Exercise 47. Prove Lemma **??**.

Solution Let $A_0, \ldots, A_{n-1}$ be the n threads. Consider the sequence of initial states $s_0, \ldots, s_n$ where $A_0, \ldots, A_{i-1}$ have inputs 1 and $A_i, \ldots, A_{n-1}$ have inputs 0. (That is, s_0 has all inputs 0 and s_n all inputs 1.)

We argue by induction on i that if all s_i are univalent, then are 0-valent. Since s_n (all 1 inputs) is clearly 1-valent, we have a contradiction.

For the base case, s_0 (all 0 inputs) is clearly 0-valent. By the induction hypothesis, assume s_{i-1} is 0-valent. Any execution starting from s_{i-1} in which A_i halts before taking any steps must decide 0 (because s_{i-1} is 0-valent). But an execution starting from s_i in which A_i halts before taking any steps is indistinguishable from the same execution starting from s_{i-1}, so that execution must decide 0, and s_{i-1} is 0-valent.

Exercise 48. Prove that every n-thread consensus protocol has a bivalent initial state.

Solution This question is the same as the previous one.

Exercise 49. Prove that in a critical state, one successor state must be 0-valent, and the other 1-valent.

Solution If all successor states were 0-valent, the critical state would not be bivalent.

Exercise 50. Show that if binary consensus using atomic registers is impossible for two threads, then it is also impossible for n threads, where $n > 2$. (Hint: argue by *reduction*: if we had a protocol to solve binary consensus for n threads, then we can transform it into a two-thread protocol.)

Solution Any wait-free n-thread consensus protocol must still work if only two threads take steps, so the two thread can simply run the n-thread protocol.

Exercise 51. Show that if binary consensus using atomic registers is impossible for n threads, then so is consensus over k values, where $k > 2$.

Solution Proof by contradiction. If we had a k-value, n-thread consensus protocol, we could solve binary 2-thread consensus simply by having only two threads participate, and restricting them to binary values.

Exercise 52. Show that with sufficiently many n-thread binary consensus objects and atomic registers one can implement n-thread consensus over n values.

Solution We agree on the value one bit at a time. The threads share an n-element array of atomic registers, and an array of $\log_2 n$ consensus objects.

Each thread announces its input in a shared array of atomic registers. At any time each thread has a *preference*, the value it is trying to convince the others to decide. Initially, each thread's preferences is its input.

At round i, each thread uses the i^{th} bit of its preference as input to the i-th binary consensus object. If it wins, it continues to the $i + 1$ consensus with the same preference. If it loses, it scans the announcement array for an input that agrees with all the binary values decided in prior consensus rounds, and uses that value for its preference.

Compare to Exercise 58 below.

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.

Solution 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.

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.

Solution Here is a simple consensus protocol. As usual, threads share a public announce[] array. In the propose() method, the each thread with 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).

Exercise 55. Consider three threads, A, B, and C, each of which has a MRSW register, X_A, X_B, and 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}, and A

and C share R_{AC}. Only the threads that share a register can call that register's compareAndSet() method or read its value.

Your mission: either give a consensus protocol and explain why it works, or sketch an impossibility proof.

Solution It's impossible. We know from Lemma **??** that every wait-free consensus protocol has a critical state in which any move by any thread will drive the protocol into a univalent state. Since there are three threads, but only two can access any single register, there must be threads A and B that are about to access different objects. Clearly, these pending method calls commute, leading to a contradiction.

Exercise 56. Consider the situation described in Exercise 5.55, except that A, B, and C can apply a *double* compareAndSet() to both registers at once.

Solution Here is a protocol. As usual, each thread announces its input in a public array. The shared registers are initialized to -1. Each thread tries to set both its registers to its own ID. Only one will succeed. If a thread succeeds, it decides its own value. Otherwise, exactly one of its shared registers holds a value distinct from -1. Decide that's thread's announced input.

