CHAPTER

Foundations of shared memory

4

For millennia, chicken farmers around the world were forced to wait 5 to 6 weeks before they could tell male and female chickens apart. This delay meant weeks of wasted time and money, because unlike females, which grow to maturity, lay eggs, and can ultimately be fried Kentucky style, the young males have no value, and are discarded. Then, in the 1920s, Japanese scientists discovered an invaluable trick: Male chicks have a small bump in their vent (anus) that females lack. If you press on a chick's behind and examine it, you can tell immediately which chicks should be discarded (no need to wait 5 weeks). The trouble was that a sizable fraction of males and females had bumps that were not clearcut, and could be either male or female. Thus began the profession of "chicken sexing." Japan opened schools for training specialists who could sex on the order of 1000 chicks an hour with almost perfect accuracy. After proper training, expert chicken sexers could reliably determine the sex of day-old chicks at a glance using a collection of subtle perceptual cues. This profession continues to this day. In interviews, chicken sexers claim that in many cases they have no idea how they make their decisions. There is a technical name for this ability: *intuition*. Our unsettling example suggests that training and practice can enhance your intuition.

In this chapter, we begin our study of the foundations of *concurrent shared-memory computation*. As you read through the algorithms, you might question their "real-world value." If you do, remember that their value is in training you, the reader, to tell which types of algorithmic approaches work in a concurrent shared-memory setting, and which do not, even when it is hard to tell. This will help you discard bad ideas earlier, saving time and money.

The foundations of sequential computing were established in the 1930s by Alan Turing and Alonzo Church, who independently formulated what has come to be known as the *Church–Turing thesis*: Anything that *can* be computed, can be computed by a Turing machine (or, equivalently, by Church's lambda calculus). Any problem that cannot be solved by a Turing machine (such as deciding whether a program halts on any input) is universally considered to be unsolvable by any kind of practical computing device. The Church–Turing thesis is a *thesis*, not a theorem, because the notion of "what is computable" is not defined in a precise, mathematically rigorous way. Nevertheless, just about everyone believes this thesis.

To study concurrent shared-memory computation, we begin with a computational model. A shared-memory computation consists of multiple *threads*, each of which is a sequential program in its own right. These threads communicate by calling methods

The Art of Multiprocessor Programming. https://doi.org/10.1016/B978-0-12-415950-1.00013-6

of objects that reside in a shared memory. Threads are *asynchronous*, meaning that they may run at different speeds, and any thread can halt for an unpredictable duration at any time. This notion of asynchrony reflects the realities of modern multiprocessor architectures, where thread delays are unpredictable, ranging from microseconds (cache misses) to milliseconds (page faults) to seconds (scheduling interruptions).

The classical theory of sequential computability proceeds in stages. It starts with finite-state automata, moves on to push-down automata, and culminates in Turing machines. We, too, consider a progression of models for concurrent computing. We start with the simplest form of shared-memory computation: Concurrent threads read and write shared memory locations, which are called *registers* for historical reasons. We start with very simple registers, and we show how to use them to construct a series of more complex registers.

The classical theory of sequential computability is, for the most part, not concerned with efficiency: To show that a problem is computable, it is enough to show that it can be solved by a Turing machine. There is little incentive to make such a Turing machine efficient, because a Turing machine is not a practical model of computation. In the same way, we make little attempt to make our register constructions efficient. We are interested in understanding whether such constructions exist and how they work. They are not intended to be practical. We prefer inefficient but easy-to-understand constructions over efficient but complicated ones.

In particular, some of our constructions use *timestamps* (i.e., counter values) to distinguish older values from newer values. The problem with timestamps is that they grow without bound, and eventually overflow any fixed-size variable. Bounded solutions (such as the one in Section 2.8) are (arguably) more intellectually satisfying, and we encourage readers to investigate them further through the references provided in the chapter notes. Here, however, we focus on simpler, unbounded constructions, because they illustrate the fundamental principles of concurrent programming with less danger of becoming distracted by technicalities.

4.1 The space of registers

At the hardware level, threads communicate by reading and writing shared memory. A good way to understand interthread communication is to abstract away from hardware primitives, and to think about communication as happening through *shared concurrent objects*. Chapter 3 provides a detailed description of shared objects. For now, it suffices to recall the two key properties of their design: *safety*, defined by consistency conditions, and *liveness*, defined by progress conditions.

A *read–write register* (or just a *register*) is an object that encapsulates a value that can be observed by a `read()` method and modified by a `write()` method (these methods are often called *load* and *store*). Fig. 4.1 shows the `Register<T>` interface implemented by all registers. The type `T` of the value is typically `Boolean`, `Integer`, or a reference to an object. A register that implements the `Register<Boolean>` interface is called a *Boolean* register (sometimes 1 and 0 are used as synonyms for *true* and

```
1 public interface Register<T> {
2   T read();
3   void write(T v);
4 }
```

FIGURE 4.1

The `Register<T>` interface.

```
1 public class SequentialRegister<T> implements Register<T> {
2   private T value;
3   public T read() {
4     return value;
5   }
6   public void write(T v) {
7     value = v;
8   }
9 }
```

FIGURE 4.2

The `SequentialRegister` class.

false). A register that implements the `Register<Integer>` for a range of M integer values is called an *M-valued register*. We do not explicitly discuss any other kind of register, except to note that any algorithm that implements integer registers can be adapted to implement registers that hold references to other objects by representing the references as integers.

If method calls do not overlap, a register implementation should behave as shown in Fig. 4.2. On a multiprocessor, however, we expect method calls to overlap all the time, so we need to specify what the concurrent method calls mean.

An *atomic register* is a linearizable implementation of the sequential register class shown in Fig. 4.2. Informally, an atomic register behaves exactly as we would expect: Each read returns the "last" value written. A model in which threads communicate by reading and writing to atomic registers is intuitively appealing, and for a long time was the standard model of concurrent computation.

One approach to implementing atomic registers is to rely on mutual exclusion: protect each register with a mutual exclusion lock acquired by each call to `read()` or `write()`. Unfortunately, we cannot use the lock algorithms of Chapter 2 here; those algorithms accomplish mutual exclusion using registers, so it makes little sense to implement registers using mutual exclusion. Moreover, as we saw in Chapter 3, using mutual exclusion, even if it is deadlock- or starvation-free, would mean that the computation's progress would depend on the operating system scheduler to guarantee that threads never get stuck in critical sections. Since we wish to examine the basic building blocks of concurrent computation using shared objects, it makes little sense to assume the existence of a separate entity to provide the key progress property.

Here is a different approach: Recall that an object implementation is *wait-free* if each method call finishes in a finite number of steps, independently of how its execution is interleaved with steps of other concurrent method calls. The wait-free condition may seem simple and natural, but it has far-reaching consequences. In particular, it rules out any kind of mutual exclusion, and guarantees independent progress, that is, without making assumptions about the operating system scheduler. We therefore require our register implementations to be wait-free.

It is also important to specify how many readers and writers are expected. Not surprisingly, it is easier to implement a register that supports only a single reader and a single writer than one that supports multiple readers and writers. For brevity, we use SRSW for "single-reader, single-writer," MRSW for "multi-reader, single-writer," and MRMW for "multi-reader, multi-writer."

