CHAPTER

# Monitors and blocking synchronization

8

## 8.1 Introduction

A *monitor* is a structured way of combining synchronization and data, encapsulating data, methods, and synchronization in a single modular package in the same way that a class encapsulates data and methods.

Here is why modular synchronization is important: Imagine an application with two threads, a producer and a consumer, that communicate through a shared FIFO queue. The threads might share two objects: an unsynchronized queue and a lock to protect the queue. The producer might look something like this:

```
mutex.lock();
try {
  queue.enq(x)
} finally {
  mutex.unlock();
}
```

This is no way to run a railroad! Suppose the queue is bounded, meaning that an attempt to add an item to a full queue cannot proceed until the queue has room. Here, the decision whether to block the call or to let it proceed depends on the queue's internal state, which is (and should be) inaccessible to the caller. Moreover, suppose the application grows to have multiple producers, consumers, or both. Each such thread must keep track of both the lock and the queue objects, and the application will be correct only if every thread follows the same locking conventions.

A more sensible approach is to allow each queue to manage its own synchronization. The queue itself has its own internal lock, acquired by each method when it is called and released when it returns. There is no need to ensure that every thread that uses the queue follows a cumbersome synchronization protocol. If a thread tries to enqueue an item to a queue that is already full, then the `enq()` method itself can detect the problem, suspend the caller, and resume the caller when the queue has room.

## 8.2 Monitor locks and conditions

As in Chapters 2 and 7, a lock is the basic mechanism for ensuring mutual exclusion. Only one thread at a time can *hold* a lock. A thread *acquires* a lock when it first starts

**The Art of Multiprocessor Programming.** https://doi.org/10.1016/B978-0-12-415950-1.00018-5

to hold the lock. A thread *releases* a lock when it stops holding the lock. A monitor exports a collection of methods, each of which acquires the lock when it is called, and releases it when it returns.

If a thread must wait for some condition to hold, it can either *spin*, repeatedly testing for the desired condition, or *block*, giving up the processor for a while to allow another thread to run.[1] Spinning makes sense on a multiprocessor if we expect to wait for a short time, because blocking a thread requires an expensive call to the operating system. On the other hand, blocking makes sense if we expect to wait for a long time, because a spinning thread keeps a processor busy without doing any work.

For example, a thread waiting for another thread to release a lock should spin if that particular lock is held briefly, while a consumer thread waiting to dequeue an item from an empty buffer should block, since there is usually no way to predict how long it may have to wait. Often, it makes sense to combine spinning and blocking: A thread waiting to dequeue an item might spin for a brief duration, and then switch to blocking if the delay appears to be long. Blocking works on both multiprocessors and uniprocessors, while spinning works only on multiprocessors.

Most of the locks in this book follow the interface shown in Fig. 8.1. Here is a description of the `Lock` interface's methods:

- The `lock()` method blocks the caller until it acquires the lock.
- The `lockInterruptibly()` method acts like `lock()`, but throws an exception if the thread is interrupted while it is waiting (see Pragma 8.2.1).
- The `unlock()` method releases the lock.
- The `newCondition()` method is a *factory* that creates and returns a `Condition` object associated with the lock (explained in Section 8.2.1).
- The `tryLock()` method acquires the lock if it is free, and immediately returns a Boolean indicating whether it acquired the lock. This method can also be called with a timeout.

```
1  public interface Lock {
2    void lock();
3    void lockInterruptibly() throws InterruptedException;
4    boolean tryLock();
5    boolean tryLock(long time, TimeUnit unit);
6    Condition newCondition();
7    void unlock();
8  }
```

**FIGURE 8.1**

The `Lock` interface.

[1] In Chapter 3, we make a distinction between blocking and nonblocking synchronization algorithms. There, we mean something entirely different: A blocking algorithm is one where a delay by one thread can cause a delay in another. Remark 3.3.1 discusses various ways in which the term *blocking* is used.

### 8.2.1 Conditions

While a thread is waiting for something to happen, say, for another thread to place an item in a queue, it must release the lock on the queue; otherwise, the other thread will never be able to enqueue the anticipated item. After the waiting thread has released the lock, we need a way to be notify it when to reacquire the lock and try again.

