exception.

Ex 1.6 (4 points)

This exercise examines a queue implementation that can be seen in Figure 4, whose enq() method does not have a linearization point.

```java
1   public class HWQueue<T> {
2     AtomicReference<T>[] items;
3     AtomicInteger tail;
4     static final int CAPACITY = 1024;
5
6     public HWQueue() {
7       items =(AtomicReference<T>[])Array.newInstance(AtomicReference.class,
8           CAPACITY);
9       for (int i = 0; i < items.length; i++) {
10        items[i] = new AtomicReference<T>(null);
11      }
12      tail = new AtomicInteger(0);
13    }
14    public void enq(T x) {
15      int i = tail.getAndIncrement();
16      items[i].set(x);
17    }
18    public T deq() {
19      while (true) {
20        int range = tail.get();
21        for (int i = 0; i < range; i++) {
22          T value = items[i].getAndSet(null);
23          if (value != null) {
24            return value;
25          }
26        }
27      }
28    }
29  }
```

Figure 4: Queue implementation

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 15. (*hint: give an execution, where two* `enq()` *calls are not linearized in the order they execute line 15*)

Give another example execution showing that the linearization point for `enq()` cannot occur at line 16.

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?

Answer: Line 15:

1. Line 15: A calls getAndIncrement(), returning 0
2. Line 15: B calls getAndIncrement(), returning 1

3. B: items[1] = b
4. C sees items[0] empty and items[1] written and dequeues b
5. A: items[0] = a
6. C sees item[0] written and dequeues c

Line 16:

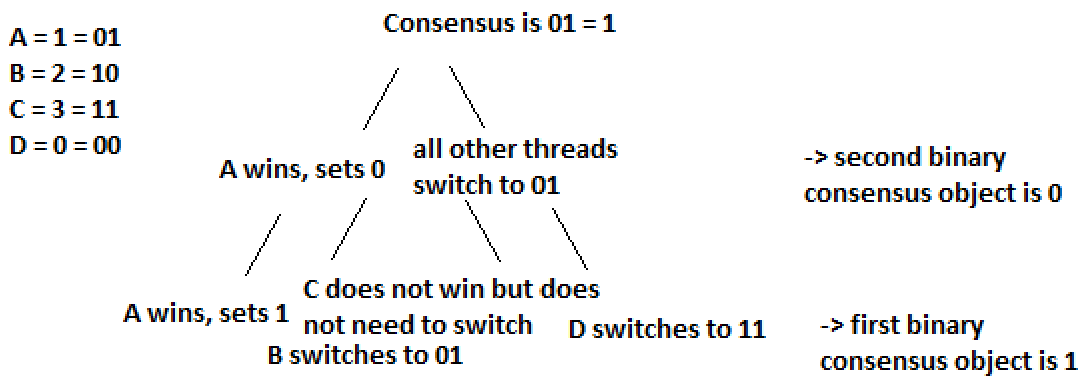
1. A calls getAndIncrement(), returning 0
2. B calls getAndIncrement(), returning 1
3. Line 16, B: items[1] = b
4. Line 16, A: items[0] = a
5. C sees items[0] written, dequeues a, then sees items[1] written and dequeues b

However, just by this we cannot conclude that the method is not linearizeable; What we have found is that there is no single linearization point working for all calls.

2 Consensus

Ex 2.1 (6 points)

Show that with sufficiently many n-thread binary consensus objects and atomic registers one can implement n-thread consensus over n values.
Answer: We have that each thread announces its input in the shared array of atomic register. Also we know that to represent n unique values in binary we need $\log_2 n$ digits, which is going to be our number of binary consensus objects. In order to decide on a number, we start by deciding on the least significant bit, whereas each thread proposes its own least significant bit. One thread is first and sets that binary consensus object; the other threads then switch to a value in the propose array that fits the consensus bits that have already been set. We then move on to the next bit and this process is repeated for all bits of the number. When the process is completed, all threads agree on the same number. The process is examplified for the case $n = 4$ here:


A = 1 = 01
B = 2 = 10
C = 3 = 11
D = 0 = 00
Consensus is 01 = 1
A wins, sets 0
all other threads switch to 01
-> second binary consensus object is 0
A wins, sets 1
C does not win but does not need to switch
B switches to 01
D switches to 11
-> first binary consensus object is 1

Ex 2.2 (6 points)

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.