In this chapter, we address the following fundamental question:

> *Can any data structure implemented using the most powerful registers also be implemented using the weakest?*

Recall from Chapter 1 that any useful form of interthread communication must be persistent: The message sent must outlive the active participation of the sender. The weakest form of persistent synchronization is (arguably) the ability to set a single persistent bit in shared memory, and the weakest form of synchronization is (unarguably) none at all: If the act of setting a bit does not overlap the act of reading that bit, then the value read is the same as the value written. Otherwise, a read overlapping a write could return any value.

Different kinds of registers come with different guarantees that make them more or less powerful. For example, we have seen that registers may differ in the range of values they may encapsulate (e.g., Boolean versus M-valued), and in the number of readers and writers they support. They may also differ in the degree of consistency they provide.

An SRSW or MRSW register implementation is *safe* if:

- A `read()` call that does not overlap a `write()` call returns the value written by the most recent `write()` call. (The "most recent `write()` call" is well defined because there is a single writer.)
- A `read()` call that overlaps a `write()` call may return any value within the register's allowed range of values (e.g., 0 to $M - 1$ for an M-valued register).

Be aware that the term "safe" is a historical accident. Because they provide such weak guarantees, "safe" registers are actually quite unsafe.

Consider the history shown in Fig. 4.3. If the register is *safe*, then the three read calls might behave as follows:

- R^1 returns 0, the most recently written value.
- R^2 and R^3 are concurrent with $W(1)$, so they may return any value in the range of the register.

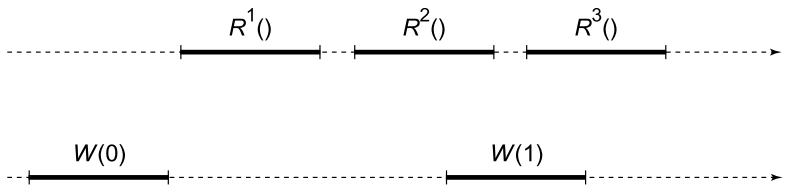

FIGURE 4.3

An SRSW register execution: R^i is the i-th read and $W(v)$ is a write of value v. Time flows from left to right. No matter whether the register is *safe*, *regular*, or *atomic*, R^1 must return 0, the most recently written value. If the register is *safe*, then because R^2 and R^3 are concurrent with $W(1)$, they may return any value in the range of the register. If the register is *regular*, R^2 and R^3 may each return either 0 or 1. If the register is *atomic*, then if R^2 returns 1, then R^3 must also return 1, and if R^2 returns 0, then R^3 may return 0 or 1.

It is convenient to define an intermediate level of consistency between safe and atomic. A *regular* register is an SRSW or MRSW register where writes do not happen atomically. Instead, while the `write()` call is in progress, the value being read may "flicker" between the old and new value before finally replacing the older value. More precisely:

- A regular register is safe, so any `read()` call that does not overlap a `write()` call returns the most recently written value.
- Suppose a `read()` call overlaps one or more `write()` calls. Let v^0 be the value written by the latest preceding `write()` call, and let $v^1, \ldots, v^k$ be the sequence of values written by `write()` calls that overlap the `read()` call. The `read()` call may return v^i for any i in the range $0 \ldots k$.

For the execution in Fig. 4.3, a regular register might behave as follows:

- R^1 returns the old value, 0.
- R^2 and R^3 each return either the old value 0 or the new value 1.

Regular registers are quiescently consistent (Chapter 3), but not vice versa. Both safe and regular registers permit only a single writer. Note that a regular register is actually a quiescently consistent single-writer sequential register.

For an atomic register, the execution in Fig. 4.3 might produce the following results:

- R^1 returns the old value, 0.
- If R^2 returns 1, then R^3 also returns 1.
- If R^2 returns 0, then R^3 returns either 0 or 1.

Fig. 4.4 shows a schematic view of the range of possible registers as a three-dimensional space: The register size defines one dimension, the numbers of readers and writers define another, and the register's consistency property defines the third.

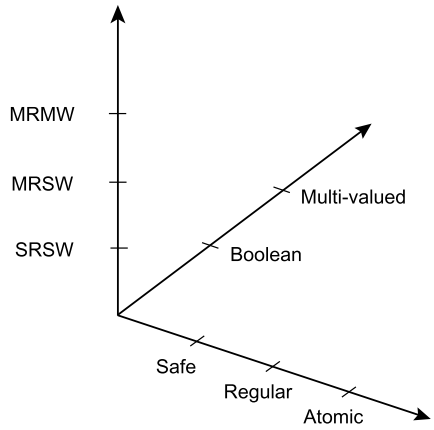

FIGURE 4.4

The three-dimensional space of possible read–write register-based implementations.

This view should not be taken literally: There are several combinations, such as multi-writer safe registers, that are not well defined.

To reason about algorithms for implementing regular and atomic registers, it is convenient to rephrase our definitions directly in terms of object histories. From now on, we consider only histories in which each `read()` call returns a value written by some `write()` call (regular and atomic registers do not allow reads to make up return values). For simplicity, we assume values read or written are unique.[1]

Recall that an object history is a sequence of *invocation* and *response* events, where an invocation event occurs when a thread calls a method, and a matching response event occurs when that call returns. A *method call* (or just a *call*) is the interval between matching invocation and response events (including the invocation and response events). Any history induces a partial order $\rightarrow$ on method calls, defined as follows: If m_0 and m_1 are method calls, $m_0 \rightarrow m_1$ if m_0's response event precedes m_1's call event. (See Chapter 3 for complete definitions.)

Any register implementation (whether safe, regular, or atomic) defines a total order on the `write()` calls called the *write order*, the order in which writes "take effect" in the register. For safe and regular registers, the write order is trivial because they allow only one writer at a time. For atomic registers, method calls have a linearization order. We use this order to index the write calls: Write call W^0 is ordered first, W^1 second, and so on. We use v^i to denote the unique value written by W^i. Note that for SRSW or MRSW safe or regular registers, the write order is exactly the same as the precedence order on writes.

[1] If values are not inherently unique, we can use the standard technique of appending to them auxiliary values invisible to the algorithm itself, used only in our reasoning to distinguish one value from another.

We use R^i to denote any read call that returns v^i. Note that although a history contains at most one W^i call, it might contain multiple R^i calls.

One can show that the following conditions provide a precise statement of what it means for a register to be regular. First, no read call returns a value from the future:

$$\text{It is never the case that} \quad R^i \to W^i. \tag{4.1.1}$$

Second, no read call returns a value from the distant past, that is, one that precedes the most recently written nonoverlapping value:

$$\text{It is never the case that for some } j, \quad W^i \to W^j \to R^i. \tag{4.1.2}$$

To prove that a register implementation is regular, we must show that its histories satisfy Conditions (4.1.1) and (4.1.2).

An atomic register satisfies one additional condition:

$$\text{if} \quad R^i \to R^j, \quad \text{then} \quad i \leq j. \tag{4.1.3}$$

This condition states that an earlier read cannot return a value later than that returned by a later read. Regular registers are *not* required to satisfy Condition (4.1.3). To show that a register implementation is atomic, we need first to define a write order, and then to show that its histories satisfy Conditions (4.1.1)–(4.1.3).

4.2 Register constructions

We now show how to implement a range of surprisingly powerful registers from simple safe Boolean SRSW registers. We consider a series of constructions, shown in Fig. 4.5, that implement stronger from weaker registers. These constructions imply that all read–write register types are equivalent, at least in terms of computability.

