hierarchy of synchronization primitives, such that no primitive at one level can be used for a wait-free or lock-free implementation of any primitives at higher levels. The basic idea is simple: each class in the hierarchy 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 or lock-free implementation of an object with consensus number n from an object with a lower consensus number.

5.1 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 a single method `decide()`, as shown in Fig. 5.1. Each thread calls the `decide()` method with its input v *at most once*. The object's `decide()` method will return a value meeting the following conditions:

- *consistent*: all threads decide the same value,
- *valid*: the common decision value is some thread's input.

In other words, a concurrent consensus object is linearizable to a sequential consensus object in which the thread whose value was chosen completes its `decide()` first. Sometimes it is useful to focus on consensus problems where all inputs are either zero or one. We call this specialized problem *binary consensus*. To simplify the presentation, we focus here on binary consensus, but our claims apply verbatim to consensus in general.

We are interested in wait-free solutions to the consensus problem, that is, wait-free concurrent implementations of consensus objects. The reader will notice that since the `decide()` method of a given consensus object is executed only once by each thread, and there are a finite number of threads, by definition a lock-free implementation would also be wait-free and vice versa. Henceforth, we mention only wait-free implementations, and for historical reasons, call any class that implements *consensus* in a wait-free manner a *consensus protocol*.

```
1  public interface Consensus<T> {
2    T decide(T value);
3  }
```

Figure 5.1 Consensus object interface.

We will restrict ourselves to object classes with deterministic sequential specifications (i.e., ones in which each sequential method call has a single outcome).[2]

We want to understand whether a particular class of objects is powerful enough to solve the consensus problem. 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* 5.1.1.** A class C *solves* n-thread consensus if there exist a consensus protocol using any number of objects of class C and any number of atomic registers.

***Definition* 5.1.2.** 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 consensus number of the class is *infinite.*

***Corollary* 5.1.1.** 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.

5.1.1 States and Valence

A good place to start is to think about the simplest interesting case: binary consensus (i.e., inputs 0 or 1) for 2 threads (call them A and B). Each thread makes moves until it 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 possible states forms a tree, where each node represents a possible protocol state, and each edge represents a possible move by some thread. Fig. 5.2 shows the tree for a 2-thread protocol in which each thread moves twice. 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.

---

2 We avoid nondeterministic objects since their structure is significantly more complex. See the discussion in the notes at the end of this chapter.

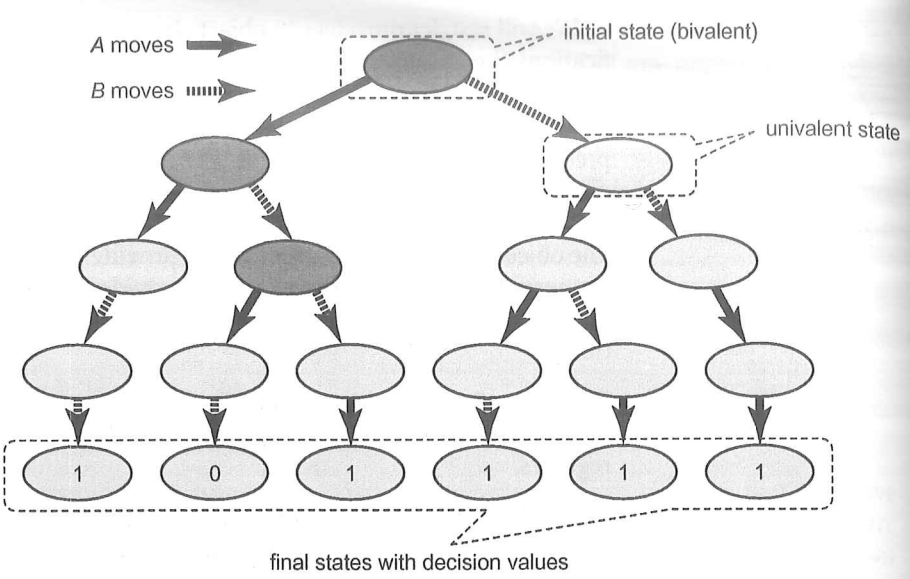

Figure 5.2 An execution tree for two threads *A* and *B*. The dark shaded nodes denote bivalent states, and the lighter ones univalent states.

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 Fig. 5.2, a bivalent state is a node whose descendants in the tree include both leaves labeled with 0 and leaves labeled with 1, while a univalent state is a node whose descendants include only leaves labeled with a single decision value.

Our next lemma says that an initial bivalent state exists. This observation means that the outcome of the protocol cannot be fixed in advance, but must depend on how `reads` and `writes` are interleaved.

Lemma 5.1.1. Every 2-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 (it cannot decide 1 because it has no way of distinguishing this state from the one in which *B* has input 0). 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. It follows that the initial state where *A* has input 0 and *B* has input 1 is bivalent. □

Lemma 5.1.2. 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.

