CHAPTER

Concurrent objects

3

The behavior of concurrent objects is best described through their safety and liveness properties, often called *correctness* and *progress*. In this chapter, we examine various ways of specifying correctness and progress.

All notions of correctness for concurrent objects are based on some notion of equivalence with sequential behavior, but different notions are appropriate for different systems. We examine three correctness conditions. *Sequential consistency* is a strong condition, often useful for describing standalone systems such as hardware memory interfaces. *Linearizability* is an even stronger condition that supports *composition*: It is useful for describing systems composed from *linearizable* components. *Quiescent consistency* is appropriate for applications that require high performance at the cost of placing relatively weak constraints on object behavior.

Along a different dimension, different method implementations provide different progress guarantees. Some are *blocking*, where the delay of one thread can prevent other threads from making progress; some are *nonblocking*, where the delay of a thread cannot delay other threads indefinitely.

3.1 Concurrency and correctness

What does it mean for a concurrent object to be correct? Fig. 3.1 shows a simple lock-based concurrent "first-in-first-out" (FIFO) queue. The `enq()` and `deq()` methods synchronize using a mutual exclusion lock of the kind studied in Chapter 2. We immediately intuit that this implementation should be correct: Because each method holds an exclusive lock the entire time it accesses and updates fields, the method calls take effect sequentially.

This idea is illustrated in Fig. 3.2, which shows an execution in which thread A enqueues a, B enqueues b, and C dequeues twice, first throwing `EmptyException`, and second returning b. Overlapping intervals indicate concurrent method calls. All the method calls overlap in time. In this figure, as in others, time moves from left to right, and dark lines indicate intervals. The intervals for a single thread are displayed along a single horizontal line. When convenient, the thread name appears on the left. A bar represents an interval with a fixed start and stop time. A bar with dotted lines on the right represents an interval with a fixed start-time and an unknown stop-time. The label "q.enq(x)" means that a thread enqueues item x at object q, while "q.deq(x)" means that the thread dequeues item x from object q.

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[])new Object[capacity];
  }
  public void enq(T x) throws FullException {
    lock.lock();
    try {
      if (tail - head == items.length)
        throw new FullException();
      items[tail % items.length] = x;
      tail++;
    } finally {
      lock.unlock();
    }
  }
  public T deq() throws EmptyException {
    lock.lock();
    try {
      if (tail == head)
        throw new EmptyException();
      T x = items[head % items.length];
      head++;
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

FIGURE 3.1

A lock-based FIFO queue. The queue's items are kept in an array `items`, where `head` is the index of the next item (if any) to dequeue, and `tail` is the index of the first open array slot (modulo the capacity). The `lock` field contains a lock that ensures that methods are mutually exclusive. Initially `head` and `tail` are zero, and the queue is empty. If `enq()` finds the queue is full (i.e., `head` and `tail` differ by the queue capacity), then it throws an exception. Otherwise, there is room, so `enq()` stores the item at array entry for `tail`, and then increments `tail`. The `deq()` method works in a symmetric way.

The timeline shows which thread holds the lock. Here, *C* acquires the lock, observes the queue to be empty, releases the lock, and throws an exception. It does not modify the queue. *B* acquires the lock, inserts *b*, and releases the lock. *A* acquires the lock, inserts *a*, and releases the lock. *C* reacquires the lock, dequeues *b*, releases the

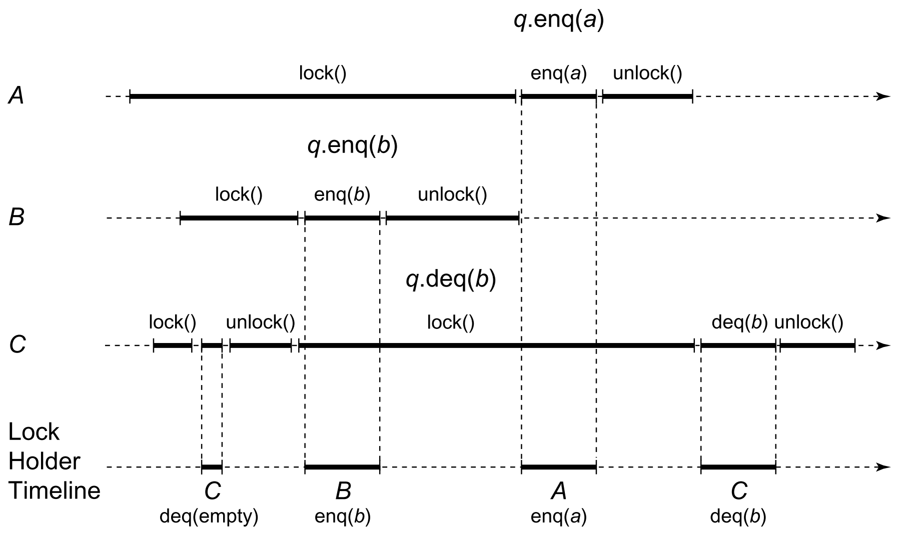

FIGURE 3.2

Lock-based queue execution. Here, *C* acquires the lock, observes the queue to be empty, releases the lock, and throws an exception. *B* acquires the lock, inserts *b*, and releases the lock. *A* acquires the lock, inserts *a*, and releases the lock. *C* reacquires the lock, dequeues *b*, releases the lock, and returns *b*.

lock, and returns *b*. Each of these calls takes effect sequentially, and we can easily verify that dequeuing *b* before *a* is consistent with our understanding of sequential FIFO queue behavior.

Unfortunately, it follows from Amdahl's law (Chapter 1) that concurrent objects whose methods hold exclusive locks, and therefore effectively execute one after the other, are less desirable than ones with finer-grained locking or no locks at all. We therefore need a way to specify the behavior required of concurrent objects, and to reason about their implementations, without relying on method-level locking.

Consider the alternative concurrent queue implementation in Fig. 3.3. It has almost the same internal representation as the lock-based queue of Fig. 3.1; the only difference is the absence of a lock. We claim that this implementation is correct provided there is only a single enqueuer and a single dequeuer. But it is no longer easy to explain why. If the queue supported concurrent enqueues or concurrent dequeues, it would not even be clear what it means for a queue *to be FIFO*.

The lock-based queue example illustrates a useful principle: It is easier to reason about the behavior of concurrent objects if we can somehow map their concurrent executions to sequential ones, and otherwise limit our reasoning to these sequential executions. This principle is the key to the correctness properties introduced in this chapter. Therefore, we begin by considering specifications of sequential objects.

```
class WaitFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;
  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity];
  }
  public void enq(T x) throws FullException {
    if (tail - head == items.length)
      throw new FullException();
    items[tail % items.length] = x;
    tail++;
  }
  public T deq() throws EmptyException {
    if (tail - head == 0)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  }
}
```

FIGURE 3.3

A single-enqueuer/single-dequeuer FIFO queue. The structure is identical to that of the lock-based FIFO queue, except that there is no need for the lock to coordinate access.

3.2 Sequential objects

An *object* in languages such as Java and C++ is a container for data and a set of *methods*, which are the only way to manipulate those data. Each object has a *class*, which defines the object's methods and how they behave. An object has a well-defined *state* (for example, the FIFO queue's current sequence of items). There are many ways to describe how an object's methods behave, ranging from formal specifications to plain English. The application program interface (API) documentation that we use every day lies somewhere in between.