Exercise 57. In the consensus protocol shown in **??**, what would happen if we announced the thread's value after dequeuing from the queue?

Solution Here is a protocol. As usual, each thread announces its input in a public array. The shared registers are initialized to -1. Each thread tries to set both its registers to its own ID. Only one will succeed. If a thread succeeds, it decides its own value. Otherwise, exactly one of its shared registers holds a value distinct from -1. Decide that's thread's announced input.

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.)

Solution

1. For binary consensus, each thread simply writes its input to the bit, reads the bit, and decides the value that it reads.

2. For m-value consensus, each thread has an atomic register, and the threads share an array of $\log_2 m$ StickyBit objects. Each thread first writes its input to its public register, and then tries to set each bit in the array to the corresponding bit in its input, starting with bit 0. It continues as long as it wins (that is, the value in the array matches its input). Each time it loses, the thread changes its own preference to another thread's preference that matches the sticky bits as they have been set so far.

Exercise 59. The SetAgree class, like the Consensus class, provides propose() and decide() methods, where each decide() call returns a value that was the argument to some thread's propose() call. 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 $k > 1$?

Solution The SetAgree class, like the Consensus class, By the standard bivalence argument used in class, there must be a critical state where two threads are about to make method calls. Among the threads there must be A and B such that if A goes first, they decide 0, and if B goes first, they decide 1. Suppose A's input is a, and B's input is b, and suppose their decide() calls each returns their respective inputs. No thread can tell which went first, yet they must decide different values depending on the order, a contradiction.

Exercise 60. The two-thread *approximate agreement* class for a given ϵ is defined as follows. Given two threads A and B, each can call decide(x_a) and decide(x_b) methods, where x_a and x_b are real numbers. These methods return values y_a and y_b respectively, such that y_a and y_b both lie in the closed interval $[\min(x_a, x_b), \max(x_a, x_b)]$, and $|y_a - y_b| \leq \epsilon$ for $\epsilon > 0$. Note that this object is nondeterministic.

What is the consensus number of the *approximate agreement* object?

Solution The consensus number of the approximate agreement object is *one*. Here is how to implement approximate agreement using two atomic registers (see Figure **??**). For simplicity, assume that the inputs to approximate agreement are non-negative.

First, here is a simple two-thread algorithm. There are two threads, A and B. Each thread has a register which it writes and the other reads. Call these registers r_a and r_b, and let them both be initialized to $\perp$.

Because exact agreement is impossible (why?), we must converge from inputs x_a and x_b to values that are only ϵ apart. Each thread repeatedly writes its value to its register, and reads the other's register. If a thread reads $\perp$ from the other's register, it decides its own value. Suppose A, whose register holds v_a, reads v_b ($\neq\perp$) from r_b. If $|v_a - v_b| \leq \epsilon$, A decides v_a. If not, A sets r_a to $\frac{v_a+v_b}{2}$ and repeats.

Note that threads may not converge directly. For example, suppose A writes v_a, reads v_b, but is delayed just before writing $\frac{v_a+v_b}{2}$. B repeatedly reads and writes, cutting the interval in half till its value is very close to v_a. Finally, A completes writing $\frac{v_a+v_b}{2}$ to r_a. Here, B has moved too far toward v_a, and the gap between r_a and r_b has suddenly become larer. This scenario can repeat. Nevertheless, if we focus on each interval in which *both* A and B perform a read and a write, it is not hard to see that the *diameter* of the proposed values, $|v_a - v_b|$, is cut by at least one half, and so the values converge in $O(\log(\frac{x_a-x_b}{\epsilon}))$ such intervals. The algorithm is wait-free, since each thread can ultimately reach a decision independently of the other taking steps.

Exercise 61. Consider a distributed system where threads communicate by message-passing. A *type A* broadcast guarantees:

1. every nonfaulty thread eventually gets each message,

2. if P broadcasts M_1 then M_2, then every thread receives M_1 before M_2, but

3. messages broadcast by different threads may be received in different orders at different threads.