In the java.util.concurrent package (and in similar packages such as Pthreads), the ability to release a lock temporarily is provided by a `Condition` object associated with a lock. (Conditions are often called *condition variables* in the literature.) Fig. 8.2 shows how to use the `Condition` interface provided in the java.util.concurrent.locks library. A condition is associated with a lock, and is created by calling that lock's `newCondition()` method. If the thread holding that lock calls the associated condition's `await()` method, it releases that lock and suspends itself, giving another thread the opportunity to acquire the lock. When the calling thread awakens, it reacquires the lock, perhaps competing with other threads.

Like locks, `Condition` objects must be used in a stylized way. Suppose a thread wants to wait until a certain property holds. The thread tests the property while holding the lock. If the property does not hold, then the thread calls `await()` to release the lock and sleep until it is awakened by another thread. Here is the key point: There is no guarantee that the property will hold when the thread awakens. The `await()` method can return spuriously (i.e., for no reason), or the thread that signaled the condition may have awakened too many sleeping threads. Whatever the reason, the thread must retest the property, and if it finds the property does not hold at that time, it must call `await()` again.

The `Condition` interface in Fig. 8.3 provides several variations of this call, some of which provide the ability to specify a maximum time the caller can be suspended, or whether the thread can be interrupted while it is waiting. When the queue changes, the thread that made the change can notify other threads waiting on a condition. Calling `signal()` wakes up one thread waiting on a condition (if there is one), while calling `signalAll()` wakes up all waiting threads.

```
1  Condition condition = mutex.newCondition();
2  ...
3  mutex.lock()
4  try {
5    while (!property) { // not happy
6      condition.await(); // wait for property
7    } catch (InterruptedException e) {
8      ... // application-dependent response
9    }
10   ... // happy: property must hold
11 }
```

**FIGURE 8.2**

How to use `Condition` objects.

```
1  public interface Condition {
2    void await() throws InterruptedException;
3    boolean await(long time, TimeUnit unit) throws InterruptedException;
4    boolean awaitUntil(Date deadline) throws InterruptedException;
5    long awaitNanos(long nanosTimeout) throws InterruptedException;
6    void awaitUninterruptibly();
7    void signal();      // wake up one waiting thread
8    void signalAll();   // wake up all waiting threads
9  }
```

**FIGURE 8.3**

The `Condition` interface: `await()` and its variants release the lock, and give up the processor, and then later awaken and reacquire the lock. The `signal()` and `signalAll()` methods awaken one or more waiting threads.

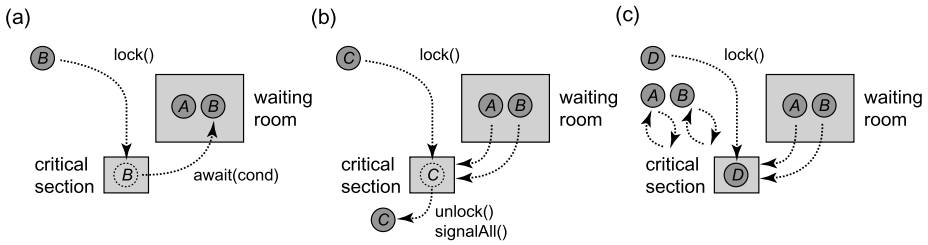


**FIGURE 8.4**

A schematic representation of a monitor execution. In part (a), thread *A* has acquired the monitor lock, called `await()` on a condition, released the lock, and is now in the waiting room. Thread *B* then goes through the same sequence of steps, entering the critical section, calling `await()` on the condition, relinquishing the lock, and entering the waiting room. In part (b), both *A* and *B* leave the waiting room after thread *C* exits the critical section and calls `signalAll()`. *A* and *B* then attempt to reacquire the monitor lock. However, thread *D* manages to acquire the critical section lock first, and so both *A* and *B* spin until *D* leaves the critical section. Note that if *C* had issued a `signal()` instead of a `signalAll()`, only *A* or *B* would have left the waiting room, and the other would have continued to wait.

**PRAGMA 8.2.1**

Threads in Java can be *interrupted* by other threads. If a thread is interrupted during a call to a `Condition`'s `await()` method, the call throws `InterruptedException`. The proper response to an interrupt is application-dependent. Fig. 8.2 shows a schematic example.

To avoid clutter, we usually omit `InterruptedException` handlers from example code, even though they would be required in actual code. (It is bad programming practice to ignore interrupts.)