The API documentation typically says something like the following: If the object is in such-and-such a state before you call the method, then the object will be in some other state when the method returns, and the call will return a particular value, or throw a particular exception. This kind of description divides naturally into a *precondition*, which describes the object's state before invoking the method, and a *postcondition*, which describes the object's state and return value when the method returns. A change to an object's state is sometimes called a *side effect*.

For example, a FIFO queue might be described as follows: The class provides two methods, `enq()` and `deq()`. The queue state is a sequence of items, possibly empty. If the queue state is a sequence q (precondition), then a call to `enq(`z`)` leaves the queue in state $q \cdot z$ (postcondition with side effect), where "·" denotes concatenation. If the queue object is nonempty, say, $a \cdot q$ (precondition), then the `deq()` method removes the sequence's first element a, leaving the queue in state q, and returns this element (postcondition). If, instead, the queue object is empty (precondition), the method throws `EmptyException` and leaves the queue state unchanged (postcondition, no side effect).

This style of documentation, called a *sequential specification*, is so familiar that it is easy to overlook how elegant and powerful it is. The length of the object's documentation is linear in the number of methods, because each method can be described in isolation. There are a vast number of potential interactions among methods, and all such interactions are characterized succinctly by the methods' side effects on the object state. The object's documentation describes the object state before and after each call, and we can safely ignore any intermediate states that the object may assume while the method call is in progress.

Defining objects in terms of preconditions and postconditions makes sense in a *sequential* model of computation, where a single thread manipulates a collection of objects. But this familiar style of documentation fails for objects shared by multiple threads. If an object's methods can be invoked concurrently by multiple threads, then method calls can overlap in time, and it no longer makes sense to talk about their order. What does it mean, for example, if x and y are enqueued onto a FIFO queue during overlapping intervals? Which will be dequeued first? Can we continue to describe methods in isolation, via preconditions and postconditions, or must we provide explicit descriptions of every possible interaction among every possible collection of concurrent method calls?

Even the notion of an object's state becomes confusing. In a single-threaded program, an object must assume a meaningful state only between method calls.[1] In a multithreaded program, however, overlapping method calls may be in progress at every instant, so a concurrent object might *never* be between method calls. Every method call must be prepared to encounter an object state that reflects the incomplete effects of concurrent method calls, a problem that does not arise in single-threaded programs.

3.3 Sequential consistency

One way to develop an intuition about how concurrent objects should behave is to review examples of concurrent computations involving simple objects, and decide, in each case, whether the behavior agrees with our intuition about how the objects should behave.

Method calls take time. A *method call* is the interval that starts with an *invocation* event and continues until the corresponding *response* event, if any. Method calls by

[1] There is an exception: Care must be taken if one method partially changes an object's state and then calls another method of that same object.

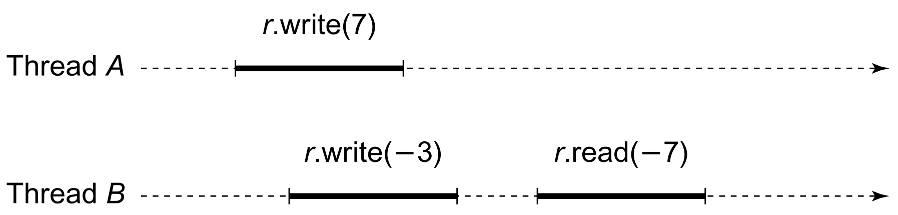

FIGURE 3.4

Why each method call should appear to take effect in one-at-a-time order. Two threads concurrently write −3 and 7 to a shared register *r*. Later, one thread reads *r* and returns the value −7. We expect to find either 7 or −3 in the register, not a mixture of both.

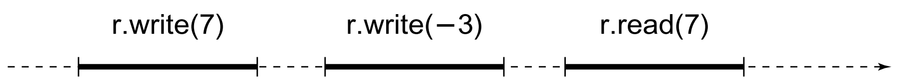

FIGURE 3.5

Why method calls should appear to take effect in program order. This behavior is not acceptable because the value the thread read is not the last value it wrote (and no other thread writes to the register).

concurrent threads may overlap, while method calls by a single thread are always sequential (nonoverlapping, one-after-the-other). We say a method call is *pending* if its invocation event has occurred, but its response event has not.

For historical reasons, the object version of a read–write memory location is called a *register* (see Chapter 4). In Fig. 3.4, two threads concurrently write −3 and 7 to a shared register *r* (as before, "r.read(x)" means that a thread reads value x from register object r, and similarly for "r.write(x)"). Later, one thread reads *r* and returns the value −7. This behavior is surprising. We expect to find either 7 or −3 in the register, not a mixture of both. This example suggests the following principle:

Principle 3.3.1. Method calls should appear to happen in a one-at-a-time, sequential order.

By itself, this principle is too weak to be useful. For example, it permits reads to always return the object's initial state, even in sequential executions (i.e., executions in which method calls do not overlap). Or consider the execution in Fig. 3.5, in which a single thread writes 7 and then −3 to a shared register *r*. Later, it reads *r* and returns 7. For some applications, this behavior might not be acceptable because the value the thread read is not the value it wrote most recently. The order in which a single thread issues method calls is called its *program order*. (Method calls by different threads are unrelated by program order.) In this example, we were surprised that operation calls did not take effect in program order. This example suggests the following principle:

Principle 3.3.2. Method calls should appear to take effect in program order.

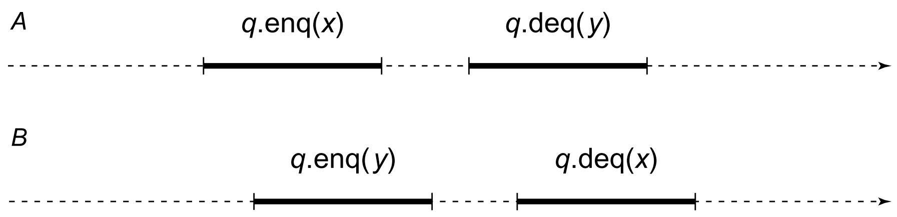

FIGURE 3.6

There are two possible sequential orders that can justify this execution. Both orders are consistent with the method calls' program order, and either one is enough to show that the execution is sequentially consistent.

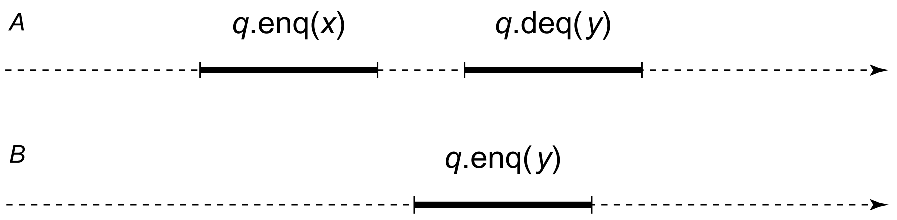

FIGURE 3.7

Sequential consistency versus real-time order. Thread *A* enqueues *x*, and later thread *B* enqueues *y*, and finally *A* dequeues *y*. This execution may violate our intuitive notion of how a FIFO queue should behave because the method call enqueuing *x* finishes before the method call enqueuing *y* starts, so *y* is enqueued after *x*. But it is dequeued before x. Nevertheless, this execution is sequentially consistent.