A *type B* broadcast guarantees:

1. every nonfaulty thread eventually gets each message,

2. if P broadcasts M_1 and Q broadcasts M_2, then every thread receives M_1 and M_2 in the same order.

For each kind of broadcast,

- give a consensus protocol if possible,

- and otherwise sketch an impossibility proof.

Solution Note that type A broadcast can be implemented out of FIFO queues, if we add one FIFO queue to each site. As there can't be a consensus protocol out of FIFO queues, since the consensus number of a FIFO queue is one, it is impossible to give a consensus protocol using type A broadcast.

A consensus protocol is possible for type B broadcast: every non-faulty site decides on the value of the very first message received. By definition, the very first message received by every *(*non-faulty) site must be the same.

```
A: V_a = input;
    if V_b == \bot return V_a;
    if V_b == 1 return 1;
       else return 0;

B: V_b = input;
    if V_a == \bot return V_b;
    if V_a == 0 return 0;
       else return 1;
```

Figure 1: Implementing QuasiConsensus with read-write registers

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.

Solution Here is a critical-state proof. If we are in a critical state, then by the usual arguments A and B must be about to call methods of a single QuasiConsensus object. The protocol state becomes (say) 0-valent if A goes first, and 1-valent otherwise. Problem is, there is no way to tell which went first. If they have the same inputs, then they get the same outputs. If they have mixed inputs, then it is possible for B to receive 0 and A to receive 1 no matter who goes first.

Here is a register based implementation. One register per thread.

So A and B try to converge to each other's value as in the solution to the approximate agreement problem. Worst that can happen is that A has input 0, B input 1, each sees the other and returns the opposite value, but still OK since A returning 1 and B returning 0 is a valid output for mixed inputs.

Exercise 63. Explain why the critical-state proof of the impossibility of consensus fails if the shared object is, in fact, a Consensus object.

Solution Critical-state proofs work by putting the protocol into a critical state, and then demonstrating that some thread can't tell which one went first. If the shared object is a Consensus object, then all threads can determine who went first (the consensus winner).

Exercise 64. In this chapter we showed that there is a bivalent initial state for 2-thread consensus. Give a proof that there is a bivalent initial state for n thread consensus.

Solution Let $A_0, \ldots, A_{n-1}$ be the n threads. Consider the sequence of initial states $s_0, \ldots, s_n$ where $A_0, \ldots, A_{i-1}$ have inputs 1 and $A_i, \ldots, A_{n-1}$ have inputs 0. (That is, s_0 has all inputs 0 and s_n all inputs 1.)

We argue by induction on i that if all s_i are univalent, then are 0-valent. Since s_n (all 1 inputs) is clearly 1-valent, we have a contradiction.

For the base case, s_0 (all 0 inputs) is clearly 0-valent. By the induction hypothesis, assume s_{i-1} is 0-valent. Any execution starting from s_{i-1} in which A_i halts before taking any steps must decide 0 (because s_{i-1} is 0-valent). But an execution starting from s_i in which A_i halts before taking any steps is indistinguishable from the same execution starting from s_{i-1}, so that execution must decide 0, and s_{i-1} is 0-valent.

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.

Solution Organize the team consensus objects into a binary tree. At the leaves, two threads share an object. They reach consensus between themselves, and both move on to the parent. At level one, four threads share an objects, two groups of two where each group has the same input. Proceeding up the tree, each team consensus object is proposed at most two values, and all threads emerge from the top with a common value.

Exercise 66. A *trinary* register holds values $\perp, 0, 1$, and provides compareAndSet() and get() methods with the usual meaning. Each such register is initially $\perp$. Give a protocol that uses one such register to solve n-thread consensus if the inputs of the threads are *binary*, that is, either 0 or 1.