This combination of methods, mutual exclusion locks, and condition objects is called a *monitor*. It is common to talk of threads that have called `await()` (and have not yet returned) as being in a "waiting room". We use this imagery to illustrate an execution of a monitor in Fig. 8.4.

Fig. 8.5 shows how to implement a bounded FIFO queue using explicit locks and conditions. The `lock` field is a lock that must be acquired by all methods. We must initialize it to hold an instance of a class that implements the `Lock` interface. Here, we choose `ReentrantLock`, a useful lock type provided by the `java.util.concurrent.locks` package. This lock is *reentrant*: A thread that is holding the lock can acquire it again without blocking. (See Section 8.4 for more discussion on reentrant locks.)

There are two condition objects: `notEmpty` notifies waiting dequeuers when the queue goes from being empty to nonempty, and `notFull` for the opposite direction. Although using two conditions instead of one is more complex, it is more efficient, since fewer threads are woken up unnecessarily.

### 8.2.2 The lost-wakeup problem

Just as locks are inherently vulnerable to deadlock, `Condition` objects are inherently vulnerable to *lost wakeups*, in which one or more threads wait forever without realizing that the condition for which they are waiting has become true.

Lost wakeups can occur in subtle ways. Fig. 8.6 shows an ill-considered optimization of the `Queue<T>` class. Instead of signaling the `notEmpty` condition each time `enq()` enqueues an item, would it not be more efficient to signal the condition only when the queue actually transitions from empty to nonempty? This optimization works as intended if there is only one producer and one consumer, but it is incorrect if there are multiple producers or consumers. Consider the following scenario: Consumers $A$ and $B$ both try to dequeue an item from an empty queue, both detect the queue is empty, and both block on the `notEmpty` condition. Producer $C$ enqueues an item in the buffer, and signals `notEmpty`, waking $A$. Before $A$ can acquire the lock, however, another producer $D$ puts a second item in the queue, and because the queue is not empty, it does not signal `notEmpty`. Then $A$ acquires the lock and removes the first item, but $B$, victim of a lost wakeup, waits forever, even though there is an item in the queue to be consumed.

Although there is no substitute for reasoning carefully about our program, there are simple programming practices that minimize vulnerability to lost wakeups.

- Always signal *all* processes waiting on a condition, not just one.
- Specify a timeout when waiting.

Either of these two practices would fix the bounded queue error we just described. Each has a small performance penalty, but negligible compared to the cost of a lost wakeup.

Java provides support for monitors in the form of **`synchronized`** blocks and methods, and built-in `wait()`, `notify()`, and `notifyAll()` methods (see Appendix A).

```
1  class LockedQueue<T> {
2    final Lock lock = new ReentrantLock();
3    final Condition notFull = lock.newCondition();
4    final Condition notEmpty = lock.newCondition();
5    final T[] items;
6    int tail, head, count;
7    public LockedQueue(int capacity) {
8      items = (T[])new Object[capacity];
9    }
10   public void enq(T x) {
11     lock.lock();
12     try {
13       while (count == items.length)
14         notFull.await();
15       items[tail] = x;
16       if (++tail == items.length)
17         tail = 0;
18       ++count;
19       notEmpty.signal();
20     } finally {
21       lock.unlock();
22     }
23   }
24   public T deq() {
25     lock.lock();
26     try {
27       while (count == 0)
28         notEmpty.await();
29       T x = items[head];
30       if (++head == items.length)
31         head = 0;
32       --count;
33       notFull.signal();
34       return x;
35     } finally {
36       lock.unlock();
37     }
38   }
39 }
```

**FIGURE 8.5**

The `LockedQueue` class: a FIFO queue using locks and conditions. There are two condition fields, one to detect when the queue becomes nonempty, and one to detect when it becomes nonfull.

```
1  public void enq(T x) {
2    lock.lock();
3    try {
4      while (count == items.length)
5        notFull.await();
6      items[tail] = x;
7      if (++tail == items.length)
8        tail = 0;
9      ++count;
10     if (count == 1) {  // Wrong!
11       notEmpty.signal();
12     }
13   } finally {
14     lock.unlock();
15   }
16 }
```

**FIGURE 8.6**

This example is *incorrect*. It suffers from lost wakeups. The `enq()` method signals `notEmpty` only if it is the first to place an item in an empty buffer. A lost wakeup occurs if multiple consumers are waiting, but only the first is awakened to consume an item.