This principle ensures that purely sequential computations behave the way we expect. Together, Principles 3.3.1 and 3.3.2 define *sequential consistency*, a correctness condition that is widely used in the literature on multiprocessor synchronization.

Sequential consistency requires that method calls act as if they occurred in a sequential order consistent with program order. That is, there is a way to order all the method calls in any concurrent execution so that they (1) are consistent with program order and (2) meet the object's sequential specification. Multiple sequential orders may satisfy these conditions. For example, in Fig. 3.6, thread A enqueues x while thread B enqueues y, and then A dequeues y while B dequeues x. Two sequential orders explain these results: (1) A enqueues x, B enqueues y, B dequeues x, and then A dequeues y, or (2) B enqueues y, A enqueues x, A dequeues y, and then B dequeues x. Both orders are consistent with the program order; either suffices to show that the execution is sequentially consistent.

3.3.1 Sequential consistency versus real-time order

In Fig. 3.7, thread A enqueues x, and later B enqueues y, and finally A dequeues y. This execution may violate our intuitive notion of how a FIFO queue should behave:

The call enqueuing x finishes before the call enqueuing y starts, so y is enqueued after x. But it is dequeued before x. Nevertheless, this execution is sequentially consistent. Even though the call that enqueues x happens before the call that enqueues y, these calls are unrelated by program order, so sequential consistency is free to reorder them. When one operation completes before another begins, we say that the first operation precedes the second in the *real-time order*. This example shows that sequential consistency need not preserve the real-time order.

One could argue whether it is acceptable to reorder method calls whose intervals do not overlap, even if they occur in different threads. For example, we might be unhappy if we deposit our paycheck on Monday, but the bank bounces our rent check the following Friday because it reordered our deposit after your withdrawal.

3.3.2 Sequential consistency is nonblocking

How much does sequential consistency limit concurrency? Specifically, under what circumstances does sequential consistency require one method call to block waiting for another to complete? Perhaps surprisingly, the answer is (essentially) *never*. More precisely, for any pending method call in a sequentially consistent concurrent execution, there is some sequentially consistent response, that is, a response to the invocation that could be given immediately without violating sequential consistency. We say that a correctness condition with this property is *nonblocking*. Sequential consistency is a *nonblocking* correctness condition.

Note that this observation does not mean that it is easy to figure out a sequentially consistent response for a pending method call, only that the correctness condition itself does not stand in the way. The observation holds only for *total methods*, which

REMARK 3.3.1

The term *nonblocking* is used to denote several different notions. In this context, referring to correctness conditions, it means that for any pending call of a total method, there is a response that satisfies the correctness condition. In Section 3.8, referring to progress conditions, it means that a progress condition guarantees that the delay of one or more threads cannot prevent other threads from making progress. When referring to an object implementation, it means that the implementation meets a nonblocking progress condition. (It may even be used with finer granularity, referring to an individual method of an object implementation that cannot be prevented from making progress by the delay of other threads.) In the systems literature, a *nonblocking* operation returns immediately without waiting for the operation to take effect, whereas a *blocking* operation does not return until the operation is complete. (*Blocking* is also used to describe a lock implementation that suspends a thread that tries to acquire a lock that is held by another thread, as opposed to *spinning* implementations, which we discuss in Chapter 7). Unfortunately, these various uses are all too well established to change, but it should be clear from the context which meaning is intended.

are defined for every object state (i.e., for any state on which a total method is invoked, there is some response allowed by the sequential specification). There is, of course, no sequentially consistent response to a method call if there is no response that satisfies the sequential specification. Our informal description of sequential consistency thus far is not sufficient to capture this and other important details, such as what it exactly means for an execution with pending method calls to be sequentially consistent. We make this notion more precise in Section 3.6.

3.3.3 Compositionality

Any sufficiently complex system must be designed and implemented in a *modular* fashion. Components are designed, implemented, and proved correct independently. Each component makes a clear distinction between its *implementation*, which is hidden, and its *interface*, which characterizes the guarantees it makes to the other components. For example, if a concurrent object's interface states that it is a sequentially consistent FIFO queue, then users of the queue need not know anything about how the queue is implemented. The result of composing individually correct components that rely only on one another's interfaces should itself be a correct system.

A correctness property $\mathcal{P}$ is *compositional* if, whenever each object in the system satisfies $\mathcal{P}$, the system as a whole satisfies $\mathcal{P}$. Compositionality is important because it enables a system to be assembled easily from independently derived components. A system based on a noncompositional correctness property cannot rely solely on its components' interfaces: Some kind of additional constraints are needed to ensure that the components are actually compatible.

Is sequential consistency compositional? That is, is the result of composing multiple sequentially consistent objects itself sequentially consistent? The answer, unfortunately, is *no*. In Fig. 3.8, two threads, A and B, call enqueue and dequeue methods for two queue objects, p and q. It is not hard to see that p and q are each sequentially consistent: The sequence of method calls for p is the same as in the sequentially consistent execution shown in Fig. 3.7, and the behavior of q is symmetric. Nevertheless, the execution as a whole is *not* sequentially consistent.

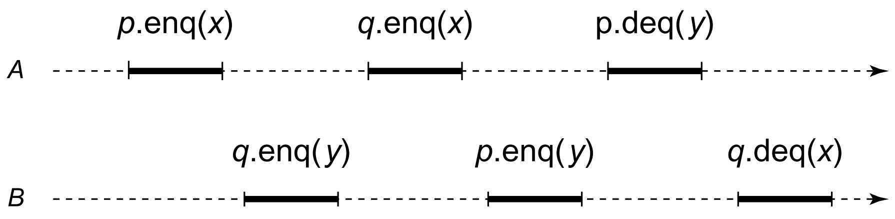

FIGURE 3.8

Sequential consistency is not compositional. Two threads, *A* and *B*, call enqueue and dequeue methods on two queue objects, *p* and *q*. It is not hard to see that *p* and *q* are each sequentially consistent, yet the execution as a whole is *not* sequentially consistent.

To see that there is no correct sequential execution of these methods calls that is consistent with their program order, assume, by way of contradiction, that there is such an execution. We use the following shorthand: $\langle p.\texttt{enq}(x)\ A\rangle \to \langle p.\texttt{deq}()\texttt{x}\ B\rangle$ means that any sequential execution must order A's enqueue of x at p before B's dequeue of x at p, and so on. Because p is FIFO and A dequeues y from p, y must have been enqueued at p before x:

$$\langle p.\texttt{enq}(y)\ B\rangle \to \langle p.\texttt{enq}(x)\ A\rangle.$$

Similarly, x must have been enqueued onto q before y:

$$\langle q.\texttt{enq}(x)\ A\rangle \to \langle q.\texttt{enq}(y)\ B\rangle.$$

But program order implies that

$$\langle p.\texttt{enq}(x)\ A\rangle \to \langle q.\texttt{enq}(x)\ A\rangle \quad \text{and} \quad \langle q.\texttt{enq}(y)\ B\rangle \to \langle p.\texttt{enq}(y)\ B\rangle.$$

Together, these orderings form a cycle.

3.4 Linearizability

Sequential consistency has a serious drawback: it is not compositional. That is, the result of composing sequentially consistent components is not itself necessarily sequentially consistent. To fix this shortcoming, we replace the requirement that method calls appear to happen in program order with the following stronger constraint:

Principle 3.4.1. Each method call should appear to take effect instantaneously at some moment between its invocation and response.