Can you use multiple such registers (perhaps with atomic read–write registers) to solve n-thread consensus even if the threads' inputs are in the range $0 \ldots 2^K - 1$, for $K > 1$. (You may assume an input fits in an atomic register.) *Important:* remember that a consensus protocol must be wait-free.

- Devise a solution that uses at most $O(n)$ trinary registers.
- Devise a solution that uses $O(K)$ trinary registers.

Feel free to use all the atomic registers you want (they are cheap).

Solution The binary n-consensus protocol is simple: each thread first calls compareAndSet() to try to install its value, and then calls get() to observe the winning value.

Here is how to solve consensus over the range $0, \ldots, K-1$ using n trinary registers. Create an array of n trinary registers, one for each thread. Each thread announces its preference in a public array. Thread i tries to declare itself the winner as follows:

1. Try to set register i to 1.

2. Try to set every other register to 0.

3. Reread the array, and decide the value proposed by thread j, where j is the least register set to 1.

At the start of Step 3, every register has had compareAndSet() applied to it, so every register is either 0 or 1. The thread whose first compareAndSet() is linearized first will succeed in setting its register to 1, so at least one register is 1.

Here is how to solve consensus over the range $0, \ldots, 2^K - 1$ using K trinary registers. The threads share an n-element array announce[] of atomic registers, one slot per thread. as well as an K-element array of trinary registers. Thread A first writes its input to its announce[A]. and then tries to set each bit in the array to the corresponding bit in its input, starting with bit 0. It continues as long as the value in the array matches its input. Each time it loses, the thread changes its own preference to another thread's preference that matches the values of the trinary registers as they have been set so far.

Exercise 67. Earlier we defined lock-freedom. Prove that there is no lock-free implementation of consensus using read–write registers for two or more threads.

Solution The binary n-consensus protocol is simple: each thread first calls compareAndSet() to try to install its value, and then calls get() to observe the winning value.

Here is how to solve consensus over the range $0, \ldots, K-1$ using n trinary registers. Create an array of n trinary registers, one for each thread. Each thread announces its preference in a public array. Thread i tries to declare itself the winner as follows:

1. Try to set register i to 1.

2. Try to set every other register to 0.

3. Reread the array, and decide the value proposed by thread j, where j is the least register set to 1.

```
class Queue {
  AtomicInteger head = new AtomicInteger(0);
  AtomicReference items[] =
    new AtomicReference[Integer.MAX_VALUE];
  void enq(Object x){
    int slot = head.getAndIncrement();
    items[slot] = x;
  }
  Object deq() {
    while (true) {
      int limit = head.get();
      for (int i = 0; i < limit; i++) {
        Object y = items[i].getAndSet(); // swap
        if (y != null)
          return y;
      }
    }
  }
}
```

Figure 2: Queue Implementation

At the start of Step 3, every register has had compareAndSet() applied to it, so every register is either 0 or 1. The thread whose first compareAndSet() is linearized first will succeed in setting its register to 1, so at least one register is 1.

Here is how to solve consensus over the range $0, \ldots, 2^K - 1$ using K trinary registers. The threads share an n-element array announce[] of atomic registers, one slot per thread. as well as an K-element array of trinary registers. Thread A first writes its input to its announce[A]. and then tries to set each bit in the array to the corresponding bit in its input, starting with bit 0. It continues as long as the value in the array matches its input. Each time it loses, the thread changes its own preference to another thread's preference that matches the values of the trinary registers as they have been set so far.

Solution The binary n-consensus protocol is simple: each thread first calls compareAndSet() to try to install its value, and then calls get() to observe the winning value.

Here is how to solve consensus over the range $0, \ldots, K - 1$ using n trinary registers. Create an array of n trinary registers, one for each thread. Each thread announces its preference in a public array. Thread i tries to declare itself the winner as follows:

1. Try to set register i to 1.

2. Try to set every other register to 0.
3. Reread the array, and decide the value proposed by thread j, where j is the least register set to 1.