Base class	Implemented class	Section
safe SRSW	safe MRSW	4.2.1
safe Boolean MRSW	regular Boolean MRSW	4.2.2
regular Boolean MRSW	regular MRSW	4.2.3
regular SRSW	atomic SRSW	4.2.4
atomic SRSW	atomic MRSW	4.2.5
atomic MRSW	atomic MRMW	4.2.6
atomic MRSW	atomic snapshot	4.3

FIGURE 4.5

The sequence of register constructions.

In the last step, we show how atomic registers (and therefore safe registers) can implement an atomic snapshot: an array of MRSW registers written by different threads that can be read atomically by any thread.

Some of these constructions are more powerful than necessary to complete the sequence of derivations (for example, we do not need to provide the multi-reader property for regular and safe registers to complete the derivation of an atomic SRSW register). We present them anyway because they provide valuable insights.

Our code samples follow these conventions. When we display an algorithm to implement a particular kind of register, say, a safe Boolean MRSW register, we present the algorithm using a form somewhat like this:

```
class SafeBooleanMRSWRegister implements Register<Boolean>
  {
  ...
  }
```

While this notation makes clear the properties of the `Register<>` class being implemented, it becomes cumbersome when we want to use this class to implement other classes. Instead, when describing a class implementation, we use the following conventions to indicate whether a particular field is safe, regular, or atomic: A field otherwise named `mumble` is called `s_mumble` if it is safe, `r_mumble` if it is regular, and `a_mumble` if it is atomic. Other important aspects of the field, such as its type and whether it supports multiple readers or writers, are noted as comments within the code, and should also be clear from the context.

4.2.1 Safe MRSW registers

Fig. 4.6 shows how to construct a safe MRSW register from safe SRSW registers.

Lemma 4.2.1. The construction in Fig. 4.6 is a *safe MRSW register*.

Proof. If A's `read()` call does not overlap any `write()` call, then it does not overlap any `write()` call of the component register `s_table`[A], so the `read()` call returns

```
1  public class SafeBooleanMRSWRegister implements Register<Boolean> {
2    boolean[] s_table; // array of safe SRSW registers
3    public SafeBooleanMRSWRegister(int capacity) {
4      s_table = new boolean[capacity];
5    }
6    public Boolean read() {
7      return s_table[ThreadID.get()];
8    }
9    public void write(Boolean x) {
10     for (int i = 0; i < s_table.length; i++)
11       s_table[i] = x;
12   }
13 }
```

FIGURE 4.6

The `SafeBooleanMRSWRegister` class: a safe Boolean MRSW register.

the value of s_table[A], which is the most recently written value. If A's read() call overlaps a write() call, it is allowed to return any value. □

4.2.2 A regular Boolean MRSW register

The next construction, shown in Fig. 4.7, builds a regular Boolean MRSW register from a safe Boolean MRSW register. For Boolean registers, the only difference between safe and regular registers arises when the newly written value x is the same as the old. A regular register can only return x, while a safe register may return either Boolean value. We circumvent this problem simply by ensuring that a value is written only if it is distinct from the previously written value.

Lemma 4.2.2. The construction in Fig. 4.7 is a regular Boolean MRSW register.

Proof. A read() call that does not overlap any write() call returns the most recently written value. If the calls do overlap, there are two cases to consider:

- If the value being written is the same as the last value written, then the writer avoids writing to the safe register, ensuring that the reader reads the correct value.
- If the value written now is distinct from the last value written, then those values must be *true* and *false* because the register is Boolean. A concurrent read returns some value in the range of the register, namely, either *true* or *false*, either of which is correct. □

```
1  public class RegularBooleanMRSWRegister implements Register<Boolean> {
2    ThreadLocal<Boolean> last;
3    boolean s_value; // safe MRSW register
4    RegularBooleanMRSWRegister(int capacity) {
5      last = new ThreadLocal<Boolean>() {
6        protected Boolean initialValue() { return false; };
7      };
8    }
9    public void write(Boolean x) {
10     if (x != last.get()) {
11       last.set(x);
12       s_value = x;
13     }
14   }
15   public Boolean read() {
16     return s_value;
17   }
18 }
```

FIGURE 4.7

The RegularBooleanMRSWRegister class: a regular Boolean MRSW register constructed from a safe Boolean MRSW register.

```
1  public class RegularMRSWRegister implements Register<Byte> {
2    private static int RANGE = Byte.MAX_VALUE - Byte.MIN_VALUE + 1;
3    boolean[] r_bit = new boolean[RANGE]; // regular Boolean MRSW
4    public RegularMRSWRegister(int capacity) {
5      for (int i = 1; i < r_bit.length; i++)
6        r_bit[i] = false;
7      r_bit[0] = true;
8    }
9    public void write(Byte x) {
10     r_bit[x] = true;
11     for (int i = x - 1; i >= 0; i--)
12       r_bit[i] = false;
13   }
14   public Byte read() {
15     for (int i = 0; i < RANGE; i++)
16       if (r_bit[i]) {
17         return i;
18       }
19     return -1; // impossible
20   }
21 }
```

FIGURE 4.8

The `RegularMRSWRegister` class: a regular *M*-valued MRSW register.

4.2.3 A regular *M*-valued MRSW register

The jump from Boolean to M-valued registers is simple, if astonishingly inefficient: We represent the value in unary notation. In Fig. 4.8, we implement an M-valued register as an array of M Boolean registers. Initially the register is set to value zero, indicated by the 0th bit being set to *true*. A write method of value x writes *true* in location x and then in descending array-index order sets all lower locations to *false*. A reading method reads the locations in ascending index order until the first time it reads the value *true* in some index i. It then returns i. The example in Fig. 4.9 illustrates an *8-valued* register.

Lemma 4.2.3. The `read()` call in the construction in Fig. 4.8 always returns a value corresponding to a bit in $0..M-1$ set by some `write()` call.

Proof. The following property is invariant: If a reading thread is reading `r_bit`$[j]$, then some bit at index j or higher, written by a `write()` call, is set to *true*.

When the register is initialized, there are no readers; the constructor sets `r_bit`[0] to *true*. Assume a reader is reading `r_bit`$[j]$, and that `r_bit`$[k]$ is *true* for $k \geq j$.

- If the reader advances from j to $j+1$, then `r_bit`$[j]$ is *false*, so $k > j$ (i.e., a bit greater than or equal to $j+1$ is *true*).
- The writer clears `r_bit`$[k]$ only if it has set a higher `r_bit`$[\ell]$ to *true* for $\ell > k$. □

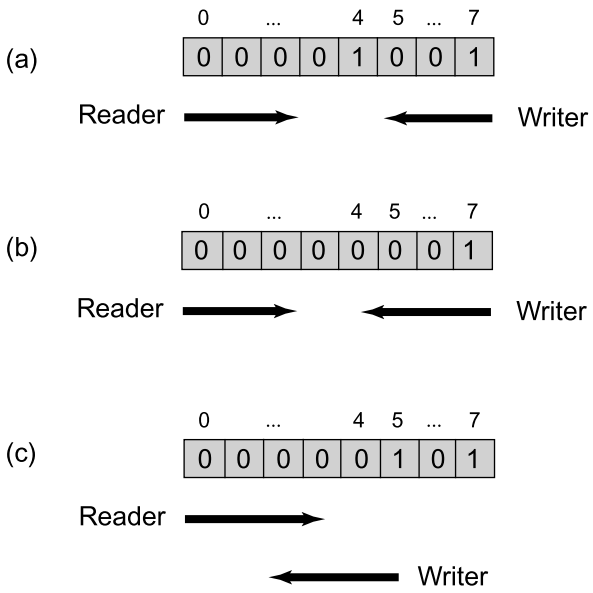

FIGURE 4.9

The `RegularMRSWRegister` class: an execution of a regular *8-valued* MRSW register. The values *false* and *true* are represented by 0 and 1 respectively. In part (a), the value prior to the write was 4, and thread *W*'s write of 7 is not read by thread *R* because *R* reaches array entry 4 before *W* overwrites *false* at that location. In part (b), entry 4 is overwritten by *W* before it is read, so the read returns 7. In part (c), *W* starts to write 5. Since it wrote array entry 5 before it was read, the reader returns 5 even though entry 7 is also set to *true*.