This principle states that the real-time order of method calls must be preserved. We call this correctness property *linearizability*. Every linearizable execution is sequentially consistent, but not vice versa.

3.4.1 Linearization points

The usual way to show that a concurrent object implementation is linearizable is to identify for each method a *linearization point*, an instant when the method takes effect. We say that a method *is linearized at* its linearization point. For lock-based implementations, any point within each method's critical section can serve as its linearization point. For implementations that do not use locks, the linearization point is typically a single step where the effects of the method call become visible to other method calls.

For example, recall the single-enqueuer/single-dequeuer queue of Fig. 3.3. This implementation has no critical sections, and yet we can identify linearization points for its methods. For example, if a `deq()` method returns an item, its linearization point is when the `head` field is updated (line 17). If the queue is empty, the `deq()` method is linearized when it reads the `tail` field (line 14). The `enq()` method is similar.

3.4.2 Linearizability versus sequential consistency

Like sequential consistency, linearizability is nonblocking: There is a linearizable response to any pending call of a total method. In this way, linearizability does not limit concurrency.

Threads that communicate only through a single shared object (e.g., the memory of a shared-memory multiprocessor) cannot distinguish between sequential consistency and linearizability. Only an external observer, who can see that one operation precedes another in the real-time order, can tell that a sequentially consistent object is not linearizable. For this reason, the difference between sequential consistency and linearizability is sometimes called *external consistency*. Sequential consistency is a good way to describe standalone systems, where composition is not an issue. However, if the threads share multiple objects, these objects may be external observers for each other, as we saw in Fig. 3.8.

Unlike sequential consistency, linearizability is compositional: The result of composing linearizable objects is linearizable. For this reason, linearizability is a good way to describe components of large systems, where components must be implemented and verified independently. Because we are interested in systems that compose, most (but not all) data structures considered in this book are linearizable.

3.5 Quiescent consistency

For some systems, implementors may be willing to trade consistency for performance. That is, we may relax the consistency condition to allow cheaper, faster, and/or more efficient implementations. One way to relax consistency is to enforce ordering only when an object is *quiescent*, that is, when it has no pending method calls. Instead of Principles 3.3.2 and 3.4.1, we would adopt the following principle:

Principle 3.5.1. Method calls separated by a period of quiescence should appear to take effect in their real-time order.

For example, suppose A and B concurrently enqueue x and y in a FIFO queue. The queue becomes quiescent, and then C enqueues z. We are not able to predict the relative order of x and y in the queue, but we do know they are ahead of z.

Together, Principles 3.3.1 and 3.5.1 define a correctness property called *quiescent consistency*. Informally, it says that any time an object becomes quiescent, the execution so far is equivalent to some sequential execution of the completed calls.

As an example of quiescent consistency, consider the shared counter from Chapter 1. A quiescently consistent shared counter would return numbers, not necessarily in the order of the `getAndIncrement()` requests, but always without duplicating or omitting a number. The execution of a quiescently consistent object is somewhat like a game of musical chairs: At any point, the music might stop, that is, the state could become quiescent. At that point, each pending method call must return an index so that all the indices together meet the specification of a sequential counter, implying no duplicated or omitted numbers. In other words, a quiescently consistent counter is an *index distribution* mechanism, useful as a "loop counter" in programs that do not care about the order in which indices are issued.

3.5.1 Properties of quiescent consistency

Note that sequential consistency and quiescent consistency are *incomparable*: There exist sequentially consistent executions that are not quiescently consistent, and vice versa. Quiescent consistency does not necessarily preserve program order, and sequential consistency is unaffected by quiescent periods. On the other hand, linearizability is stronger than both quiescent consistency and sequential consistency. That is, a linearizable object is both quiescently consistent and sequentially consistent.

Like sequential consistency and linearizability, quiescent consistency is nonblocking: Any pending call to a total method in a quiescently consistent execution can be completed.

Quiescent consistency is compositional: A system composed of quiescently consistent objects is itself quiescently consistent. It follows that quiescently consistent objects can be composed to construct more complex quiescently consistent objects. It is interesting to consider whether we could build useful systems using quiescent consistency rather than linearizability as the fundamental correctness property, and how the design of such systems would differ from existing system designs.

3.6 Formal definitions

We now consider more precise definitions. We focus on linearizability, since it is the property most often used in this book. We leave it as an exercise to provide analogous definitions for quiescent consistency and sequential consistency.

Informally, a concurrent object is linearizable if each method call appears to take effect instantaneously at some moment between that method's invocation and return events. This statement suffices for most informal reasoning, but a more precise formulation is needed to cover some tricky cases (such as method calls that have not returned), and for more rigorous styles of argument.

3.6.1 Histories

We model the observable behavior of an execution of a concurrent system by a sequence of *events* called a *history*, where an event is an *invocation* or *response* of a method. We write a method invocation as $\langle x.m(a^*)\ A\rangle$, where x is an object, m is a method name, a^* is a sequence of arguments, and A is a thread. We write a method response as $\langle x : t(r^*)\ A\rangle$, where t is either OK or an exception name, and r^* is a sequence of result values.

An invocation and a response *match* if they name the same object and thread. An invocation in H is *pending* if no matching response follows the invocation. A *method* *call* in a history H is a pair consisting of an invocation and either the next matching response in H or a special $\perp$ value (pronounced "bottom") if the invocation is pending (i.e., if there is no subsequent matching response). We say that a method call is *pending* if its invocation is pending, and that it is *complete* otherwise. A history is complete if all its method calls are complete. For a history H, we denote the subsequence of H consisting of all events of complete method calls (i.e., eliding all the pending invocations of H) by $\mathit{complete}(H)$.

The *interval* of a method call in a history H is the history's sequence of events starting from its invocation and ending with its response, or the suffix of H starting from its invocation if the method call is pending. Two method calls *overlap* if their intervals overlap.

A history is *sequential* if its first event is an invocation, and each invocation, except possibly the last, is followed immediately by a matching response, and each response is immediately preceded by an invocation. No method calls overlap in a sequential history, and a sequential history has at most one pending invocation.

A *subhistory* of a history H is a subsequence of H. Sometimes we focus on a single thread or object: A *thread subhistory*, $H|A$ ("H at A"), of a history H is the subsequence of all events in H whose thread names are A. An *object subhistory* $H|x$ is similarly defined for an object x. We require each thread to complete each method call before calling another method: A history H is *well formed* if each thread subhistory is sequential. Henceforth, we consider only well-formed histories. Although thread subhistories of a well-formed history are always sequential, object subhistories need not be; method calls to the same object may overlap in a well-formed history. Finally, because what matters in the end is how each thread views the history, we say that two histories are *equivalent* if every thread has the same thread subhistory in both histories; that is, H and H' are equivalent if $H|A = H'|A$ for every thread A.

How can we tell whether a concurrent object is correct? Or, said differently, how do we define correctness for a concurrent object? The basic idea is to require a concurrent execution to be equivalent, in some sense, to some sequential history; the exact sense of equivalence is different for different correctness properties. We assume that we can tell whether a sequential object is correct, that is, whether a sequential object history is a legal history for the object's class. A *sequential specification* for an object is just a set of legal sequential histories for the object. A sequential history H is *legal* if each object subhistory is legal for that object.