## 8.3 Readers–writers locks

Many shared objects have the property that most method calls return information about the object's state without modifying the object, and relatively few calls actually modify the object. We call method calls of the first kind *readers*, and method calls of the latter kind *writers*.

Readers need not synchronize with one another; it is perfectly safe for them to access the object concurrently. Writers, on the other hand, must lock out readers as well as other writers. A *readers–writers lock* allows multiple readers or a single writer to enter the critical section concurrently. We use the following interface:

```
public interface ReadWriteLock {
  Lock readLock();
  Lock writeLock();
}
```

This interface exports two lock objects: the *read lock* and the *write lock*. They satisfy the following safety properties:

- No thread can acquire the write lock while any thread holds either the write lock or the read lock.
- No thread can acquire the read lock while any thread holds the write lock.

Naturally, multiple threads may hold the read lock at the same time.

We now consider two readers–writers lock implementations.

```
1  public class SimpleReadWriteLock implements ReadWriteLock {
2    int readers;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public SimpleReadWriteLock() {
8      writer = false;
9      readers = 0;
10     lock = new ReentrantLock();
11     readLock = new ReadLock();
12     writeLock = new WriteLock();
13     condition = lock.newCondition();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
21   ...
22 }
```

**FIGURE 8.7**

The `SimpleReadWriteLock` class: fields and public methods.

### 8.3.1 Simple readers–writers lock

The `SimpleReadWriteLock` class appears in Figs. 8.7 and 8.8. To define the associated read and write locks, this code uses *inner classes*, a Java feature that allows an object to create other objects that can access the first object's private fields. The `SimpleReadWriteLock` object has fields that keep track of the number of readers that hold the lock and whether a writer holds the lock; the read lock and write lock use these fields to guarantee the readers–writers lock properties. To allow the methods of the read lock and the write lock to synchronize access to these fields, the class also maintains a private `lock` and a `condition` associated with that lock.

How are waiting writers notified when the last reader releases its lock? When a writer tries to acquire the write lock, it acquires `lock` (i.e., the `SimpleReadWriteLock` object's private lock), and if any readers (or another writer) hold the lock, it waits on `condition`. A reader releasing the read lock also acquires `lock`, and signals `condition` if all readers have released their locks. Similarly, readers that try to acquire the lock while a writer holds it wait on `condition`, and writers releasing the lock signal `condition` to notify waiting readers and writers.

Although the `SimpleReadWriteLock` algorithm is correct, it is not quite satisfactory. If readers are much more frequent than writers, as is usually the case, then writers could be locked out indefinitely by a continual stream of readers.

```
23    class ReadLock implements Lock {
24      public void lock() {
25        lock.lock();
26        try {
27          while (writer)
28            condition.await();
29          readers++;
30        } finally {
31          lock.unlock();
32        }
33      }
34      public void unlock() {
35        lock.lock();
36        try {
37          readers--;
38          if (readers == 0)
39            condition.signalAll();
40        } finally {
41          lock.unlock();
42        }
43      }
44    }
45    protected class WriteLock implements Lock {
46      public void lock() {
47        lock.lock();
48        try {
49          while (readers > 0 || writer)
50            condition.await();
51          writer = true;
52        } finally {
53          lock.unlock();
54        }
55      }
56      public void unlock() {
57        lock.lock();
58        try {
59          writer = false;
60          condition.signalAll();
61        } finally {
62          lock.unlock();
63        }
64      }
65    }
```

**FIGURE 8.8**

The `SimpleReadWriteLock` class: the inner read and write locks classes.

```
1  public class FifoReadWriteLock implements ReadWriteLock {
2    int readAcquires, readReleases;
3    boolean writer;
4    Lock lock;
5    Condition condition;
6    Lock readLock, writeLock;
7    public FifoReadWriteLock() {
8      readAcquires = readReleases = 0;
9      writer = false;
10     lock = new ReentrantLock();
11     condition = lock.newCondition();
12     readLock = new ReadLock();
13     writeLock = new WriteLock();
14   }
15   public Lock readLock() {
16     return readLock;
17   }
18   public Lock writeLock() {
19     return writeLock;
20   }
21   ...
22 }
```

**FIGURE 8.9**

The `FifoReadWriteLock` class: fields and public methods.