Lemma 4.2.4. The construction in Fig. 4.8 is a regular M-valued MRSW register.

Proof. For any read, let x be the value written by the most recent nonoverlapping `write()`. At the time the `write()` completed, `a_bit`$[x]$ was set to *true*, and `a_bit`$[i]$ is *false* for $i < x$. By Lemma 4.2.3, if the reader returns a value that is not x, then it observed some `a_bit`$[j]$, $j \neq x$, to be *true*, and that bit must have been set by a concurrent write, proving Conditions (4.1.1) and (4.1.2). □

4.2.4 An atomic SRSW register

We show how to construct an atomic SRSW register from a regular SRSW register. (Note that our construction uses unbounded timestamps.)

A regular register satisfies Conditions (4.1.1) and (4.1.2), while an atomic register must also satisfy Condition (4.1.3). Since a regular SRSW register has no concurrent reads, the only way Condition (4.1.3) can be violated is if two reads that overlap the same write read values out-of-order, the first returning v^i and the latter returning v^j, where $j < i$.

Fig. 4.10 describes a class of values that each have an added tag that contains a timestamp. Our implementation of an `AtomicSRSWRegister`, shown in Fig. 4.11, uses

```
1  public class StampedValue<T> {
2    public long stamp;
3    public T value;
4    // initial value with zero timestamp
5    public StampedValue(T init) {
6      stamp = 0;
7      value = init;
8    }
9    // later values with timestamp provided
10   public StampedValue(long ts, T v) {
11     stamp = ts;
12     value = v;
13   }
14   public static StampedValue max(StampedValue x, StampedValue y) {
15     if (x.stamp > y.stamp) {
16       return x;
17     } else {
18       return y;
19     }
20   }
21   public static StampedValue MIN_VALUE = new StampedValue(null);
22 }
```

FIGURE 4.10

The `StampedValue<T>` class: allows a timestamp and a value to be read or written together.

these tags to order write calls so that they can be ordered properly by concurrent read calls. Each read remembers the latest (highest timestamp) timestamp/value pair ever read, so that it is available to future reads. If a later read then reads an earlier value (one having a lower timestamp), it ignores that value and simply uses the remembered latest value. Similarly, the writer remembers the latest timestamp it wrote, and tags each newly written value with a later timestamp (i.e., a timestamp greater by 1).

This algorithm requires the ability to read or write a value and a timestamp as a single unit. In a language such as C, we would treat both the value and the timestamp as uninterpreted bits ("raw seething bits"), and use bit shifting and logical masking to pack and unpack both values in and out of one or more words. In Java, it is easier to create a `StampedValue<T>` structure that holds a timestamp/value pair, and to store a *reference* to that structure in the register.

Lemma 4.2.5. The construction in Fig. 4.11 is an atomic SRSW register.

Proof. The register is regular, so Conditions (4.1.1) and (4.1.2) are met. The algorithm satisfies Condition (4.1.3) because writes are totally ordered by their timestamps, and if a read returns a given value, a later read cannot read an earlier written value, since it would have a lower timestamp. □

```
1  public class AtomicSRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    ThreadLocal<StampedValue<T>> lastRead;
4    StampedValue<T> r_value;            // regular SRSW timestamp-value pair
5    public AtomicSRSWRegister(T init) {
6      r_value = new StampedValue<T>(init);
7      lastStamp = new ThreadLocal<Long>() {
8        protected Long initialValue() { return 0; };
9      };
10     lastRead = new ThreadLocal<StampedValue<T>>() {
11       protected StampedValue<T> initialValue() { return r_value; };
12     };
13   }
14   public T read() {
15     StampedValue<T> value = r_value;
16     StampedValue<T> last = lastRead.get();
17     StampedValue<T> result = StampedValue.max(value, last);
18     lastRead.set(result);
19     return result.value;
20   }
21   public void write(T v) {
22     long stamp = lastStamp.get() + 1;
23     r_value = new StampedValue(stamp, v);
24     lastStamp.set(stamp);
25   }
26 }
```

FIGURE 4.11

The `AtomicSRSWRegister` class: an atomic SRSW register constructed from a regular SRSW register.

4.2.5 An atomic MRSW register

To understand how to construct an atomic MRSW register from atomic SRSW registers, we first consider a simple algorithm based on direct use of the construction in Section 4.2.1, which took us from safe SRSW to safe MRSW registers. Let the SRSW registers composing the table array `a_table`$[0..n-1]$ be atomic instead of safe, with all other calls remaining the same: The writer writes the array locations in increasing index order and then each reader reads and returns its associated array entry. The result is not an atomic multi-reader register. Condition (4.1.3) holds for any single reader because each reader reads from an atomic register, yet it does not hold for distinct readers. Consider, for example, a write that starts by setting the first SRSW register `a_table`[0], and is delayed before writing the remaining locations `a_table`$[1..n-1]$. A subsequent read by thread 0 returns the correct new value, but a subsequent read by thread 1 that completely follows the read by thread 0 reads and returns the earlier value because the writer has yet to update `a_table`$[1..n-1]$. We

```
1  public class AtomicMRSWRegister<T> implements Register<T> {
2    ThreadLocal<Long> lastStamp;
3    private StampedValue<T>[][] a_table; // each entry is an atomic SRSW register
4    public AtomicMRSWRegister(T init, int readers) {
5      lastStamp = new ThreadLocal<Long>() {
6         protected Long initialValue() { return 0; };
7      };
8      a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
9      StampedValue<T> value = new StampedValue<T>(init);
10     for (int i = 0; i < readers; i++) {
11       for (int j = 0; j < readers; j++) {
12         a_table[i][j] = value;
13       }
14     }
15   }
16   public T read() {
17     int me = ThreadID.get();
18     StampedValue<T> value = a_table[me][me];
19     for (int i = 0; i < a_table.length; i++) {
20       value = StampedValue.max(value, a_table[i][me]);
21     }
22     for (int i = 0; i < a_table.length; i++) {
23       if (i == me) continue;
24       a_table[me][i] = value;
25     }
26     return value;
27   }
28   public void write(T v) {
29     long stamp = lastStamp.get() + 1;
30     lastStamp.set(stamp);
31     StampedValue<T> value = new StampedValue<T>(stamp, v);
32     for (int i = 0; i < a_table.length; i++) {
33       a_table[i][i] = value;
34     }
35   }
36 }
```

FIGURE 4.12

The `AtomicMRSWRegister` class: an atomic MRSW register constructed from atomic SRSW registers.

address this problem by having earlier reader threads *help out* later threads by telling them which value they read.

