Multiprocessor Synchronization and Concurrent Data Structures

2 authors:

Maurice Herlihy
Brown University
339 PUBLICATIONS **15,669** CITATIONS

SEE PROFILE

Nir Shavit
Massachusetts Institute of Technology
193 PUBLICATIONS **7,666** CITATIONS

SEE PROFILE

Multiprocessor Synchronization and Concurrent Data Structures

Maurice Herlihy Nir Shavit

October 19, 2005

Chapter 7

The Relative Power of Synchronization Methods

Imagine you are in charge of designing a new multiprocessor. What kinds of atomic instructions should you include? The literature includes a bewildering array of different choices: atomic registers, `getAndDecrement()`, `swap()`, `getAndComplement()`, `compareAndSet()`, and many, many others. Supporting them all would be complicated and inefficient, but supporting the wrong ones could make it difficult or even impossible to solve important synchronization problems.

Our goal is to identify a set of synchronization primitives powerful enough to solve synchronization problems likely to arise in practice. (Of course, we might want to support other, non-essential synchronization operations for convenience.) To this end, we need some way to evaluate the *power* of various synchronization primitives: what synchronization problems can they solve, and how efficiently can they solve them?

A concurrent object implementation is *wait-free* if each method call finishes in a finite number of steps, no matter how that method call's steps are interleaved with steps of concurrent method calls. One way to evaluate the power of synchronization instructions is how well they support wait-free implementations of shared objects such as queues, stacks, trees, and so on.

We will see that all synchronization instructions are not created equal. We will show that there is an infinite hierarchy of synchronization instructions such that no instruction at one level can be used for a wait-free im-

[0] *Portions of this work are from the book, Multiprocessor Synchronization and Concurrent Data Structures, by Maurice Herlihy and Nir Shavit, published by Morgan Kaufmann Publishers, Copyright 200x Elsevier Inc. All rights reserved..*

```
1 public abstract class Consensus {
2   protected Object[] proposed = new Object[N];
3   // announce my input value to the world
4   public void propose(Object value) {
5     proposed[ThreadID.get()] = value;
6   }
7   // figure out which thread was first
8   abstract public Object decide();
9 }
```

Figure 7.1: Generic Consensus Protocol

plementation of any object at higher levels. The basic idea is simple: each class has an associated *consensus number*, which is the maximum number of threads for which objects of the class can solve an elementary synchronization problem called *consensus*. We will see that in a system of n or more concurrent threads, it is impossible to construct a wait-free implementation of an object with consensus number n from an object with a lower consensus number.

7.1 Consensus Numbers

Consensus is an innocuous-looking, somewhat abstract problem that will have enormous consequences for everything from algorithm design to hardware architecture. A *Consensus Object* provides two methods. Each thread first *proposes* a value `v`, called that thread's *input*, by calling the `propose(v)` method, and then *decides* a value by calling the `decide()` method. The `decide()` method must be:

- *consistent*: all threads decide the same value,
- *valid*: the common decision value is some thread's input, and
- *wait-free*: each thread decides after a finite number of steps.

Sometimes it is useful to focus on consensus problems where all inputs are either zero or one. We call this specialized problem *binary consensus*.

It is convenient to establish a stylized form for consensus protocols, illustrated in Figure 7.1. Each thread first calls the `propose()` method, which

"announces" the thread's input value by writing it to a shared `proposed` array, and immediately returns. Each thread then calls the `decide()` method, which (in essence) figures out which thread "went first", and each thread then decides the value proposed by that thread.

For historical reasons, we call any class that implements `Consensus` a *consensus protocol*.

We want to understand whether a particular class of objects is powerful enough to solve consensus. How can we make this notion more precise? If we think of such objects as supported by a lower level of the system, perhaps the operating system, or even the hardware, then we care about the properties of the class, not about the number of objects. (If the system can provide one object of this class, it can probably provide more.) Second, it is reasonable to suppose that any modern system can provide a generous amount of read/write memory for bookkeeping. These two observations suggest the following definition.

Definition 7.1.1 A class C *solves* n-thread consensus if there exist a consensus protocol which uses any number of objects of class C and any number of atomic registers.

Definition 7.1.2 The *consensus number* of a class C is the largest n for which that class solves n-thread consensus. If no largest n exists, we say the class's consensus number is *infinite*.

Corollary 7.1.1 *Suppose one can implement an object of class C from one or more objects of class D, together with some number of atomic registers. If class C solves n-consensus, then so does class D.*

7.1.1 States and Valence

A good place to start is to think about the simplest interesting case: binary consensus (that is, inputs 0 or 1) for two threads (call them A and B). Each thread makes two moves and decides on a value. Here, a *move* is a method call to a shared object. A *protocol state* consists of the states of the threads and the shared objects. An *initial state* is a protocol state before any thread has moved, and a *final state* is a protocol state after all threads have finished. The *decision value* of any final state is the value decided by all threads in that state.