A method m of an object x is *total* if for every finite complete history H in the sequential specification of x and every invocation $\langle x.m(a^*)\ A\rangle$ of m, there is a response $\langle x : t(r^*)\ A\rangle$ such that $H \cdot \langle x.m(a^*)\ A\rangle \cdot \langle x : t(r^*)\ A\rangle$ is in the sequential specification of x. A method is *partial* if it is not total.

3.6.2 Linearizability

A key concept in defining linearizability is the *real-time order* of a history. Recall that a *(strict) partial order* $\rightarrow$ on a set X is a relation that is irreflexive and transitive. That is, it is never true that $x \rightarrow x$, and whenever $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

Note that there may be distinct x and y such that neither $x \rightarrow y$ nor $y \rightarrow x$. A *total order* $<$ on X is a partial order such that for all distinct x and y in X, either $x < y$ or $y < x$.

Any partial order can be extended to a total order.

Fact 3.6.1. If $\rightarrow$ is a partial order on X, then there exists a total order $<$ on X such that if $x \rightarrow y$ then $x < y$.

We say that a method call m_0 *precedes* a method call m_1 in history H if m_0 finishes before m_1 starts, that is, if m_0's response event occurs before m_1's invocation event in H. This notion is important enough to introduce some shorthand notation: Given a history H containing method calls m_0 and m_1, we write $m_0 \rightarrow_H m_1$ if m_0 precedes m_1 in H. We leave it as an exercise to show that $\rightarrow_H$ is a partial order. Note that if H is sequential, then $\rightarrow_H$ is a total order. Given a history H and an object x such that $H|x$ contains method calls m_0 and m_1, when H is clear from the context, we write $m_0 \rightarrow_x m_1$ if m_0 precedes m_1 in $H|x$.

For linearizability, the basic rule is that if one method call precedes another, then the earlier call must take effect before the later call (each call must linearize within its interval, and the interval of the earlier interval is entirely in front of the interval of the later call). By contrast, if two method calls overlap, then their order is ambiguous, and we are free to order them in any convenient way.

Definition 3.6.2. A legal sequential history S is a *linearization* of a history H if H can be extended to a history H' by appending zero or more responses such that:

L1 *complete*(H') is equivalent to S, and
L2 if method call m_0 precedes method call m_1 in H, then the same is true in S (i.e., $m_0 \rightarrow_H m_q$ implies $m_0 \rightarrow_S m_1$).

H is *linearizable* if there is a linearization of H.

Informally, extending H to H' captures the idea that some pending invocations may have taken effect, even though their responses have not yet been returned to the caller. Fig. 3.9 illustrates the notion: We must complete the pending `enq`(x) method call to justify the `deq`() call that returns x. The second condition says that if one method call precedes another in the original history, then that ordering must be preserved in the linearization.

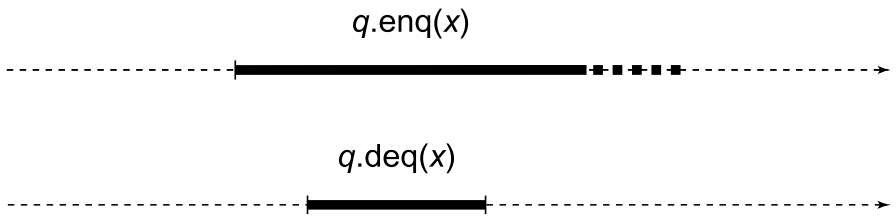

FIGURE 3.9

The pending `enq`(x) method call must take effect early to justify the `deq`() call that returns x.

3.6.3 Linearizability is compositional

Linearizability is compositional.

Theorem 3.6.3. *H is linearizable if, and only if, for each object x, $H|x$ is linearizable.*

Proof. The "only if" part is left as an exercise.

For each object x, pick a linearization of $H|x$. Let R_x be the set of responses appended to $H|x$ to construct that linearization, and let $\rightarrow_x$ be the corresponding linearization order. Let H' be the history constructed by appending to H each response in R_x (the order in which they are appended does not matter).

We argue by induction on the number of method calls in H'. For the base case, if H' contains no method calls, we are done. Otherwise, assume the claim for every H' containing fewer than $k \geq 1$ method calls. For each object x, consider the last method call in $H'|x$. One of these calls m must be maximal with respect to $\rightarrow_H$; that is, there is no m' such that $m \rightarrow_H m'$. Let G' be the history defined by removing m from H'. Because m is maximal, H' is equivalent to $G' \cdot m$. By the induction hypothesis, G' is linearizable to a sequential history S', and both H' and H are linearizable to $S' \cdot m$. □

3.6.4 Linearizability is nonblocking

Linearizability is a *nonblocking* property: A pending invocation of a total method is never required to wait for another pending invocation to complete.

Theorem 3.6.4. *If m is a total method of an object x and $\langle x.m(a^*)\ P\rangle$ is a pending invocation in a linearizable history H, then there exists a response $\langle x : t(r^*)\ P\rangle$ such that $H \cdot \langle x : t(r^*)\ P\rangle$ is linearizable.*

Proof. Let S be any linearization of H. If S includes a response $\langle x : t(r^*)\ P\rangle$ to $\langle x.m(a^*)\ P\rangle$, we are done, since S is also a linearization of $H \cdot \langle x : t(r^*)\ P\rangle$. Otherwise, $\langle x.m(a^*)\ P\rangle$ does not appear in S either, since a linearization, by definition, has no pending invocations. Because the method is total, there exists a response $\langle x : t(r^*)\ P\rangle$ such that

$$S' = S \cdot \langle x.m(a^*)\ P\rangle \cdot \langle x : t(r^*)\ P\rangle$$

is a legal sequential history. S' is a linearization of $H \cdot \langle x : t(r^*)\ P\rangle$, and hence is also a linearization of H. □

This theorem implies that linearizability by itself never forces a thread with a pending invocation of a total method to block. Of course, blocking (or even deadlock) may occur as artifacts of particular implementations of linearizability, but it is not inherent to the correctness property itself. This theorem suggests that linearizability is an appropriate correctness property for systems where concurrency and real-time response are important.

The nonblocking property does not rule out blocking in situations where it is explicitly intended. For example, it may be sensible for a thread attempting to dequeue from an empty queue to block, waiting until another thread enqueues an item. A queue specification would capture this intention by making the `deq()` method's specification partial, leaving its effect undefined when applied to an empty queue. The most natural concurrent interpretation of a partial sequential specification is simply to wait until the object reaches a state in which that method call's response is defined.

3.7 Memory consistency models

We can consider the memory read and written by a program as a single object—the composition of many registers—shared by all threads of the program. This shared memory is often the only means of communication among threads (i.e., the only way that threads can observe the effects of other threads). Its correctness property is called the *memory consistency model*, or *memory model* for short.

Early concurrent programs assumed sequentially consistent memory. Indeed, the notion of sequential consistency was introduced to capture the assumptions implicit in those programs. However, the memory of most modern multiprocessor systems is *not* sequentially consistent: Compilers and hardware may reorder memory reads and writes in complex ways. Most of the time no one can tell, because the vast majority of reads and writes are not used for synchronization. These systems also provide synchronization primitives that inhibit reordering.