This implementation appears in Fig. 4.12. The n threads share an n-by-n array `a_table`$[0..n-1][0..n-1]$ of stamped values. As in Section 4.2.4, we use time-stamped values to allow early reads to tell later reads which of the values read is

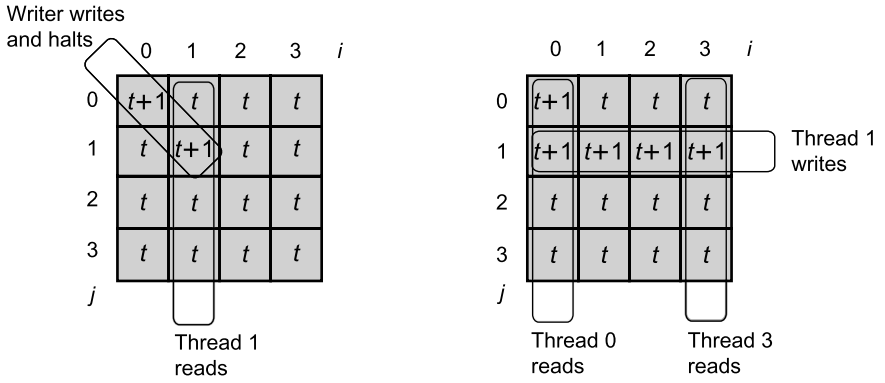

FIGURE 4.13

An execution of the atomic MRSW register. Each reader thread has an index between 0 and 3, and we refer to each thread by its index. Here, the writer writes a new value with timestamp $t+1$ to locations `a_table`[0][0] and `a_table`[1][1] and then halts. Then, thread 1 reads its corresponding column `a_table`[i][1] for all i, and writes its corresponding row `a_table`[1][i] for all i, returning the new value with timestamp $t+1$. Threads 0 and 3 both read completely after thread 1's read. Thread 0 reads `a_table`[0][0] with value $t+1$. Thread 3 cannot read the new value with timestamp $t+1$ because the writer has yet to write `a_table`[3][3]. Nevertheless, it reads `a_table`[1][3] and returns the correct value with timestamp $t+1$ that was read by the earlier thread 1.

the latest. The locations along the diagonal, `a_table`[i][i] for all i, correspond to the registers in the failed simple construction mentioned earlier. The writer simply writes the diagonal locations one after the other with a new value and a timestamp that increases from one `write`() call to the next. Each reader A first reads `a_table`[A][A] as in the earlier algorithm. It then uses the remaining SRSW locations `a_table`[A][B], $A \neq B$, for communication between readers A and B. Each reader A, after reading `a_table`[A][A], checks to see if some other reader has read a later value by traversing its corresponding column (`a_table`[B][A] for all B), and checking if it contains a later value (one with a higher timestamp). The reader then lets all later readers know the latest value it read by writing this value to all locations in its corresponding row (`a_table`[A][B] for all B). It thus follows that after a read by A is completed, every later read by B sees the last value A read (since it reads `a_table`[A][B]). Fig. 4.13 shows an example execution of the algorithm.

Lemma 4.2.6. The construction in Fig. 4.12 is an atomic MRSW register.

Proof. First, no reader returns a value from the future, so Condition (4.1.1) is clearly satisfied. By construction, `write`() calls write strictly increasing timestamps. The key to understanding this algorithm is the simple observation that the maximum timestamp along any row or column is also strictly increasing. If A writes v with timestamp t, then any subsequent `read`() call by B (where A's call completely precedes B's) reads (from the diagonal of `a_table`) a maximum timestamp greater than

or equal to t, satisfying Condition (4.1.2). Finally, as noted earlier, if a read call by A completely precedes a read call by B, then A writes a stamped value with timestamp t to B's row, so B chooses a value with a timestamp greater than or equal to t, satisfying Condition (4.1.3). □

On an intuitive, "chicken sexing" level, note that our counterexample that violates atomicity is caused by two read events that do not overlap, the earlier read reading an older value than the latter read. If the reads overlapped, we could have reordered their linearization points however we wanted. However, because the two reads do not overlap, the order of their linearization points is fixed, so we cannot satisfy the atomicity requirement. This is the type of counterexample we should look for when designing algorithms. (We used this same counterexample, by the way, in the single-reader atomic register construction.)

Our solution used two algorithmic tools: timestamping, which appears later in many practical algorithms, and indirect helping, where one thread tells the others what it read. In this way, if a writer pauses after communicating information to only a subset of readers, then those readers collaborate by passing on that information.

4.2.6 An atomic MRMW register

Here is how to construct an atomic MRMW register from an array of atomic MRSW registers, one per thread.

To write to the register, A reads all the array elements, chooses a timestamp higher than any it has observed, and writes a stamped value to array element A. To read the register, a thread reads all the array elements, and returns the one with the highest timestamp. This is exactly the timestamp algorithm used by the Bakery algorithm of Section 2.7. As in the Bakery algorithm, we resolve ties in favor of the thread with the lesser index, in other words, using a lexicographic order on pairs of timestamp and thread IDs.

Lemma 4.2.7. The construction in Fig. 4.14 is an atomic MRMW register.

Proof. Define the write order among `write()` calls based on the lexicographic order of their timestamps and thread IDs so that the `write()` call by A with timestamp t_A precedes the `write()` call by B with timestamp t_B if $t_A < t_B$ or if $t_A = t_B$ and $A < B$. We leave as an exercise to the reader to show that this lexicographic order is consistent with $\rightarrow$. As usual, index `write()` calls in write order: $W^0, W^1, \ldots$.

Clearly a `read()` call cannot read a value written in `a_table[]` after it is completed, and any `write()` call completely preceded by the read has a timestamp higher than all those written before the read is completed, implying Condition (4.1.1).

Consider Condition (4.1.2), which prohibits skipping over the most recent preceding `write()`. Suppose a `write()` call by A preceded a write call by B, which in turn preceded a `read()` by C. If $A = B$, then the later write overwrites `a_table[`A`]` and the `read()` does not return the value of the earlier write. If $A \neq B$, then since A's timestamp is smaller than B's timestamp, any C that sees both returns B's value (or one with higher timestamp), meeting Condition (4.1.2).

```
1  public class AtomicMRMWRegister<T> implements Register<T>{
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public AtomicMRMWRegister(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      StampedValue<T> value = new StampedValue<T>(init);
6      for (int j = 0; j < a_table.length; j++) {
7        a_table[j] = value;
8      }
9    }
10   public void write(T value) {
11     int me = ThreadID.get();
12     StampedValue<T> max = StampedValue.MIN_VALUE;
13     for (int i = 0; i < a_table.length; i++) {
14       max = StampedValue.max(max, a_table[i]);
15     }
16     a_table[me] = new StampedValue(max.stamp + 1, value);
17   }
18   public T read() {
19     StampedValue<T> max = StampedValue.MIN_VALUE;
20     for (int i = 0; i < a_table.length; i++) {
21       max = StampedValue.max(max, a_table[i]);
22     }
23     return max.value;
24   }
25 }
```

FIGURE 4.14

Atomic MRMW register.