A wait-free protocol's set of states forms a tree, where each node represents a possible protocol state, and each edge represents a move by some thread. Figure 7.2 shows the tree for a two-thread protocol in which each

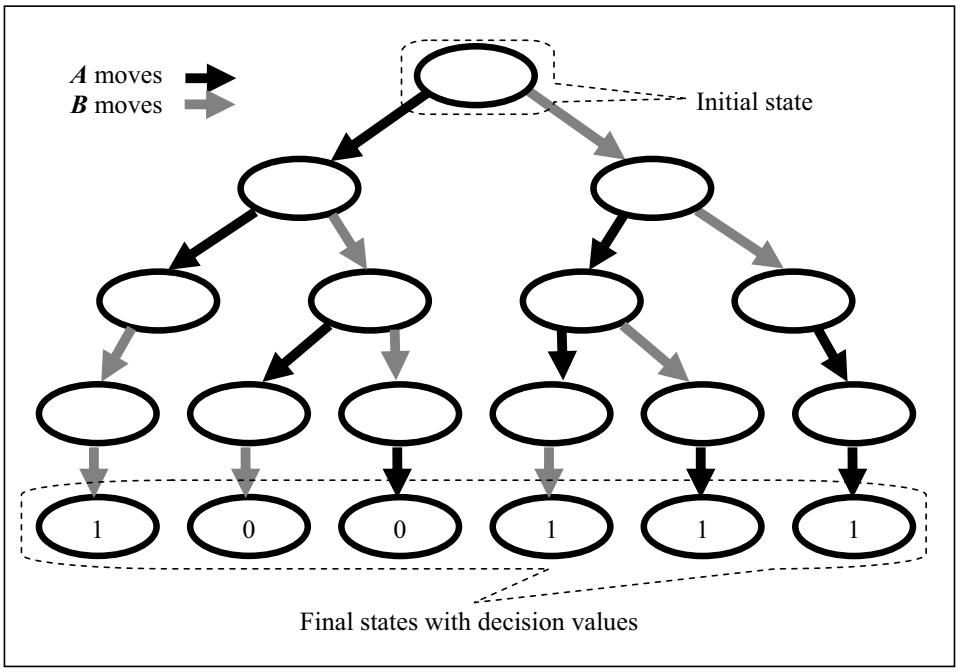

Figure 7.2: Execution Tree for Two Threads

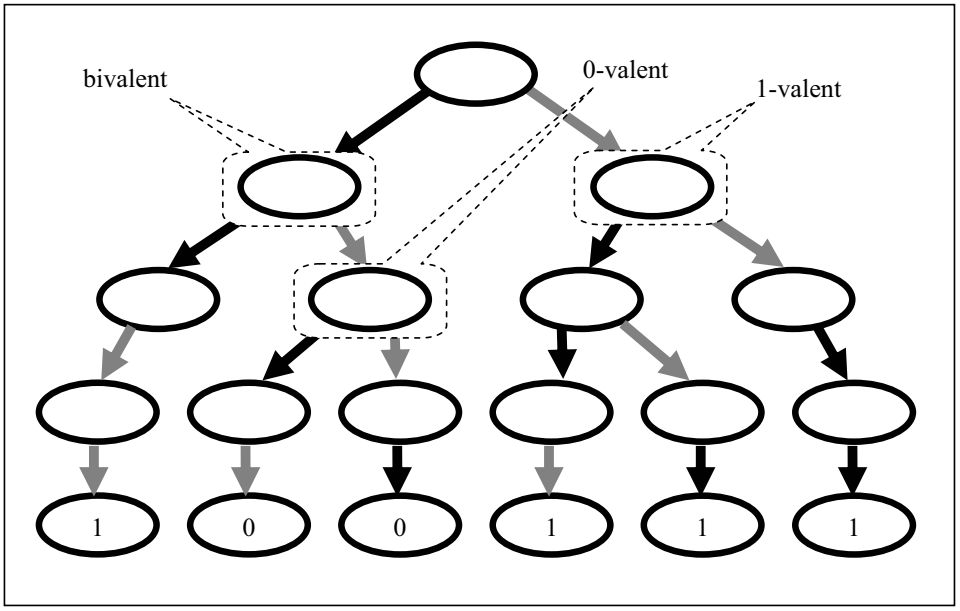

Figure 7.3: Bivalent and Univalent States

thread each moves twice. A's moves are shown in black, and B's in gray. An edge for A from node s to node s' means that if A moves in protocol state s, then the new protocol state is s'. We refer to s' as a *successor state* to s. Because the protocol is wait-free, the tree must be finite. Leaf nodes represent final protocol states, and are labeled with their decision values, either 0 or 1.

A protocol state is *bivalent* if the decision value is not yet fixed: there is some execution starting from that state in which the threads decide 0, and one in which they decide 1. By contrast, the protocol state is *univalent* if the outcome is fixed: every execution starting from that state decides the same value. A protocol state is 1-*valent* if it is univalent, and the decision value will be 1, and similarly for 0-*valent*. As illustrated in Figure 7.3, a bivalent state is a node whose descendents in the tree include both leaves labeled with 0 and leaves labeled with 1, while a univalent state is a node whose descendents include only leaves labeled with a single decision value.

Our next lemma says that initial bivalent states exist. This observation means that the outcome of the protocol cannot be fixed in advance, but must depend on how `read()` and `write()` calls are interleaved.