We follow this approach in the first part of this book, where we focus on the principles of multiprocessor programming. For example, the pseudocode for the various lock algorithms in Chapter 2 assumes that if a thread writes two locations, one after the other, then the two writes are made visible to other threads in the same order, so that any thread that sees the later write will also see the earlier write. However, Java does not guarantee this ordering for ordinary reads and writes. As mentioned in Pragma 2.3.1, these locations would need to be declared **`volatile`** to work in real systems. We omit these declarations because these algorithms are not practical in any case, and the declarations would clutter the code and obscure the ideas embodied in those algorithms. In the second part of the book, where we discuss practical algorithms, we include these declarations. (We describe the Java memory model in Appendix A.3.)

3.8 Progress conditions

The nonblocking property of linearizability (and sequential consistency and quiescent consistency) ensures that any pending invocation has a correct response. But linearizability does not tell us how to compute such a response, nor even require an implementation to produce a response at all. Consider, for example, the lock-based queue shown in Fig. 3.1. Suppose the queue is initially empty, and thread A halts halfway through enqueuing x, while holding the lock, and B then invokes `deq()`. The nonblocking property guarantees that there is a correct response to B's call to `deq()`; indeed, there are two: It could throw an exception or return x. In this implementation, however, B is unable to acquire the lock, and will be delayed as long as A is delayed.

Such an implementation is called *blocking*, because delaying one thread can prevent others from making progress. Unexpected thread delays are common in multiprocessors. A cache miss might delay a processor for a hundred cycles, a page fault for a few million cycles, preemption by the operating system for hundreds of millions of cycles. These delays depend on the specifics of the machine and the operating system. The part of the system that determines when threads take steps is called the *scheduler*, and the order in which threads take steps is the *schedule*.

In this section, we consider *progress conditions*, which require implementations to produce responses to pending invocations. Ideally, we would like to say simply that every pending invocation gets a response. Of course, this is not possible if the threads with pending invocations stop taking steps. So we require progress only for those threads that keep taking steps.

3.8.1 Wait-freedom

A method of an object implementation is *wait-free* if every call finishes its execution in a finite number of steps; that is, if a thread with a pending invocation to a wait-free method keeps taking steps, it completes in a finite number of steps. We say that an object implementation is *wait-free* if all its methods are wait-free, and that a class is *wait-free* if every object of that class is wait-free.

The queue shown in Fig. 3.3 is wait-free. For example, if B invokes `deq()` while A is halted halfway through enqueuing x, then B will either throw `EmptyException` (if A halted before incrementing `tail`) or it will return x (if A halted afterward). In contrast, the lock-based queue is not wait-free because B may take an unbounded number of steps unsuccessfully trying to acquire the lock.

We say that wait-freedom is a *nonblocking* progress condition[2] because a wait-free implementation cannot be blocking: An arbitrary delay by one thread (say, one holding a lock) cannot prevent other threads from making progress.

3.8.2 Lock-freedom

Wait-freedom is attractive because it guarantees that every thread that takes steps makes progress. However, wait-free algorithms can be inefficient, and sometimes we are willing to settle for a weaker progress guarantee.

One way to relax the progress condition is to guarantee progress only to *some* thread, rather than *every* thread. A method of an object implementation is *lock-free* if executing the method guarantees that *some* method call finishes in a finite number of

[2] See Remark 3.3.1 for various ways in which the term *nonblocking* is used.

steps; that is, if a thread with a pending invocation to a lock-free method keeps taking steps, then within a finite number of its steps, some pending call to a method of that object (not necessarily the lock-free method) completes. An object implementation is *lock-free* if all its methods are lock-free. We say that lock-freedom guarantees *minimal progress* because executing a lock-free method guarantees that the system as a whole makes progress, but not that any thread in particular makes progress. In contrast, wait-freedom guarantees *maximal progress*: Every thread that keeps taking steps makes progress.

Clearly, any wait-free method implementation is also lock-free, but not vice versa. Although lock-freedom is weaker than wait-freedom, if a program executes only a finite number of method calls, then lock-freedom is equivalent to wait-freedom for that program.

Lock-free algorithms admit the possibility that some threads could starve. As a practical matter, there are many situations in which starvation, while possible, is extremely unlikely, so a fast lock-free algorithm may be preferable to a slower wait-free algorithm. We consider several lock-free concurrent objects in later chapters.

Lock-freedom is also a nonblocking progress condition: A delayed thread does not prevent other threads from making progress as long as the system as a whole keeps taking steps.

3.8.3 Obstruction-freedom

Another way to relax the progress condition is to guarantee progress only under certain assumptions about how threads are scheduled, that is, about the order in which threads take steps. For example, an implementation may guarantee progress only if no other threads actively interfere with it. We say that a thread executes *in isolation* in an interval if no other threads take steps in that interval. A method of an object implementation is *obstruction-free* if, from any point after which it executes in isolation, it finishes in a finite number of steps; that is, if a thread with a pending invocation to an obstruction-free method executes in isolation from any point (not necessarily from its invocation), it completes in a finite number of steps.

Like other nonblocking progress conditions, obstruction-freedom ensures that a thread cannot be blocked by the delay of other threads. Obstruction-freedom guarantees progress to every thread that executes in isolation, so like wait-freedom, it guarantees maximal progress.

By guaranteeing progress only when one thread is scheduled to execute in isolation (i.e., preventing other threads from taking steps concurrently), obstruction-freedom seems to defy most operating system schedulers, which try to ensure a schedule in which every thread keeps taking steps (such a schedule is called *fair*). In practice, however, there is no problem. Ensuring progress for an obstruction-free method does not require pausing all other threads, only those threads that *conflict*, meaning those that are executing method calls on the same object. In later chapters, we consider a variety of *contention management techniques* to reduce or eliminate conflicting concurrent method calls. The simplest such technique is to introduce a *back-off* mechanism: a thread that detects a conflict pauses to give an earlier thread time to finish. Choosing when to back off, and for how long, is discussed in detail in Chapter 7.

3.8.4 Blocking progress conditions

In Chapter 2, we defined two progress conditions for lock implementations: *deadlock-freedom* and *starvation-freedom*. Analogous to lock-freedom and wait-freedom, respectively, deadlock-freedom guarantees that some thread makes progress and starvation-freedom guarantees that every thread makes progress *provided the lock is not held by some other thread*. The caveat that the lock is not held is necessary because, while one thread holds the lock, no other thread can acquire it without violating mutual exclusion, the correctness property for locks. To guarantee progress, we must also assume that a thread holding a lock will eventually release it. This assumption has two parts: (a) Each thread that acquires the lock must release it after a finite number of steps, and (b) the scheduler must allow a thread holding the lock to keep taking steps.

We can generalize deadlock-freedom and starvation-freedom to concurrent objects by making a similar assumption for threads executing method calls. Specifically, we assume that the scheduler is fair; that is, it allows every thread with a pending method call to take steps. (The first part of the assumption must be guaranteed by the concurrent object implementation.) We say that a method of an object implementation is *starvation-free* if it completes in a finite number of steps provided that every thread with a pending method call keeps taking steps. We say that a method of an object implementation is *deadlock-free* if, whenever there is a pending call to that method and every thread with a pending method call keeps taking steps, some method call completes in a finite number of steps.

Deadlock-freedom and starvation-freedom are useful progress conditions when the operating system guarantees that every thread keeps taking steps, and particularly that each thread takes a step *in a timely manner*. We say these properties are *blocking progress conditions* because they admit blocking implementations where the delay of a single thread can prevent all other threads from making progress.