Finally, we consider Condition (4.1.3), which prohibits values from being read out of write order. Consider any `read()` call by A completely preceding a `read()` call by B, and any `write()` call by C which is ordered before the `write()` by D in the write order. We must show that if A returns D's value, then B does not return C's value. If $t_C < t_D$, then if A reads timestamp t_D from `a_table`$[D]$, B reads t_D or a higher timestamp from `a_table`$[D]$, and does not return the value associated with t_C. If $t_C = t_D$, that is, the writes were concurrent, then from the write order, $C < D$, so if A reads timestamp t_D from `a_table`$[D]$, B also reads t_D from `a_table`$[D]$, and returns the value associated with t_D (or higher), even if it reads t_C in `a_table`$[C]$. □

Our series of constructions shows that one can construct a wait-free atomic multi-valued MRMW register from safe Boolean SRSW registers. Naturally, no one wants to write a concurrent algorithm using safe registers, but these constructions show that any algorithm using atomic registers can be implemented on an architecture that supports only safe registers. Later on, when we consider more realistic architectures, we return to the theme of implementing algorithms that assume strong synchronization properties on architectures that directly provide only weaker properties.

```
1 public interface Snapshot<T> {
2   public void update(T v);
3   public T[] scan();
4 }
```

FIGURE 4.15

The `Snapshot` interface.

4.3 Atomic snapshots

We have seen how a register value can be read and written atomically. What if we want to read multiple register values atomically? We call such an operation an *atomic snapshot*.

An atomic snapshot constructs an instantaneous view of an array of MRSW registers. We construct a wait-free snapshot, meaning that a thread can take a snapshot of the array without delaying any other thread. Atomic snapshots can be useful for backups or checkpoints.

The `Snapshot` interface (Fig. 4.15) is just an array of atomic MRSW registers, one for each thread. The `update()` method writes a value v to the calling thread's register in that array; the `scan()` method returns an atomic snapshot of that array.

Our goal is to construct a wait-free implementation that is equivalent (that is, linearizable) to the sequential specification shown in Fig. 4.16. The key property of this sequential implementation is that `scan()` returns a collection of values, each corresponding to the latest preceding `update()`; that is, it returns a collection of register values that existed together in the same instant.

4.3.1 An obstruction-free snapshot

We begin with a `SimpleSnapshot` class for which `update()` is wait-free but `scan()` is obstruction-free. We then extend this algorithm to make `scan()` wait-free.

As in the atomic MRSW register construction, each value is a `StampedValue<T>` object with `stamp` and `value` fields. Each `update()` call increments the timestamp.

A *collect* is the nonatomic act of copying the register values one-by-one into an array. If we perform two collects one after the other, and both collects read the same set of timestamps, then we know that there was an interval during which no thread updated its register, so the result of the collect is a snapshot of the array immediately after the end of the first collect. We call such a pair of collects a *clean double collect*.

In the construction shown in the `SimpleSnapshot<T>` class (Fig. 4.17), each thread repeatedly calls `collect()` (line 25), and returns as soon as it detects a clean double collect (one in which both sets of timestamps were identical).

This construction always returns correct values. The `update()` calls are wait-free, but `scan()` is not because any call can be repeatedly interrupted by `update()`, and may run forever without completing. It is, however, obstruction-free: a `scan()` completes if it runs by itself for long enough.

```
1  public class SeqSnapshot<T> implements Snapshot<T> {
2    T[] a_value;
3    public SeqSnapshot(int capacity, T init) {
4      a_value = (T[]) new Object[capacity];
5      for (int i = 0; i < a_value.length; i++) {
6        a_value[i] = init;
7      }
8    }
9    public synchronized void update(T v) {
10     a_value[ThreadID.get()] = v;
11   }
12   public synchronized T[] scan() {
13     T[] result = (T[]) new Object[a_value.length];
14     for (int i = 0; i < a_value.length; i++)
15       result[i] = a_value[i];
16     return result;
17   }
18 }
```

FIGURE 4.16

A sequential snapshot.

Note that we use timestamps to verify the double collect, and not the values in the registers. Why? We encourage the reader to come up with a counterexample in which the repeated appearance of the same value is interleaved with others so that reading the same value creates the illusion that "nothing has changed." This is a common mistake that concurrent programmers make, trying to save the space needed for timestamps by using the values being written as indicators of a property. We advise against it: More often than not, this will lead to a bug, as in the case of the clean double collect: It must be detected by checking timestamps, not the equality of the sets of values collected.

4.3.2 A wait-free snapshot

To make the `scan()` method wait-free, each `update()` call *helps* a potentially interfering `scan()` by taking a snapshot before writing to its register. A `scan()` that repeatedly fails to take a clean double collect can use the snapshot from one of the interfering `update()` calls as its own. The tricky part is that we must make sure that the snapshot taken from the helping update is one that can be linearized within the `scan()` call's execution interval.

We say that a thread *moves* if it completes an `update()`. If thread A fails to make a clean collect because thread B moved, then can A simply take B's most recent snapshot as its own? Unfortunately, no. As illustrated in Fig. 4.18, it is possible for A to see B move when B's snapshot was taken before A started its `scan()` call, so the snapshot did not occur within the interval of A's scan.

```
1  public class SimpleSnapshot<T> implements Snapshot<T> {
2    private StampedValue<T>[] a_table; // array of atomic MRSW registers
3    public SimpleSnapshot(int capacity, T init) {
4      a_table = (StampedValue<T>[]) new StampedValue[capacity];
5      for (int i = 0; i < capacity; i++) {
6        a_table[i] = new StampedValue<T>(init);
7      }
8    }
9    public void update(T value) {
10     int me = ThreadID.get();
11     StampedValue<T> oldValue = a_table[me];
12     StampedValue<T> newValue = new StampedValue<T>((oldValue.stamp)+1, value);
13     a_table[me] = newValue;
14   }
15   private StampedValue<T>[] collect() {
16     StampedValue<T>[] copy = (StampedValue<T>[]) new StampedValue[a_table.length];
17     for (int j = 0; j < a_table.length; j++)
18       copy[j] = a_table[j];
19     return copy;
20   }
21   public T[] scan() {
22     StampedValue<T>[] oldCopy, newCopy;
23     oldCopy = collect();
24     collect: while (true) {
25       newCopy = collect();
26       if (! Arrays.equals(oldCopy, newCopy)) {
27         oldCopy = newCopy;
28         continue collect;
29       }
30       T[] result = (T[]) new Object[a_table.length];
31       for (int j = 0; j < a_table.length; j++)
32         result[j] = newCopy[j].value;
33       return result;
34     }
35   }
36 }
```

FIGURE 4.17

Simple snapshot object.

The wait-free construction is based on the following observation: If a scanning thread A sees a thread B move *twice* while it is performing repeated collects, then B executed a complete `update()` call within the interval of A's `scan()`, so it is correct for A to use B's snapshot.

Figs. 4.19 and 4.20 show the wait-free snapshot algorithm. Each `update()` calls `scan()`, and appends the result of the scan to the value (in addition to the timestamp).

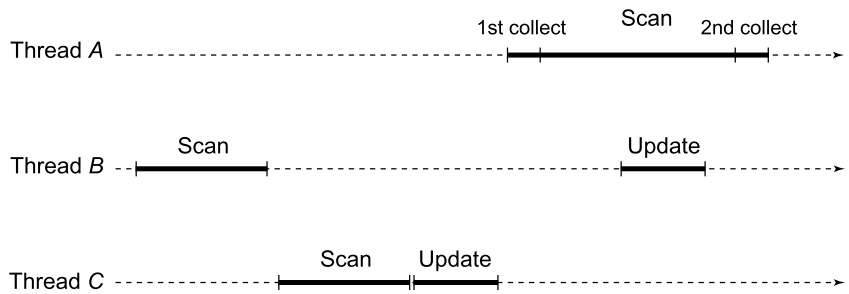

FIGURE 4.18

Here is why a thread *A* that fails to complete a clean double collect cannot simply take the latest snapshot of a thread *B* that performed an `update()` during *A*'s second collect. *B*'s snapshot was taken before *A* started its `scan()`, i.e., *B*'s snapshot did not overlap *A*'s scan. The danger, illustrated here, is that a thread *C* could have called `update()` after *B*'s `scan()` and before *A*'s `scan()`, making it incorrect for *A* to use the results of *B*'s `scan()`.

```
1  public class StampedSnap<T> {
2    public long stamp;
3    public T value;
4    public T[] snap;
5    public StampedSnap(T value) {
6      stamp = 0;
7      value = value;
8      snap = null;
9    }
10   public StampedSnap(long ts, T v, T[] s) {
11     stamp = ts;
12     value = v;
13     snap = s;
14   }
15 }
```

FIGURE 4.19

The stamped snapshot class.