Lemma 7.1.2 *Every two-thread consensus protocol has a bivalent initial state.*

Proof: Consider the initial state where A has input 0 and B has input 1. If A finishes the protocol before B takes a step, then A must decide 0, because it must decide some thread's input, and 0 is the only input it has seen. Symmetrically, If B finishes the protocol before A takes a step, then B must decide 1, because it must decide some thread's input, and 1 is the only input it has seen. If follows that the initial state where A has input 0 and B has input 1 is bivalent. ■

Lemma 7.1.3 *Every n-thread consensus protocol has a bivalent initial state.*

Proof: Left as an exercise. ■ A protocol state is *critical* if

- It is bivalent, and
- if any thread moves, the protocol state becomes univalent.

We leave as an exercise to the reader to show that one successor state must be 0-valent, and the other 1-valent.

Lemma 7.1.4 *Every consensus protocol has a critical state.*

Proof: Suppose not. By Lemma 7.1.3, the protocol has a bivalent initial state. Start the protocol in this state. As long as there is some thread that can move without making the protocol state univalent, let that thread move. If the protocol runs forever, then it is not wait-free. Otherwise, the protocol eventually enters a state where no such move is possible, which must be a critical state. ■

Everything we have proved so far applies to any consensus protocol, no matter what class (or classes) of shared objects it uses. Now we turn our attention to specific classes of objects.

7.2 Atomic Registers

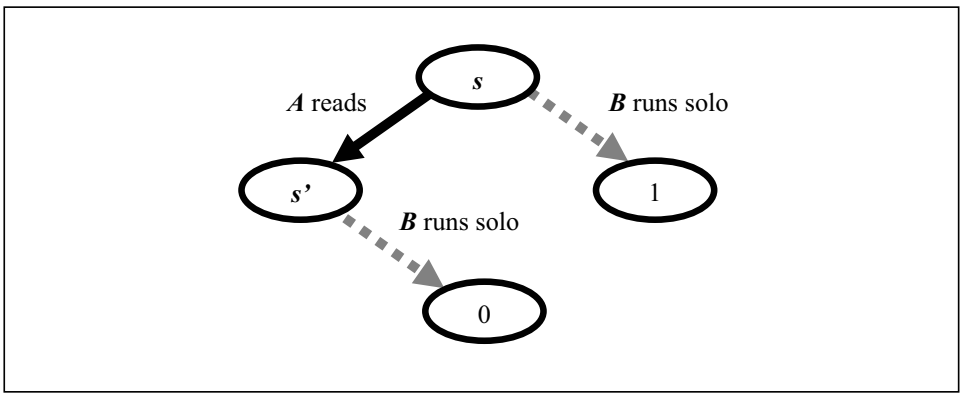

Figure 7.4: Case: A reads first

The obvious place to begin is to ask whether we can solve consensus using atomic registers. Surprisingly, perhaps, the answer is no. We will show that there is no binary consensus protocol for two threads. We leave it as exercises to show that if two threads cannot reach consensus on two values, then n threads cannot reach consensus on k values, where $n > 2$ and $k > 2$.

Often, when we argue about whether or not there exists a protocol that solves a particular problem, we construct a scenario of the form: "if we had such a protocol, it would behave like this under these circumstances ...". One particularly useful scenario is to have one thread, say A, run completely by itself until it finishes the protocol. This particular scenario is common enough that we give it its own name: A runs *solo*.