A class whose methods rely on lock-based synchronization can guarantee, at best, a blocking progress condition. Does this observation mean that lock-based algorithms should be avoided? Not necessarily. If preemption in the middle of a critical section is sufficiently rare, then blocking progress conditions may be effectively indistinguishable from their nonblocking counterparts. If preemption is common enough to cause concern, or if the cost of preemption-based delay is sufficiently high, then it is sensible to consider nonblocking progress conditions.

3.8.5 Characterizing progress conditions

We now consider the various progress conditions and how they relate to one another. For example, wait-freedom and lock-freedom guarantee progress no matter how threads are scheduled. We say that they are *independent* progress conditions.

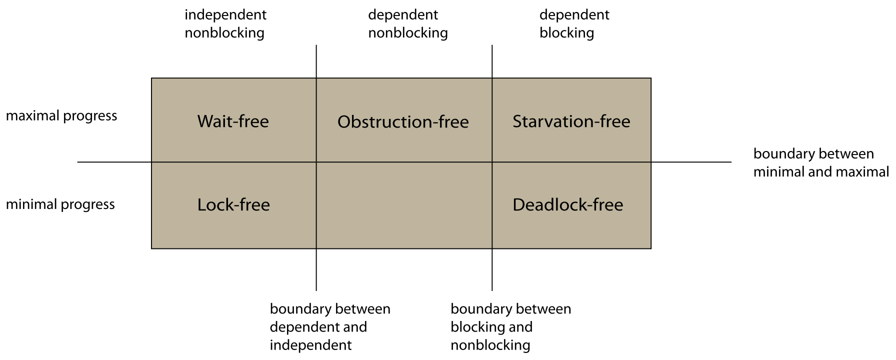

FIGURE 3.10

Progress conditions and their properties.

By contrast, obstruction-freedom, starvation-freedom, and deadlock-freedom are all *dependent* progress conditions, where progress is guaranteed only if the underlying operating system satisfies certain properties: fair scheduling for starvation-freedom and deadlock-freedom, isolated execution for obstruction-freedom. Also, as we already discussed, wait-freedom, lock-freedom, and obstruction-freedom are all nonblocking progress conditions, whereas starvation-freedom and deadlock-freedom are blocking.

We can also characterize these progress conditions by whether they guarantee maximal or minimal progress under their respective system assumptions: Wait-freedom, starvation-freedom, and obstruction-freedom guarantee maximal progress, whereas lock-freedom and deadlock-freedom guarantee only minimal progress.

Fig. 3.10 summarizes this discussion with a table that shows the various progress conditions and their properties. There is a “hole” in this table because any condition that guarantees minimal progress to threads that execute in isolation also guarantees maximal progress to these threads.

Picking a progress condition for a concurrent object implementation depends on both the needs of the application and the characteristics of the underlying platform. Wait-freedom and lock-freedom have strong theoretical properties, they work on just about any platform, and they provide guarantees useful to real-time applications such as music, electronic games, and other interactive applications. The dependent obstruction-free, deadlock-free, and starvation-free properties rely on guarantees provided by the underlying platform. Given those guarantees, however, the dependent properties often admit simpler and more efficient implementations.

3.9 Remarks

Which correctness condition is right for your application? It depends on your application’s needs. A lightly loaded printer server might be satisfied with a quiescently consistent queue of jobs, since the order in which documents are printed is of little importance. A banking server that must execute customer requests in program order (e.g., transfer $100 from savings to checking, then write a check for $50), might require a sequentially consistent queue. A stock trading server that is required to be fair, ensuring that orders from different customers must be executed in the order they arrive, would require a linearizable queue.

Which progress condition is right for your application? Again, it depends on the application's needs. In a way, this is a trick question. Different methods, even ones for the same object, might have different progress conditions. For example, the table lookup method of a firewall program, which checks whether a packet source is suspect, is called frequently and is time-critical, so we might want it to be wait-free. By contrast, the method that updates table entries, which is rarely called, might be implemented using mutual exclusion. As we shall see, it is quite natural to write applications whose methods differ in their progress guarantees.

So what progress condition is right for a particular operation? Programmers typically intend any operation they execute to eventually complete. That is, they want maximal progress. However, ensuring progress requires assumptions about the underlying platform. For example, how does the operating system schedule threads for execution? The choice of progress condition reflects what the programmer is willing to assume to guarantee that an operation will complete. For any progress guarantee, the programmer must assume that the thread executing the operation is eventually scheduled. For certain critical operations, the programmer may be unwilling to assume more than that, incurring extra overhead to ensure progress. For other operations, stronger assumptions, such as fairness or a particular priority scheme for scheduling, may be acceptable, enabling less expensive solutions.

The following joke circulated in Italy in the 1920s: According to Mussolini, the ideal citizen is intelligent, honest, and fascist. Unfortunately, no one is perfect, which explains why everyone you meet is either intelligent and fascist but not honest, honest and fascist but not intelligent, or honest and intelligent but not fascist.

As programmers, it would be ideal to have linearizable hardware, linearizable data structures, and good performance. Unfortunately, technology is imperfect, and for the time being, hardware that performs well is usually not even sequentially consistent. As the joke goes, that leaves open the possibility that data structures might still be linearizable while performing well. Nevertheless, there are many challenges to make this vision work, and the remainder of this book presents a road map toward attaining this goal.

3.10 Chapter notes

Leslie Lamport [102] introduced the notion of *sequential consistency*, while Christos Papadimitriou [137] formulated the canonical formal characterization of *serializability*. William Weihl [166] was the first to point out the importance of *compositionality* (which he called *locality*). Maurice Herlihy and Jeannette Wing [75] introduced the notion of *linearizability*. *Quiescent consistency* was introduced implicitly by James Aspnes, Maurice Herlihy, and Nir Shavit [14], and more explicitly by Nir Shavit and Asaph Zemach [158]. Leslie Lamport [99,105] introduced the notion of an *atomic register*.

The two-thread queue is considered folklore; as far as we are aware, it first appeared in print in a paper by Leslie Lamport [103].

To the best of our knowledge, the notion of *wait-freedom* first appeared implicitly in Leslie Lamport's Bakery algorithm [100]. *Lock-freedom* has had several historical meanings and only in recent years has it converged to its current definition. The notions of *dependent progress* and of *minimal* and *maximal progress* and the table of progress conditions were introduced by Maurice Herlihy and Nir Shavit [72]. *Obstruction-freedom* was introduced by Maurice Herlihy, Victor Luchangco, and Mark Moir [68].

3.11 Exercises

Exercise 3.1. Explain why quiescent consistency is compositional.

Exercise 3.2. Consider a *memory object* that encompasses two register components. We know that if both registers are quiescently consistent, then so is the memory. Does the converse hold? That is, if the memory is quiescently consistent, are the individual registers quiescently consistent? Outline a proof, or give a counterexample.

Exercise 3.3. Give an example of an execution that is quiescently consistent but not sequentially consistent, and another that is sequentially consistent but not quiescently consistent.

Exercise 3.4. For each of the histories shown in Figs. 3.11 and 3.12, are they quiescently consistent? Sequentially consistent? Linearizable? Justify your answer.

Exercise 3.5. If we drop condition L2 from the linearizability definition, is the resulting property the same as sequential consistency? Explain.