Answer: We need to show that consensus works for two threads, while also being $consensus < 3$. We do not need to show consensus 1, as if it works for 2, 1 is also given.

If we use a two thread consensus protocol for a stack such as:

```
public class QueueConsensus<T> extends ConsensusProtocol<T> {
  private static final int WIN = 0; // first thread
  private static final int LOSE = 1; // second thread
  Stack stack;
  // initialize stack with two items
  public StackConsensus() {
      stack = new Stack();
      stack.push(LOSE);
      stack.push(WIN);
  }
  // figure out which thread was first
  public T decide(T Value) {
      propose(value);
      int status = stack.pop();
      int i = ThreadID.get();
      if (status == WIN)
          return proposed[i];
      else
          return proposed[1-i];
  }
}
```

Now, in decide, each thread pops an item from the stack, and, depending on if WIN or LOSE was popped, the thread returns its own value (WIN case) or the other threads value (LOSE case), which satisfies consensus 2.

Now, consider a case with three threads. If in our execution two threads decide on different numbers, we have three cases:

1. Both threads push
2. Both threads pop
3. One thread pushes, one pops

In all of these cases however, the third thread cannot tell which thread went first, and consensus number 3 does not hold.

Ex 2.3 (2 points)

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.

Answer: Let us take this consensus protocol:

```
public class QueueConsensus<T> extends ConsensusProtocol<T> {
  protected T[] proposed  (T[]) new Object[N];
  Queue queue;
  // initialize queue with two items
  public QueueConsensus() {
      queue = new Queue();
  }
  public T decide(T Value) {
     propose(value);
      int i = ThreadID.get();
      queue.enq(i);
      return proposed[peek()];
  }
}
```

As we can see, if a thread A is the first thread, its value will be stored in proposed[A] and its index in the queue, whereas later calls will not change that position, and each thread will peek A.

Ex 2.4 (2 points)

Consider three threads, A, B, C, each of which has a MRSW register, X_A, X_B, X_C, that it alone can write and the others can read.

In addition each pair shares a `RMWRegister` register that provides only a `compareAndSet()` method: A and B share R_{AB}, B and C share R_{BC}, A and C share R_{AC}. Only the threads that share a register can call that register's `compareAndSet()` method – which is the only way to interact with said register.

Your mission: either give a consensus protocol and prove that it works or sketch an impossibility proof.

Answer: We know that any wait-free consensus protocol has a critical state. In our case, we have three threads, where each one can access two registers. This leads to a problem:

Imagine A announces its value in R_{AC}. B now does not have any access to this register, and announces its value to either R_{AB} or R_{BC}. At this point, C cannot decide which value should be picked. Hence it should be impossible to give a consensus protocol.

Ex 2.5 (6 points)

Consider the situation described in Ex 2.4 except that A, B, C can apply a `doubleCompareAndSet()` operation to both of their shared registers at once. Regarding the semantics and signature of this new operation consider two variations [1]:

a) Suppose the signature is

   `bool` `doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2,` **`int`** `expect,` **`int`** `update)`.

   For a call of `doubleCompareAndSet(X, Y, exp, up)`, atomically the following is done: it

[1] If any of the `doubleCompareAndSet()` variations is called by a thread for at least one register, which it does not have access to, we assume the method throws an `IllegalAccess` exception with no further side effects.

checks if the value in `X` and `Y` is equal to `exp`. If both are equal to `exp`, `X` and `Y` are updated with `up` and `true` is returned. Otherwise no update is conducted and it returns `false`. The equivalent description in code can be seen in Listing 1.

b) Suppose the signature is

`TwoBool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)`,
where `TwoBool` is a simple class consisting of two (publicly accessible) **`bool`** values.

`class TwoBool { public bool val1; public bool val2; }`
Consider the semantics given by Listing 2.