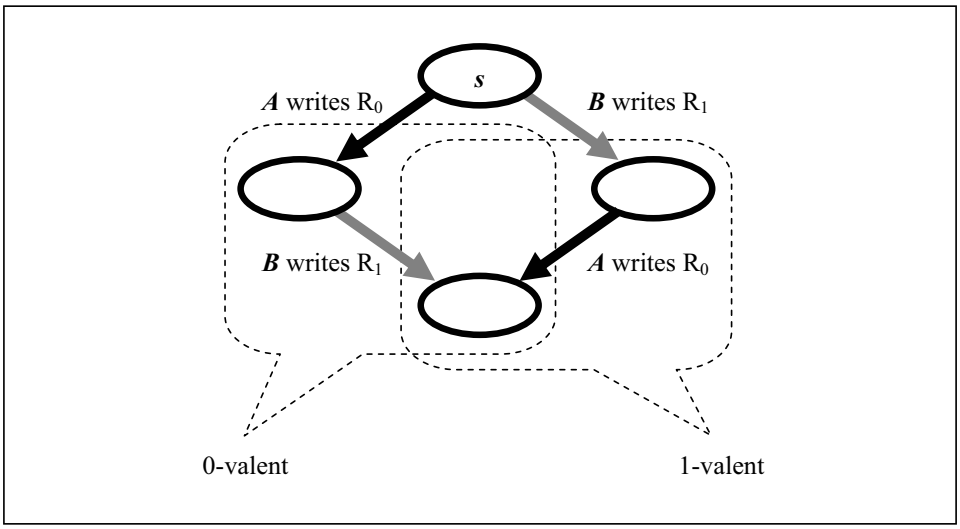

Figure 7.5: Case: A and B write to different registers

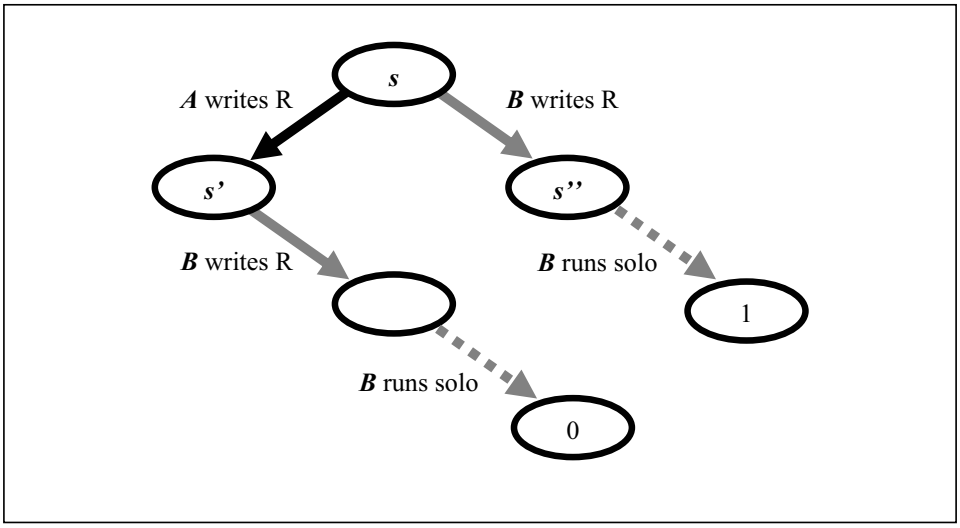

Figure 7.6: Case: A and B write to the same register

Theorem 7.2.1 *Atomic registers have consensus number 1.*

Proof: Suppose there exists a binary consensus protocol for two threads

A and B. We will reason about the properties of such a protocol and derive a contradiction.

By Lemma 7.1.4, we can run the protocol until it reaches a critical state s. Suppose A's next move carries the protocol to an 0-valent state, and B's next move carries the protocol to a 1-valent state. (If not, then switch thread names.) What methods could A and B be about to call? We now consider an exhaustive list of the possibilities:

Suppose A is about to read a shared register (Figure 7.4). In the first scenario, B moves first, driving the protocol to a 1-valent state. and then B then runs solo and eventually decides 1. In the second scenario, A moves first, driving the protocol to a 0-valent state s'. B then runs solo starting in s' and eventually decides 0. The problem is that the states s and s' are indistinguishable to B, which means that B must decide the same value in both scenarios, a contradiction.

Suppose instead that both threads are about to write to different registers (Figure 7.5). A is about to write to R_0 and B to R_1. Consider two scenarios. In the first, A writes to R_0 and then B writes to R_1, so the resulting protocol state is 0-valent because A went first. In the second, B writes to R_1 and then A writes to R_0, so the resulting protocol state is 1-valent because B went first.

The problem is that both scenarios lead to indistinguishable protocol states. Neither A nor B can tell which move was first. The resulting state is therefore both 0-valent and 1-valent, a contradiction.

Finally, suppose both threads write to the same register R (Figure 7.6). If A writes first, the resulting protocol state s' is 0-valent, so if B then runs solo it must decide 0. If, instead, B writes first, the resulting protocol state s'' is 1-valent, so if B then runs solo it must decide 1. The problem is that B cannot tell the difference between s' and s'', so B must decide the same value starting from either one, a contradiction.

■

Corollary 7.2.2 *It is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic registers.*

7.3 Consensus Protocols

We now consider a variety of interesting object classes, asking how well each can solve the consensus problem.

```
1  public class QueueConsensus extends Consensus {
2    private static final int WIN = 0; // first thread
3    private static final int LOSE = 1; // second thread
4    Queue queue;
5    // initialize queue with two elements
6    public QueueConsensus() {
7      queue = new Queue();
8      queue.enq(WIN);
9      queue.enq(LOSE);
10   }
11   // figure out which thread was first
12   public Object decide() {
13     int status = queue.deq();
14     int i = ThreadID.get();
15     if (status == WIN)
16       return proposed[i];
17     else
18       return proposed[1−i];
19   }
20 }
```

Figure 7.7: Two-Thread Consensus using a FIFO Queue

7.4 FIFO Queues

In the previous chapter, we saw a wait-free FIFO Queue implementation using only atomic registers, subject to the limitation that only one thread could enqueue to the queue, and only one thread could dequeue from the queue. It is natural to ask whether one can implement a wait-free FIFO queue that supports multiple enqueuers and dequeuers. For now, let us focus on a more specific problem: can we implement a wait-free two-dequeuer FIFO queue using atomic registers?

Theorem 7.4.1 *The two-dequeuer FIFO queue class has consensus number at least 2.*

Proof: Figure 7.7 shows a two-thread consensus protocol using a single FIFO queue. Here, the queue stores integers. The queue is initialized by enqueuing the value `WIN` followed by the value `LOSE`. As in all the consensus

protocol considered here, each thread first calls `propose(v)`, which stores `v` in a shared array. It then calls `decide()`, which dequeues the next item from the queue. If that item is the value `WIN`, then the calling thread was first, and it decides its own value. If that item is the value `LOSE`, then the other thread was first, so the calling thread returns the other thread's input, as declared in the `proposed` array.

The protocol is wait-free, since it contains no loops. If each thread returns its own input, then they must both have dequeued `WIN`, violating the FIFO queue specification. If each returns the others' input, then they must both have dequeued `LOSE`, also violating the queue specification.

The validity condition follows from the observation that the thread that dequeued `WIN` stored its input in the `proposed` array before any value was dequeued. ■

Trivial variations of this program yield protocols for stacks, priority queues, lists, sets, or any object with methods that return different results if applied in different orders.