More precisely, each value written to a register has the structure shown in Fig. 4.19: a `stamp` field incremented each time the thread updates its value, a `value` field containing the register's actual value, and a `snap` field containing that thread's most recent scan. The snapshot algorithm is described in Fig. 4.20. A scanning thread creates a Boolean array called `moved[]` (line 24), which records which threads have been observed to move in the course of the scan. As before, each thread performs two collects (lines 25 and 27) and tests whether any thread's timestamp has changed. If no thread's timestamp has changed, then the collect is clean, and the scan returns the result of the collect. If any thread's timestamp has changed (line 29), the scanning thread tests the

```
1  public class WFSnapshot<T> implements Snapshot<T> {
2    private StampedSnap<T>[] a_table; // array of atomic MRSW registers
3    public WFSnapshot(int capacity, T init) {
4      a_table = (StampedSnap<T>[]) new StampedSnap[capacity];
5      for (int i = 0; i < a_table.length; i++) {
6        a_table[i] = new StampedSnap<T>(init);
7      }
8    }
9    private StampedSnap<T>[] collect() {
10     StampedSnap<T>[] copy = (StampedSnap<T>[]) new StampedSnap[a_table.length];
11     for (int j = 0; j < a_table.length; j++)
12       copy[j] = a_table[j];
13     return copy;
14   }
15   public void update(T value) {
16     int me = ThreadID.get();
17     T[] snap = scan();
18     StampedSnap<T> oldValue = a_table[me];
19     StampedSnap<T> newValue = new StampedSnap<T>(oldValue.stamp+1, value, snap);
20     a_table[me] = newValue;
21   }
22   public T[] scan() {
23     StampedSnap<T>[] oldCopy, newCopy;
24     boolean[] moved = new boolean[a_table.length]; // initially all false
25     oldCopy = collect();
26     collect: while (true) {
27       newCopy = collect();
28       for (int j = 0; j < a_table.length; j++) {
29         if (oldCopy[j].stamp != newCopy[j].stamp) {
30           if (moved[j]) {
31             return newCopy[j].snap;
32           } else {
33             moved[j] = true;
34             oldCopy = newCopy;
35             continue collect;
36           }
37         }
38       }
39       T[] result = (T[]) new Object[a_table.length];
40       for (int j = 0; j < a_table.length; j++)
41         result[j] = newCopy[j].value;
42       return result;
43     }
44   }
45 }
```

FIGURE 4.20

Single-writer atomic snapshot class.

moved[] array to detect whether this is the second time this thread has moved (line 30). If so, it returns that thread's scan (line 31); otherwise, it updates moved[] and resumes the outer loop (line 32).

4.3.3 Correctness arguments

In this section, we review the correctness arguments for the wait-free snapshot algorithm a little more carefully.

Lemma 4.3.1. If a scanning thread makes a clean double collect, then the values it returns were the values that existed in the registers in some state of the execution.

Proof. Consider the interval between the last read of the first collect and the first read of the second collect. If any register were updated in that interval, the timestamps would not match, and the double collect would not be clean. □

Lemma 4.3.2. If a scanning thread A observes changes in another thread B's timestamp during two different double collects, then the value of B's register read during the last collect was written by an update() call that began after the first collect started.

Proof. If during a scan(), two successive reads by A of B's register return different timestamps, then at least one write by B occurs between this pair of reads. Thread B writes to its register as the final step of an update() call, so some update() call by B ended sometime after the first read by A, and the write step of another update() call occurs between the last pair of reads by A. The claim follows because only B writes to its register. □

Lemma 4.3.3. The values returned by a scan() were in the registers at some state between the call's invocation and response.

Proof. If the scan() call made a clean double collect, then the claim follows from Lemma 4.3.1. If the call took the scan value from another thread B's register, then by Lemma 4.3.2, the scan value found in B's register was obtained by a scan() call by B whose interval lies between A's first and last reads of B's register. Either B's scan() call had a clean double collect, in which case the result follows from Lemma 4.3.1, or there is an embedded scan() call by a thread C occurring within the interval of B's scan() call. This argument can be applied inductively, noting that there can be at most $n - 1$ nested calls before we run out of threads, where n is the maximum number of threads (see Fig. 4.21). Eventually, some nested scan() call must have had a clean double collect. □

Lemma 4.3.4. Every scan() or update() returns after at most $O(n^2)$ reads or writes.

Proof. Consider a particular scan(). There are only $n - 1$ other threads, so after n double collects, either one double collect is clean, or some thread is observed to move twice. The claim follows because each double collect does $O(n)$ reads. □

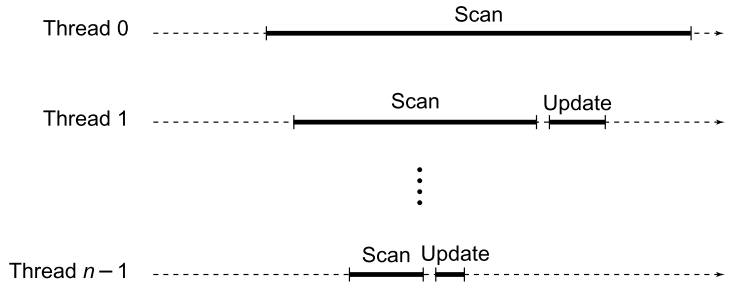

FIGURE 4.21

There can be at most $n - 1$ nested calls of `scan()` before we run out of threads, where n is the maximum number of threads. The `scan()` by thread $n - 1$, contained in the intervals of all other `scan()` calls, must have a clean double collect.

By Lemma 4.3.3, the values returned by a `scan()` form a snapshot as they are all in the registers in some state during the call: linearize the call at that point. Similarly, linearize `update()` calls at the point the register is written.

Theorem 4.3.5. The code in Fig. 4.20 is a wait-free snapshot implementation.

Our wait-free atomic snapshot construction is another, somewhat different example of the dissemination approach we discussed in our atomic register constructions. In this example, threads tell other threads about their snapshots, and those snapshots are reused. Another useful trick is that even if one thread interrupts another and prevents it from completing, we can still guarantee wait-freedom if the interrupting thread completes the interrupted thread's operation. This helping paradigm is extremely useful in designing multiprocessor algorithms.