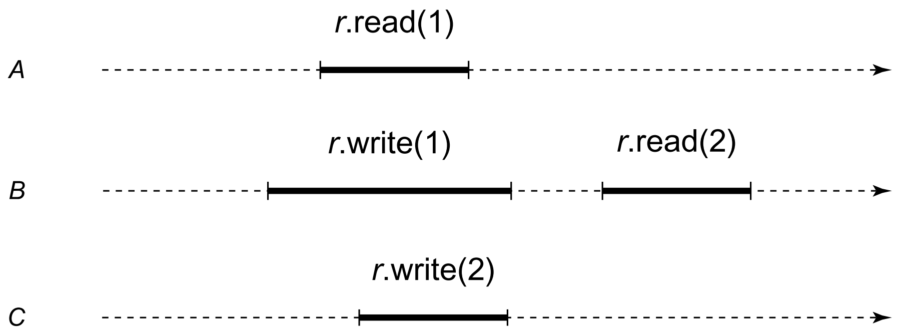

FIGURE 3.11

First history for Exercise 3.4.

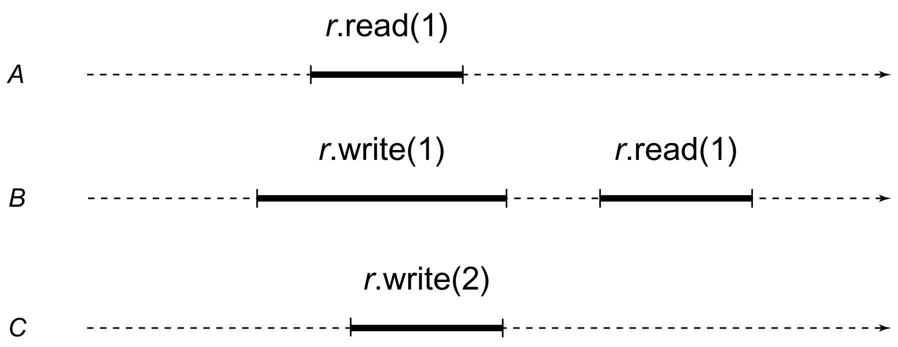

FIGURE 3.12

Second history for Exercise 3.4.

Exercise 3.6. Prove the "only if" part of Theorem 3.6.3.

Exercise 3.7. The `AtomicInteger` class (in the java.util.concurrent.atomic package) is a container for an integer value. One of its methods is

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

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

```
int get()
```

which returns the object's value.

Consider the FIFO queue implementation shown in Fig. 3.13. It stores its items in an array `items`, which, for simplicity, we assume has unbounded size. It has two `AtomicInteger` fields: `head` is the index of the next slot from which to remove an item, and `tail` is the index of the next slot in which to place an item. Give an example showing that this implementation is *not* linearizable.

Exercise 3.8. Consider the following rather unusual implementation of a method m: In every history, the i-th time a thread calls m, the call returns after 2^i steps. Is this method wait-free?

Exercise 3.9. Consider a system with an object x and n threads. Determine if each of the following properties are equivalent to saying x is deadlock-free, starvation-free, obstruction-free, lock-free, wait-free, or none of these. Briefly justify your answers.

1. For every infinite history H of x, an infinite number of method calls complete.
2. For every finite history H of x, there is an infinite history $H' = H \cdot G$.
3. For every infinite history H of x, every thread takes an infinite number of steps.
4. For every infinite history H of x, every thread that takes an infinite number of steps in H completes an infinite number of method calls.

```
class IQueue<T> {
  AtomicInteger head = new AtomicInteger(0);
  AtomicInteger tail = new AtomicInteger(0);
  T[] items = (T[]) new Object[Integer.MAX_VALUE];
  public void enq(T x) {
    int slot;
    do {
      slot = tail.get();
    } while (!tail.compareAndSet(slot, slot+1));
    items[slot] = x;
  }
  public T deq() throws EmptyException {
    T value;
    int slot;
    do {
      slot = head.get();
      value = items[slot];
      if (value == null)
        throw new EmptyException();
    } while (!head.compareAndSet(slot, slot+1));
    return value;
  }
}
```

FIGURE 3.13

`IQueue` implementation for Exercise 3.7.

5. For every finite history H of x, there are n infinite histories $H'_i = H \cdot G_i$ where only thread i takes steps in G_i, where it completes an infinite number of method calls.
6. For every finite history H of x, there is an infinite history $H' = H \cdot G$ where every thread completes an infinite number of method calls in G.

Exercise 3.10. This exercise examines the queue implementation in Fig. 3.14, whose `enq`() method does not have a single fixed linearization point in the code.

The queue stores its items in an `items` array, which, for simplicity, we assume is unbounded. The `tail` field is an `AtomicInteger`, initially zero.

The `enq`() method reserves a slot by incrementing `tail`, and then stores the item at that location. Note that these two steps are not atomic: There is an interval after `tail` has been incremented but before the item has been stored in the array.

The `deq`() method reads the value of `tail`, and then traverses the array in ascending order from slot zero to the tail. For each slot, it swaps *null* with the current contents, returning the first non-*null* item it finds. If all slots are *null*, the procedure is restarted.

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

FIGURE 3.14

Herlihy–Wing queue for Exercise 3.10.

- Give an execution showing that the linearization point for `enq()` cannot occur at line 15. (Hint: Give an execution in which two `enq()` calls are not linearized in the order they execute line 15.)
- Give another execution showing that the linearization point for `enq()` cannot occur at line 16.
- Since these are the only two memory accesses in `enq()`, we must conclude that `enq()` has no single linearization point. Does this mean `enq()` is not linearizable?

Exercise 3.11. This exercise examines a stack implementation (Fig. 3.15) whose `push()` method does not have a single fixed linearization point in the code.

The stack stores its items in an `items` array, which, for simplicity, we assume is unbounded. The `top` field is an `AtomicInteger`, initially zero.

```
 1  public class AGMStack<T> {
 2    AtomicReferenceArray<T> items;
 3    AtomicInteger top;
 4    static final int CAPACITY = Integer.MAX_VALUE;
 5
 6    public AGMStack() {
 7      items = new AtomicReferenceArray<T>(CAPACITY);
 8      top = new AtomicInteger(0);
 9    }
10    public void push(T x) {
11      int i = top.getAndIncrement();
12      items.set(i,x);
13    }
14    public T pop() {
15      int range = top.get();
16      for (int i = range - 1; i > -1; i--) {
17        T value = items.getAndSet(i, null);
18        if (value != null) {
19          return value;
20        }
21      }
22      // Return Empty.
23      return null;
24    }
25  }
```

FIGURE 3.15

Afek–Gafni–Morrison stack for Exercise 3.11.

The push() method reserves a slot by incrementing top, and then stores the item at that location. Note that these two steps are not atomic: There is an interval after top has been incremented but before the item has been stored in the array.

The pop() method reads the value of top and then traverses the array in descending order from the top to slot zero. For each slot, it swaps *null* with the current contents, returning the first *nonnull* item it finds. If all slots are *null*, the method returns *null*, indicating an empty stack.

- Give an execution showing that the linearization point for push() cannot occur at line 11. (Hint: Give an execution in which two push() calls are not linearized in the order they execute line 11.)
- Give another execution showing that the linearization point for push() cannot occur at line 12.
- Since these are the only two memory accesses in push(), we conclude that push() has no single fixed linearization point. Does this mean push() is not linearizable?

Exercise 3.12. Prove that sequential consistency is nonblocking.