Corollary 7.4.2 *It is impossible to construct a wait-free implementation of a queue, stack, priority queue, set, or list from a set of atomic registers.*

Although FIFO queues solve two-thread consensus, they cannot solve three-thread consensus.

Theorem 7.4.3 *FIFO queues have consensus number 2.*

Proof: By contradiction. Assume we have a consensus protocol for threads A, B, and C. By Lemma 7.1.4, the protocol has a critical state. Without loss of generality, we can assume that A's next move takes the protocol to an x-valent state, and B's next move takes the protocol to a y-valent state, where x and y are distinct. The rest is a case analysis.

First, suppose A and B both call `deq()`. Let s be the protocol state if A dequeues and then B dequeues, and let s' be the state if the dequeues occur in the opposite order. Since s is x-valent, then if C runs uninterrupted from s, then it decides x. Since s' is y-valent, then if C runs uninterrupted from s, then it decides y. But s and s' look the same to C, so C must decide the same value in both states, a contradiction.

Second, suppose A calls `enq(x)` and B calls `deq()`. If the queue is non-empty, the contradiction is immediate because the two methods commute: C cannot observe the order in which they occurred. If the queue is empty, then the y-valent state reached if B dequeues and then A enqueues is indistinguishable to C from the x-valent state reached if A alone enqueues.

Finally, suppose A calls `enq(a)` and B calls `enq(b)`. Let s be the state at the end of the following execution:

1. A and B enqueue items a and b in that order.
2. Run A until it dequeues a. (Since the only way to observe the queue's state is via the `deq()` method, A cannot decide before it observes one of a or b.)
3. Run B until it dequeues b.

Let s' be the state after the following alternative execution:

1. B and A enqueue items b and a in that order.
2. Run A until it dequeues b.
3. Run B until it dequeues a.

Clearly, s is x-valent and s' is y-valent. Both of A's executions are identical until A dequeues a or b. Since A is halted before it can modify any other objects, B's executions are also identical until it dequeues a or b. By a now-familiar argument, a contradiction arises because s and s' are indistinguishable to C. ■

Trivial variations of this argument can be applied to show that many similar data types, such as sets, stacks, double-ended queues, and priority queues, all have consensus number exactly two.

7.5 Multiple Assignment

In the *n-assignment* problem, for $n > 1$, we are given n registers, $r_0, \ldots, r_{n-1}$, and n values $v_0, \ldots, v_{n-1}$, and we must assign each value to each register in a way that appears to be atomic to any other register method. This problem is the dual of the *snapshot* problem, where we must read multiple registers atomically.

Figure 7.5 shows a simple double-assignment object, where threads can assign atomically to any two elements of a three-element array.

Theorem 7.5.1 *There is no wait-free implementation of multiple assignment by atomic registers.*

```
1  public class Assign2 {
2    int [] r = new int[3];
3    public Assign2(int v) {        // initialize
4      for (int i = 0; i < 3; i++)
5        r[i] = v;
6    }
7    public synchronized void assign(int i1, int v1, int i2, int v2) {
8      r[i1] = v1;
9      r[i2] = v2;
10   }
11   public int read(int i) {
12     return r[i];
13   }
14 }
```

Figure 7.8: Double assignment

Proof: It is enough to show that given two threads, an array of three registers, and a two-assignment method, then we can solve two-thread consensus.

As usual, the `decide()` method must figure out which thread went first. Suppose thread 0 writes (atomically) to array elements 0 and 1, while thread 1 writes (atomically) to elements 1 and 2.

The threads perform their respective assignments, and then try to determine who went first. From A's point of view, there are three cases:

- If A's assignment was ordered first, and B's assignment has not happened, then positions 0 and 1 have A's value, and position 2 is null. A decides its own input.
- If A's assignment was ordered first, and B's second, then position 0 has A's value, and positions 1 and 2 have B's. A decides its own input.
- If B's assignment was ordered first, and A's second, then positions 0 and 1 have A's value, and 2 has B's. A decides B's input.

A similar analysis holds for B. ■

Theorem 7.5.2 *Atomic m-register assignment has consensus number at least m.*

```
1  public class MultiConsensus extends Consensus {
2    Assign2 assign2 = new Assign2(−1);
3    public Object decide() {
4      int i = ThreadID.get();
5      int j = 1 − i;                    // other index
6      // double assignment
7      assign2.assign(i, i, i+1, i);
8      int other = assign2.read((j+1) % 3);
9      if (other == −1 || other == assign2.read(j))
10       return proposed[i];             // I win
11     else
12       return proposed[j];             // I lose
13   }
14 }
```

Figure 7.9: Two-Thread Consensus from Double Assignment

Proof: We are given m threads $A_0, \ldots, A_{m-1}$. The protocol uses m single-writer registers $r_0, \ldots, r_{m-1}$, where thread A_i writes to register r_i, and $m(m-1)/2$ multi-writer registers r_{ij}, where $i > j$, where A_i and A_j both write to register r_{ij}. All registers are initialized to $\perp$. Each thread atomically assigns its input value to m registers: its single-writer register and its $m-1$ multi-writer registers. The decision value of the protocol is the first value to be assigned.

After assigning to its registers, a thread determines the relative ordering of the assignments for every two threads A_i and A_j as follows.