4.4 Chapter notes

Alonzo Church introduced lambda calculus around 1935 [30]. Alan Turing defined the Turing machine in a classic paper in 1937 [163]. Leslie Lamport defined the notions of *safe*, *regular*, and *atomic* registers and the register hierarchy, and was the first to show that one could implement nontrivial shared memory from safe bits [99, 105]. Gary Peterson suggested the problem of constructing atomic registers [139]. Jaydev Misra gave an axiomatic treatment of atomic registers [128]. The notion of *linearizability*, which generalizes Lamport's and Misra's notions of atomic registers, is due to Herlihy and Wing [75]. Susmita Haldar and Krishnamurthy Vidyasankar gave a bounded atomic MRSW register construction from regular registers [55]. The problem of constructing an atomic multi-reader register from atomic single-reader registers was mentioned as an open problem by Leslie Lamport [99,105] and by Paul Vitányi and Baruch Awerbuch [165], who were the first to propose an approach for atomic MRMW register design. The first solution is due to Jim Anderson, Mohamed

Gouda, and Ambuj Singh [87,160]. Other atomic register constructions, to name only a few, were proposed by Jim Burns and Gary Peterson [25], Richard Newman-Wolfe [134], Lefteris Kirousis, Paul Spirakis, and Philippas Tsigas [92], Amos Israeli and Amnon Shaham [86], and Ming Li, John Tromp and Paul Vitányi [113]. The simple timestamp-based atomic MRMW construction we present here is due to Danny Dolev and Nir Shavit [39].

Collect operations were first formalized by Mike Saks, Nir Shavit, and Heather Woll [152]. The first atomic snapshot constructions were discovered concurrently and independently by Jim Anderson [10] and Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit [2]. The latter algorithm is the one presented here. Later snapshot algorithms are due to Elizabeth Borowsky and Eli Gafni [21] and Yehuda Afek, Gideon Stupp, and Dan Touitou [4].

The timestamps in all the algorithms mentioned in this chapter can be bounded so that the constructions themselves use registers of bounded size. Bounded timestamp systems were introduced by Amos Israeli and Ming Li [85], and bounded concurrent timestamp systems by Danny Dolev and Nir Shavit [39].

Horsey [78] has a beautiful article on chicken sexing and its relation to intuition.

4.5 Exercises

Exercise 4.1. Consider the safe Boolean MRSW construction shown in Fig. 4.6. True or false: If we replace the safe Boolean SRSW register array with an array of safe M-valued SRSW registers, then the construction yields a safe M-valued MRSW register. Justify your answer.

Exercise 4.2. Consider the safe Boolean MRSW construction shown in Fig. 4.6. True or false: If we replace the safe Boolean SRSW register array with an array of regular Boolean SRSW registers, then the construction yields a regular Boolean MRSW register. Justify your answer.

Exercise 4.3. Consider the safe Boolean MRSW construction shown in Fig. 4.6. True or false: If we replace the safe Boolean SRSW register array with an array of regular M-valued SRSW registers, then the construction yields a regular M-valued MRSW register. Justify your answer.

Exercise 4.4. Consider the regular Boolean MRSW construction shown in Fig. 4.7. True or false: If we replace the safe Boolean MRSW register with a safe M-valued MRSW register, then the construction yields a regular M-valued MRSW register. Justify your answer.

Exercise 4.5. Consider the atomic MRSW construction shown in Fig. 4.12. True or false: If we replace the atomic SRSW registers with regular SRSW registers, then the construction still yields an atomic MRSW register. Justify your answer.

Exercise 4.6. Give an example of a quiescently consistent register execution that is not regular.

```
1  public class AtomicSRSWRegister implements Register<int> {
2    private static int RANGE = M;
3    boolean[] r_bit = new boolean[RANGE]; // atomic boolean SRSW
4    public AtomicSRSWRegister(int capacity) {
5      for (int i = 1; i <= RANGE; i++)
6        r_bit[i] = false;
7      r_bit[0] = true;
8    }
9    public void write(int x) {
10     r_bit[x] = true;
11     for (int i = x - 1; i >= 0; i--)
12       r_bit[i] = false;
13   }
14   public int read() {
15     for (int i = 0; i <= RANGE; i++)
16       if (r_bit[i]) {
17         return i;
18       }
19     return -1; // impossible
20   }
21 }
```

FIGURE 4.22

Boolean to M-valued atomic SRSW register algorithm.

Exercise 4.7. You are given the algorithm in Fig. 4.22 for constructing an atomic M-valued SRSW register using atomic Boolean SRSW registers. Does this proposal work? Either prove the correctness or present a counterexample.

Exercise 4.8. Imagine running a 64-bit system on a 32-bit system, where each 64-bit memory location (register) is implemented using two atomic 32-bit memory locations (registers). A write operation is implemented by simply writing the first 32 bits in the first register and then the second 32 bits in the second register. A read, similarly, reads the first half from the first register, then reads the second half from the second register, and returns the concatenation. What is the strongest property that this 64-bit register satisfies?

- safe register,
- regular register,
- atomic register,
- it does not satisfy any of these properties.

Exercise 4.9. Does Peterson's two-thread mutual exclusion algorithm work if the shared atomic `flag` registers are replaced by regular registers?

Exercise 4.10. Consider the following implementation of a register in a distributed, message passing system. There are n processors $P_0, \ldots, P_{n-1}$ arranged in a ring,

where P_i can send messages only to $P_{i+1 \bmod n}$. Messages are delivered in FIFO order along each link. Each processor keeps a copy of the shared register.

- To read the register, the processor reads the copy in its local memory.
- A processor P_i starts a `write()` call of value v to register x, by sending the message "P_i: write v to x" to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_j: write v to x," for $i \neq j$, then it writes v to its local copy of x, and forwards the message to $P_{i+1 \bmod n}$.
- If P_i receives a message "P_i: write v to x," then it writes v to its local copy of x, and discards the message. The `write()` call is now complete.

Give a short justification or counterexample.
If `write()` calls never overlap,

- is this register implementation regular?
- is it atomic?

If multiple processors call `write()`,

- is this register implementation safe?

Exercise 4.11. Fig. 4.23 shows an implementation of a multivalued *write-once*, MRSW register from an array of multivalued safe, MRSW registers. Remember, there is one writer, who can overwrite the register's initial value with a new value, but it can only write once. You do not know the register's initial value.

Is this implementation regular? Atomic?

```
1  class WriteOnceRegister implements Register{
2    private SafeMRSWRegister[] s = new SafeMRSWRegister[3];
3
4    public void write(int x) {
5      s[0].write(x);
6      s[1].write(x);
7      s[2].write(x);
8    }
9    public int read() {
10     v2 = s[2].read()
11     v1 = s[1].read()
12     v0 = s[0].read()
13     if (v0 == v1) return v0;
14     else if (v1 == v2) return v1;
15     else return v0;
16   }
17 }
```

FIGURE 4.23

Write-once register.

Exercise 4.12. A (single-writer) register is *1-regular* if the following conditions hold:

- If a read() operation does not overlap with any write() operations, then it returns the value written by the last write() operation.
- If a read() operation overlaps with exactly one write() operation, then it returns a value written either by the last write() operation or the concurrent write() operation.
- Otherwise, a read() operation may return an arbitrary value.

Construct an SRSW M-valued 1-regular register using $O(\log M)$ SRSW Boolean regular registers. Explain why your construction works.

Exercise 4.13. Prove that the safe Boolean MRSW register construction from safe Boolean SRSW registers illustrated in Fig. 4.6 is a correct implementation of a regular MRSW register if the component registers are regular SRSW registers.

Exercise 4.14. Define a *wraparound* register that has the property that there is a value k such that writing the value v sets the value of the register to $v \bmod k$.

If we replace the Bakery algorithm's shared variables with either (a) regular, (b) safe, or (c) atomic wraparound registers, then does it still satisfy (1) mutual exclusion and (2) FIFO ordering?

You should provide six answers (some may imply others). Justify each claim.