At the start of Step 3, every register has had compareAndSet() applied to it, so every register is either 0 or 1. The thread whose first compareAndSet() is linearized first will succeed in setting its register to 1, so at least one register is 1.

Here is how to solve consensus over the range $0, \ldots, 2^K - 1$ using K trinary registers. The threads share an n-element array announce[] of atomic registers, one slot per thread. as well as an K-element array of trinary registers. Thread A first writes its input to its announce[A]. and then tries to set each bit in the array to the corresponding bit in its input, starting with bit 0. It continues as long as the value in the array matches its input. Each time it loses, the thread changes its own preference to another thread's preference that matches the values of the trinary registers as they have been set so far.

Exercise 68. Fig. 2 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.

Solution 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.

Exercise 69. Recall that in our definition of compareAndSet() we noted that strictly speaking, compareAndSet() is not a RMW method for $f_{e,u}$, because a RMW method would return the register's prior value instead of a Boolean value. Use an object that supports compareAndSet() and get() to provide a new object with a linearizable NewCompareAndSet() method that returns the register's current value instead of a Boolean.

Solution We assume, without loss of generality, that values written to this object unique. (This property is easy to provide by tagging values with unique version numbers.)

The new method takes expected and update arguments, just like compareAndSet(). It first tries to use compareAndSet() to replace the expected value with the new value. If it succeeds, it simply returns the expected state. If it fails, then we reread the object state, (guaranteed to be different from the expected value), and pretend that was the value we encountered in the original compareAndSet().

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

Exercise 70. Define an *n-bounded compareAndSet()* object as follows. It provides a compareAndSet() method that takes two values, an *expected* value e, and an *update* value u. For the first n times compareAndSet() is called, it behaves like a conventional compareAndSet() register: if the register value is equal to e, it is atomically replaced with u, and otherwise it is unchanged, and returns a Boolean value indicating whether the change occurred. After compareAndSet() has been called n times, however, the object enters a *faulty* state, and all subsequent method calls return $\perp$.

Show that an n-bounded compareAndSet() object has consensus number exactly n.

Solution It is easy to see that n threads can reach consensus using an n-bounded compareAndSet() object using the same protocol they would use for a regular compareAndSet() object.

To see why the consensus number cannot exceed n, suppose we have a protocol for $N > n$ threads, and consider a critical state. Since the $n + 1$ pending method calls cannot commute, they must all be to the same object. Some thread, A would send the protocol to a 0-valent state, while another, B, would send the protocol to a 1-valent state.

Suppose A moves first, followed by the other n threads, with thread Z moving last. Z's method call returns $\perp$ and if it runs solo it must decide 0. Suppose instead B moves first, followed by the other n threads, with thread Z moving

```
public class Assign23<T> {
  AtomicReference<T>[] r = (AtomicReference<T>[])new Object[3];
  public Assign23(T init) {
   for (int i = 0; i < r.length; i++)
     r[i] = new AtomicReference<T>(init);
  }
  public synchronized void assign(T v0, T v1, int i0, int i1) {
    T t0 = r[i0].get();
    T t1 = r[i1].get();
    r[i0].compareAndSet(t0, v0);
    r[i1].compareAndSet(t0, v0);
  }
  public synchronized T read(int i) {
    return r[i].get();
  }
}
```

Figure 3: Implementing Assign23 with compareAndSet() objects

last. Z's method call returns $\perp$ and if it runs solo it must decide 1. But the two protocol states after Z's first move are indistinguishable to Z, a contradiction.

Exercise 71. Provide a wait-free implementation of a two-thread three-location Assign23 multiple assignment object from three compareAndSet() objects (that is, objects supporting the operations compareAndSet() and get()).

Solution See Figure 3. The assign() has two phases. In the first, it reads both registers. In the second, it calls compareAndSet() to change each one from the value read in the first phase to desired value. It ignores the return value. There are three cases to consider.