- Read r_{ij}. If the value is $\perp$, then neither assignment has occurred.
- Otherwise, read r_i and r_j. If r_i's value is $\perp$, then A_j precedes A_i, and similarly for r_j.
- If neither r_i nor r_j is $\perp$, reread r_{ij}. If its value is equal to the value read from r_i, then A_j precedes A_i, else vice-versa.

By repeating this procedure, a thread can determine which value was written by the earliest assignment. ■

In fact, this result can be improved: m-register assignment actually solves $(2m-2)$-consensus, but not $(2m-1)$-consensus.

7.6 Read-Modify-Write Registers

Many, if not all, of the classical synchronization primitives can be expressed as *read-modify-write* (RMW) register, defined as follows. Consider a `RMWRegister` that encapsulates integer values, and let $\mathcal{F}$ be a set of functions from integers to integers.[1]

A method is a *read-modify-write* (RMW) method for the function set $\mathcal{F}$ if it atomically replaces the current register value `v` with `f(v)`, for some `f` $\in \mathcal{F}$, and returns the original value `v`. We(mostly) follow the Java convention that a RMW method that applies the fuction `mumble` is called `getAndMumble()`.

For example, the `java.util.concurrent.atomic` package provides an `AtomicInteger` class with a rich set of RMW methods.

- The `getAndSet(v)` method atomically replaces the register's current value with `v` and returns the prior value. This method (also called `swap()`) is a RMW method for the constant function $c_v(\mathtt{x}) = \mathtt{v}$.
- The `getAndIncrement()` method atomically adds 1 to the register's current value with `v` and returns the prior value. This method (also called `fetch-and-increment` is a RMW method for the function $\oplus(\mathtt{x}) = \mathtt{x} + 1$.
- The `getAndAdd(k)` method atomically adds `k` to the register's current value with `v` and returns the prior value. This method (also called `fetch-and-add`) is a RMW method for the function $\oplus_k(\mathtt{x}) = \mathtt{x+k}$.
- The `compareAndSet()` takes two values, an *expected* value e, and an *update* value u. If the register value is equal to e, it is atomically replaced with u, and otherwise it is unchanged. Either way, the method returns a Boolean value indicating whether the value was changed. Strictly speaking, `compareAndSet()` is not a RMW method for $c_{e,u}$, because a RMW method would return the register's prior value instead of a Boolean value, but this distinction is a technicality.
- The `get()` method returns the register's value. This method is a RMW method for the identify function $\iota(v) = v$.

RMW methods are interesting because they

RMW methods are interesting precisely because they are potential hardware primitives, engraved not in stone, but in silicon. Here, we define RMW

[1]For brevity, we consider only registers that hold integer values, but they could equally well hold references to other objects.

```
1  class RMWConsensus extends Consensus {
2    // initialize  to v such that f(v) != vmostly
3    private RMWRegister r = new RMWRegister(v);
4    public Object decide() {
5      int i = ThreadID.get();      // my index
6      int j = 1 - i;               // other's index
7      if (r.rmw() == v)            // I'm first, I win
8        return proposed[i];
9      else                         // I'm second, I lose
10       return proposed[j];
11   }
12 }
```

Figure 7.10: Two-thread consensus using RMW

registers and their methods in terms of `synchronized` Java methods, but, pragmatically, they correspond (exactly or nearly so) to many real or proposed hardware synchronization primitives.

A RMW method is *non-trivial* if its set of functions includes at least one function that is not the identity function.

Theorem 7.6.1 *Any non-trivial RMW register has consensus number at least 2.*

Proof: Figure 7.10 shows a two-thread consensus protocol. Since there exists a `f` in $\mathcal{F}$ is not the identity, there exists a value `v` such that `f(v)` $\neq$ `v`. As usual, the `propose(v)` method writes the thread's input `v` to the `proposed` array. In the `decide()` method, each thread applies the RMW method to a shared register. If a thread's call returns `v`, it is linearized first, and it decides its own value. Otherwise, it is linearized second, and it decides the other thread's proposed value.

■

Corollary 7.6.2 *It is impossible to construct a wait-free implementation of any nontrivial RMW method from atomic registers for two or more threads.*

7.7 Common2 RMW Registers

We now identify a class of RMW registers, called *Common2*, that correspond to many of the synchronization primitives provided by processors in the late

20^{th} Century. Although *Common2* registers, like all non-trivial RMW registers, are more powerful than atomic registers, we will show that they have consensus number exactly 2, implying that they have limited synchronization power. Fortunately, these synchronization primitives have by-and-large fallen from favor in contemporary processor architectures.

Definition 7.7.1 A set of functions $\mathcal{F}$ belongs to *Common2* if for all values `v` and all f_i and f_j in $\mathcal{F}$, either:

- f_i and f_j *commute*: $f_i(f_j(\texttt{v})) = f_j(f_i(\texttt{v}))$, or
- one function *overwrites* the other: $f_i(f_j(\texttt{v}v)) = f_i(\texttt{v}v)$ or $f_j(f_i(\texttt{v})) = f_j(\texttt{v})$.

Definition 7.7.2 A RMW register belongs to *Common2* if its set of functions $\mathcal{F}$ belongs to *Common2*.

For example, many RMW registers in the literature provide only one non-trivial function. For example, the `getAndSet()` method (also known as `swap()`), uses a constant function, which overwrites any prior value. The `getAndIncrement()` and `getAndAdd()` methods use functions that commute with one another.