Listing 1: doubleCompareAndSet() - variation 1

```
bool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)
{
    bool retVal = false;
    atomic
    {
        if(shReg1 == expect && shReg2 == expect)
        {
            shReg1 = update;
            shReg2 = update;
            retVal = true;
        }
    }
    return retVal;
}
```

Listing 2: doubleCompareAndSet() - variation 2

```
TwoBool doubleCompareAndSet(RMWRegister shReg1, RMWRegister shReg2, int expect, int update)
{
    TwoBool retVal;
    atomic
    {
        retVal.val1 = shReg1.compareAndSet(expect, update); // the 'usual' CAS operation on a
        retVal.val2 = shReg2.compareAndSet(expect, update); // single register
    }
    return retVal;
}
```

For both semantics of the `doubleCompareAndSet()` operation either provide a consensus protocol and prove that it works or sketch an impossibility proof. (Note that threads can still use their shared register's normal `compareAndSet()` method!)

Answer: For both semantics, we can find a consensus protocol in similar manners:

Ex 2.5.1 a)

```
class ThreeConsensus extends ConsensusProtocol {
    private final int FIRST = -1;
    private AtomicInteger r_AB = new AtomicInteger(FIRST);
    private AtomicInteger r_BC = new AtomicInteger(FIRST);
    private AtomicInteger r_AC = new AtomicInteger(FIRST);
    private List<AtomicInteger> RMWRegisters = {r_AB, r_AC, r_BC};

    public Tuple<Integer, Integer> getIndices(integer ID) {
        if (ID == 0) return Tuple(0, 2);
        else if (ID == 1) return Tuple(0, 1);
        else if (ID == 2) return Tuple(1, 2);
    }
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get()
        Tuple indices = getIndices(i);
        if (doubleCompareAndSet(RMWRegisters.get(indices.getValue0()), RMWRegisters.get(indices
        Integer val1 = RMWRegister.get(indices.getValue0());
        Integer val2 = RMWRegister.get(indices.getValue1());
        if (val1 != FIRST) return val1;
        return val2;
    }
}
```

Ex 2.5.2 b)

```
class ThreeConsensus extends ConsensusProtocol {
    private final int FIRST = -1;
    private AtomicInteger r_AB = new AtomicInteger(FIRST);
    private AtomicInteger r_BC = new AtomicInteger(FIRST);
    private AtomicInteger r_AC = new AtomicInteger(FIRST);
    private List<AtomicInteger> RMWRegisters = {r_AB, r_AC, r_BC};

    public Tuple<Integer, Integer> getIndices(integer ID) {
        if (ID == 0) return Tuple(0, 2);
        else if (ID == 1) return Tuple(0, 1);
        else if (ID == 2) return Tuple(1, 2);
    }
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get()
        Tuple indices = getIndices(i);
        TwoBool retVal = doubleCompareAndSet(RMWRegisters.get(indices.getValue0()), RMWRegisters
        if (retVal.val1 && retVal.val2) {
            return proposed[i];
        }
        else {
            Integer val1 = RMWRegister.get(indices.getValue0()).get();
            Integer val2 = RMWRegister.get(indices.getValue1()).get();
            if (retVal.val1) return val2;
            return val1;
        }
    }
}
```

Ex 2.6 (2 points)

The `SetAgree` class, like the `Consensus` class provides a `decide()` method whose call returns a value that was the input of some thread's `decide()` call. However unlike the `Consensus` class, the values returned by `decide()` calls are not required to agree. Instead these calls may return no more than k distinct values. (When k is 1, `SetAgree` is the same as consensus.)

What is the consensus number of the `SetAgree` class when $k > 1$? Sketch a proof!

Answer:

By definition of the consensus problem, we are searching for the number of threads for which the consensus protocol can solve the consensus problem (in which n threads need to agree on a common value among local values provided by the threads, and no thread should possibly block another thread).

For the `SetAgree` class and $k > 1$, that means that more than one unique value can be returned, which does not solve the consensus problem. So the consensus number for $k > 1$ is still 1, for the case in which we only have one thread.