- If both calls succeed, then the caller has atomically changed the contents of both registers.
- If both calls fail, then both registers have changed since Phase 1. The call acts as if it atomically changed both, but both have since been overwritten by newer values.
- If one call fails and one succeeds, then register has changed since Phase 1, and one has not. The call acts as if it atomically changed both, but one has since been overwritten by a newer value.

Exercise 72. In the proof of Theorem **??**, we claimed that it is enough to show that we can solve 2-consensus given two threads and an $(2,3)$-assignment object. Justify this claim.

Proof. Suppose we have a wait-free (m, n)-assignment object implemented by atomic registers for some $n > m > 1$. If only two threads, A and B use this object, where A assigns to indexes 0 and 1, but not 2, and B assigns to indexes 1 and 2, but not 0, then by restricting attention to array indexes 0, 1, and 2, we have implemented a $(2, 3)$-assignment object. It follows that if a wait-free $(2, 3)$-assignment object is impossible using only atomic registers, then so is a wait-free (m, n)-assignment object. □

Exercise 73. Prove Corollary **??**.

Solution [] Modify the n-thread protocol so that if a thread lost the compareAndSet(), it will use a trivial version of the compareAndSet() method that simply applies the identity function to test each of the thread ids in sequence until one returns true. It returns the value of the corresponding entry in the proposed[] array.

Exercise 74. We can treat the scheduler as an *adversary* who uses the knowledge of our protocols and input values to frustrate our attempts at reaching consensus. One way to outwit an adversary is through randomization. Assume there are two threads that want to reach consensus, each can flip an unbiased coin, and the adversary cannot control future coin flips.

Assume the adversary scheduler can observe the result of each coin flip and each value read or written. It can stop a thread before or after a coin flip or a read or write to a shared register.

A *randomized consensus protocol* terminates with probability one against an adversary scheduler. Fig. 4 shows a plausible-looking randomized consensus protocol. Give an example showing that this protocol is incorrect.

Solution Think of the scheduler as an *adversary* that is trying to frustrate your protocol. Suppose A has input 0 and B has input 1. Have them write to the prefer [] array at the same time. Let each read the other's preference. Now run A until its coin flip instructs it to switch to B's preference. Halt A before it writes its new preference to memory. Now run B until its coin flip instructs it to switch to A's preference. Now let them both write. Repeat.

Exercise 75. One can implement a consensus object using read–write registers by implementing a deadlock- or starvation-free mutual exclusion lock. However, this implementation provides only dependent progress, and the operating system must make sure that threads do not get stuck in the critical section so that the computation as a whole progresses.

- Is the same true for obstruction-freedom, the nonblocking dependent progress condition? Show an obstruction-free implementation of a consensus object using only atomic registers.

```
public class RandomizedConsensus<T> extends ConsensusProtocol<T> {
  Random random = new Random();
  public T decide(T input) {
    int i = ThreadID.get();
    int j = 1 − i;
    proposed[i] = input;
    while (true) {
      if (proposed[j] == null) {
        return proposed[i];
      } else if (proposed[i] == proposed[j]) {
        return proposed[i];
      } else {
        if (random.nextBoolean()) {
          proposed[i] = proposed[j];
        }
      }
    }
  }
}
```

Figure 4: Is this a randomized consensus protocol?

- What is the role of the operating system in the obstruction-free solution to consensus? Explain where the critical-state-based proof of the impossibility of consensus breaks down if we repeatedly allow an oracle to halt threads so as to allow others to make progress.

(Hint, think of how you could restrict the set of allowed executions.)

Solution The (incorrect) randomized consensus protocol shown in Figure 4 is a correct obstruction-free consensus protocol if we replace the call to flip () with *true*. Eventually some thread will run by itself long enough to discover that it disagrees with the other. It then writes out its value, and uninterrupted, sees that the two values in memory agree.