Very informally, here is why RMW registers in *Common2* cannot solve 3-thread consensus. The first thread (the *winner*) can always tell it was first, and the second and third threads (the *losers*) can each tell that they are losers, but cannot identify which of the others was the winner. If the functions commute, then a loser thread cannot tell which of the others went first, and because the protocol is wait-free, it cannot wait to find out. If the functions overwrite, then a loser thread cannot tell whether it was preceded by a single winner, or by a winner followed by an overwriting loser. Let us make this argument more precise.

Theorem 7.7.1 *Any RMW register in Common2 has consensus number (exactly) 2.*

Proof: Theorem 7.6.1 states that any such register has consensus number at least 2. We need only to show that any *Common2* register cannot solve consensus for three threads.

Assume instead that a 3-thread protocol exists using only *Common2* registers and read-write registers. Suppose threads A, B, and C reach consensus through *Common2* registers. By Lemma 7.1.4, any such protocol has

a critical state in which the protocol is bivalent, but any method call by any thread will cause the protocol to enter a univalent state.

We now do a case analysis, examining each possible method call. The kind of reasoning used in the proof of theorem 7.2.1 shows that the pending methods cannot be reads or writes, nor can the threads be about to call methods of different objects.

It follows that the threads are about to call RMW methods of a single register `r`. Suppose A is about to call a method for function f_A, sending the protocol to an x-valent state, and B is about to call a method for f_B, sending the protocol to a y-valent state, for $x \neq y$. There are two possible cases:

1. The functions commute: $f_A(f_B(v)) = f_B(f_A(v))$. Let s' be the state that results if A applies f_A and then B applies f_B. Because s' is x-valent, C will decide x if runs alone until it finishes the protocol. Let s'' be the state that results if A and B perform their calls in the reverse order. Because s'' is y-valent, C will decide y if runs alone until it finishes the protocol. The problem is that the two possible register states $f_A(f_B(v))$ and $f_B(f_A(v))$ are the same, so s' and s'' differ only in the internal states of A and B. Since C completes the protocol without communicating with A or B, these two states look identical to C, so it cannot possibly decide different values from the two states.

2. One function overwrites the other: $f_B(f_A(v)) = f_B(v)$. Let s' be the state that results if A applies f_A and then B applies f_B. Because s' is x-valent, C will decide x if runs alone until it finishes the protocol. Let s'' be the state that results if B alone calls f_B. Because s'' is y-valent, C will decide y if runs alone until it finishes the protocol. The problem is that the two possible register states $f_B(f_A(v))$ and $f_B(v)$) are the same, so s' and s'' differ only in the internal states of A and B. Since C completes the protocol without communicating with A or B, these two states look identical to C, so it cannot possibly decide different values from the two states.

■

7.8 The `compareAndSet()` method

We consider another RMW method, `compareAndSet()` (CAS), supported by several contemporary architectures. (For example, it is called `CMPXCHG` on

```
1  class CASConsensus extends Consensus {
2    private final int FIRST = −1;
3    private AtomicInteger r = new AtomicInteger(FIRST);
4    public Object decide() {
5      int i = ThreadID.get();
6      if (r.compareAndSet(FIRST, i)) // I won
7        return this.proposed[i];
8      else                           // I lost
9        return this.proposed[r.get()];
10   }
11 }
```

Figure 7.11: Consensus using CAS

the Intel Pentium$^{\text{tm}}$. This method is also known in the literature *compare-and-swap*. As illustrated in Figure **??**, the `compareAndSet()` method takes two arguments: an *expected* value and an *update* value. If the current register value is equal to the expected value, then it is replaced by the update value, and otherwise the value is left unchanged. The method call returns a Boolean indicating whether the value changed.

Theorem 7.8.1 *The `compareAndSet()` method has infinite consensus number.*

Proof: Figure 7.11 shows a consensus protocol for N threads $A_0, \ldots, A_{N-1}$ using the `AtomicInteger` class's `compareAndSet()` method. The threads share an `AtomicInteger` object, initialized to a constant `FIRST`, distinct from any thread index. Each thread calls `compareAndSet()` with `FIRST` as the expected value, and its own index as the new value. If thread A's call returns *true*, then that method call was first in the linearization order, so A decides its own value. Otherwise, A reads the current `AtomicInteger` value, and takes that thread's input from the `proposed` array.

■

7.9 Exercises

Exercise 7.9.1 Prove Lemma 7.1.3.

Exercise 7.9.2 Prove that every n-thread consensus protocol has a bivalent initial state.

Exercise 7.9.3 Prove that in a critical state, one successor state must be 0-valent, and the other 1-valent.

Exercise 7.9.4 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.)

Exercise 7.9.5 Show that if binary consensus using atomic registers is impossible for n threads, then so is consensus over k values, where $k > 2$.

Exercise 7.9.6 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 This space intentionally left blank. ■

Exercise 7.9.7 Suppose we augment the FIFO `Queue` 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.

Exercise 7.9.8 A simple consensus protocol can be done as follows: In propose each thread writes its proposed value into an array at the index of their thread id. In decide, each thread calls enq with its index, and then performs a peek to see who was the first one to do so. They then decide on this value. This works since only one thread could have performed the first enqueue, and because no dequeue's are done the value in the first position never changes. This is wait-free since enqueuing is wait-free and if a thread goes to perform peek then it must have enqueued its value already so there is at least one value in the queue.