Lemma 5.1.3. Every wait-free consensus protocol has a critical state.

Proof: Suppose not. By Lemma 5.1.2, 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.

5.2 Atomic Registers

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 an exercise 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*.

***Theorem* 5.2.1.** 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 5.1.3, we can run the protocol until it reaches a critical state s. Suppose A's next move carries the protocol to a 0-valent state, and B's next move carries the protocol to a 1-valent state. (If not, then switch thread names.) What

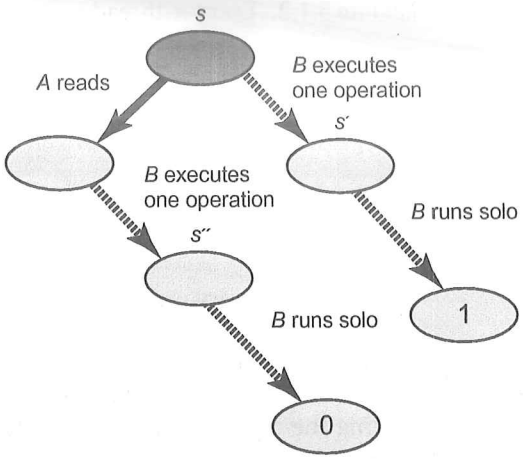

Figure 5.3 Case: *A* reads first. In the first execution scenario, *B* moves first, driving the protocol to a 1-valent state s', and then *B* runs solo and eventually decides 1. In the second execution scenario, *A* moves first, driving the protocol to a 0-valent state s''. *B* then runs solo starting in s'' and eventually decides 0.

methods could *A* and *B* be about to call? We now consider an exhaustive list of the possibilities: one of them reads from a register, they both write to separate registers, or they both write to the same register.

Suppose *A* is about to read a given register (*B* may be about to either read or write the same register or a different register), as depicted in Fig. 5.3. Consider two possible execution scenarios. In the first scenario, *B* moves first, driving the protocol to a 1-valent state s', and then *B* runs solo and eventually decides 1. In the second execution scenario, *A* moves first, then *B* executes one operation, 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* (the read *A* performed could only change its thread-local state which is not visible to *B*), which means that *B* must decide the same value in both scenarios, a contradiction.

Suppose, instead of this scenario, both threads are about to write to different registers, as depicted in Fig. 5.4. *A* is about to write to r_0 and *B* to r_1. Let us consider two possible execution 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, as depicted in Fig. 5.5. Again, consider two possible execution scenarios. In one scenario *A* writes first, the resulting protocol state s' is 0-valent, *B* then runs solo and decides 0. In

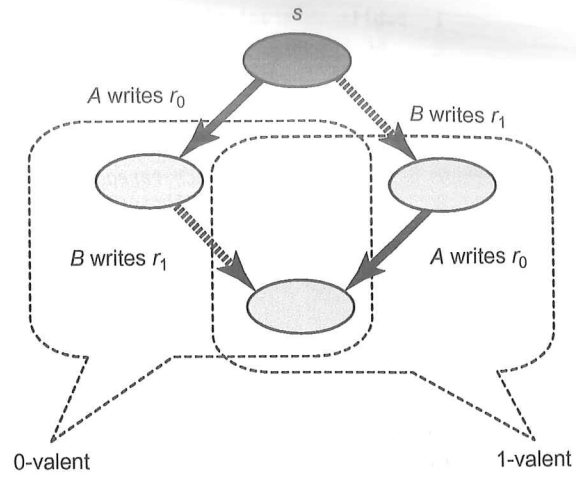

Figure 5.4 Case: *A* and *B* write to different registers.

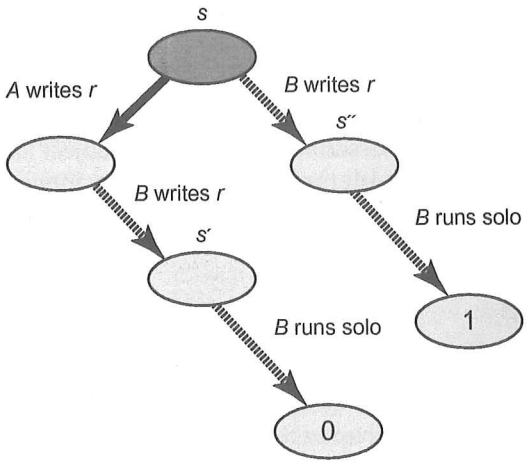

Figure 5.5 Case: *A* and *B* write to the same register.

another scenario, B writes first, the resulting protocol state s'' is 1-valent, B then runs solo and decides 1. The problem is that B cannot tell the difference between s' and s'' (because in both s' and s'' it overwrote the register r and obliterated any trace of A's write) so B must decide the same value starting from either state, a contradiction. □

***Corollary* 5.2.1.** It is impossible to construct a wait-free implementation of any object with consensus number greater than 1 using atomic registers.