Exercise 7.9.9 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.

Exercise 7.9.10 In the consensus protocol shown in 7.7, what would happen if we announced the thread's value after dequeuing from the queue?

Exercise 7.9.11 Objects of the `StickyBit` class have three possible states $\perp, 0, 1$, initially $\perp$. A call to `write(v)` 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 *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 can solve consensus for any number of threads when there are m possible inputs.

Exercise 7.9.12 Suppose we have three processors, A, B, and C. Each thread has a public register, X_A, X_B, and X_C respectively, that it alone can write and the others can read.

In addition, each pair shares a `compareAndSet()` register: A and B share R_{AB}, B and C share R_{BC}, and A and C share R_{AC}. Only the processors that share a register can apply `compareAndSet()` or `get()` (`read()`) methods to it.

Either give a consensus protocol for these processors, and explain why it works, or sketch an impossibility proof

Solution This space intentionally left blank. ■

Exercise 7.9.13 Consider the situation described in Exercise 7.9.12, except that A, B, and C can apply a *double* `compareAndSet()` to both registers at once.

Solution This space intentionally left blank. ■

Exercise 7.9.14 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 This space intentionally left blank. ■

Exercise 7.9.15 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 interval $(\min(x_a, x_b), \max(x_a, x_b))$, and $|y_a - y_b| \leq \epsilon$ for $\epsilon > 0$.

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

Solution This space intentionally left blank. ■

```
1  public class ApproxConsensus extends Consensus {
2    private final double EPSILON = 0.0001;
3    private double r[2];
4    public Object decide() {
5      int me = ThreadID.get();
6      r[me] = announce[me];
7      if (r[1−me] == null) {
8        return r[me];
9      }
10     do {
11       double min = Math.min(r[me], r[1−me]);
12       double max = Math.max(r[me], r[1−me]);
13       r[me] = (max − min) / 2.0;
14     } while (Math.abs(r[me] − r[1−me]) > EPSILON);
15     return r[me];
16   }
17 }
```

Figure 7.12: wait-free 2-process approximate agreement

Exercise 7.9.16 Consider a distributed system where processors communicate by message-passing A *type A* broadcast guarantees:

1. every non-faulty processor eventually gets each message
2. if P broadcasts M_1 then M_2, then every processor receives M_1 before M_2.
3. but messages broadcast by different processors may be received in different orders at different processors.

A *type B* broadcast guarantees:

1. every non-faulty processor eventually gets each message
2. if P broadcasts M_1 and Q broadcasts M_2, then every processor 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 This space intentionally left blank. ■

Exercise 7.9.17 Prove that the consensus number of a multiple assignment object that allows a thread to write m locations atomically (and read any one location) is at least m.

Exercise 7.9.18 What is the consensus number of the following 2 process *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).

Exercise 7.9.19 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.

Exercise 7.9.20 A *Team consensus* object provides the same `propose()` and `decide()` 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 regular consensus (with up to n distinct input values proposed) from a supply of team consensus objects.

Exercise 7.9.21 Prove that a single *three-valued* `compareAndSet()` object (values $\perp, 0, 1$) can solve n-thread consensus if the inputs of threads are in $0, 1$. Prove that n instances of a *three-valued* COMPARE&SWAP object can solve n-thread consensus even if the inputs of threads are **not** in $0, 1$.

Exercise 7.9.22 In class we defined lock-freedom. Prove that there is no lock-free implementation of consensus using read-write registers for two or more threads.

Figure 7.13: Queue Implementation

Exercise 7.9.23 What is the consensus number of *k-set consensus* object defined as follows: each process P_i has an input X_i from a set $V = \{0, 1, \ldots, v\}$, and is required to return a *valid* output $Y_i \in V$ such that the collective set of decided values $Y = \{Y_i \mid i \leq n\}$ is of size at most k. Note that this object is an example of an object that has a non-deterministic sequential specification. For $k = 1$ it is just consensus. Prove your claim.

Solution This space intentionally left blank. ■

Exercise 7.9.24 What is the consensus number of the following 2 process `mumble` problem? Each of P and Q is 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 Q may decide 0 and P may decide 1 (but not vice-versa).

Exercise 7.9.25 Figure 7.13 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 turn the queue into a `PQueue`, implementing the `peek()` method 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 Exercise7.9.7, we can use the resulting `PQueue` to solve n-consensus for any n.
- We have just constructed an n-thread consensus protocol using only objects with consensus number 2.

Explain the faulty step in this chain of reasoning.

Solution This space intentionally left blank. ■

7.10 Chapter Notes

Fischer, Lynch, and Paterson [**?**] were the first to prove that consensus is impossible in a message-passing system where a single thread can halt. They introduced the "critical-state" kind of impossibility argument. Herlihy and Wing [**?**] first formulated the notion of linearizability, and proved that this condition satisfies the non-blocking and locality properties.

Kruskal, Rudolph, and Snir [**?**] developed the notion of a read-modify-write operation as a spin-off of the NYU Ultracomputer project.

Herlihy [**?**] introduced the notion of a consensus number as a measure of computational power, and first showed most of the impossibility and universality results presented in this chapter and the next.