### 8.3.2 Fair readers–writers lock

The `FifoReadWriteLock` class (Figs. 8.9 and 8.10) shows one way to prevent writers from being starved by a continual stream of readers. This class ensures that once a writer calls the write lock's `lock()` method, no more readers will acquire the read lock until the writer has acquired and released the write lock. Eventually, the readers holding the read lock will drain out without letting any more readers in, and the writer can acquire the write lock.

The `readAcquires` field counts the total number of read-lock acquisitions, and the `readReleases` field counts the total number of read-lock releases. When these quantities match, no thread is holding the read lock. (For simplicity, we are ignoring potential integer overflow and wraparound problems.) As in the `SimpleReadWriteLock` class, the `FifoReadWriteLock` class has private `lock` and `condition` fields that the methods of the read lock and write lock use to synchronize accesses to the other fields of `FifoReadWriteLock`. The difference is that in `FifoReadWriteLock`, a thread attempting to acquire the writer lock sets the `writer` flag even if readers hold the lock. If a writer holds the lock, however, it waits for the writer to release the lock, and unset the `writer` flag, before proceeding. That is, the thread first waits until no writer holds the lock, then sets `writer`, and then waits until no reader holds the lock (lines 49–53).

```
23    private class ReadLock implements Lock {
24      public void lock() {
25        lock.lock();
26        try {
27          while (writer)
28            condition.await();
29          readAcquires++;
30        } finally {
31          lock.unlock();
32        }
33      }
34      public void unlock() {
35        lock.lock();
36        try {
37          readReleases++;
38          if (readAcquires == readReleases)
39            condition.signalAll();
40        } finally {
41          lock.unlock();
42        }
43      }
44    }
45    private class WriteLock implements Lock {
46      public void lock() {
47        lock.lock();
48        try {
49          while (writer)
50            condition.await();
51          writer = true;
52          while (readAcquires != readReleases)
53            condition.await();
54        } finally {
55          lock.unlock();
56        }
57      }
58      public void unlock() {
59        lock.lock();
60        try {
61          writer = false;
62          condition.signalAll();
63        } finally {
64          lock.unlock();
65        }
66      }
67    }
```

**FIGURE 8.10**

The `FifoReadWriteLock` class: inner read and write lock classes.

## 8.4 Our own reentrant lock

Using the locks described in Chapters 2 and 7, a thread that attempts to reacquire a lock it already holds will deadlock with itself. This situation can arise if a method that acquires a lock makes a nested call to another method that acquires the same lock.

A lock is *reentrant* if it can be acquired multiple times by the same thread. We now examine how to create a reentrant lock from a nonreentrant lock. This exercise is intended to illustrate how to use locks and conditions. The java.util.concurrent.locks package provides reentrant lock classes, so in practice there is no need to write our own.

Fig. 8.11 shows the `SimpleReentrantLock` class. The `owner` field holds the ID of the last thread to acquire the lock, and the `holdCount` field is incremented each time the lock is acquired, and decremented each time it is released. The lock is free when the `holdCount` value is zero. Because these two fields are manipulated atomically, we need an internal, short-term lock. The `lock` field is a lock used by `lock()` and `unlock()` to manipulate the fields, and the `condition` field is used by threads waiting for the lock to become free. We initialize the internal `lock` field to an object of a (fictitious) `SimpleLock` class, which is presumably not reentrant (line 6).

The `lock()` method acquires the internal lock (line 13). If the current thread is already the owner, it increments the hold count and returns (line 15). Otherwise, if the hold count is not zero, the lock is held by another thread, and the caller releases the internal lock and waits until the condition is signaled (line 20). When the caller awakens, it must still check whether the hold count is zero. If it is, the calling thread makes itself the owner and sets the hold count to 1.

The `unlock()` method acquires the internal lock (line 29). It throws an exception if either the lock is free, or the caller is not the owner (line 31). Otherwise, it decrements the hold count. If the hold count is zero, then the lock is free, so the caller signals the condition to wake up a waiting thread (line 35).

## 8.5 Semaphores

As we have seen, a mutual exclusion lock guarantees that only one thread at a time can enter a critical section. If another thread wants to enter the critical section while it is occupied, then it blocks, suspending itself until another thread notifies it to try again. One of the earliest forms of synchronization, a *semaphore* is a generalization of the mutual exclusion lock. Each semaphore has a *capacity* that is determined when the semaphore is initialized. Instead of allowing only one thread at a time into the critical section, a semaphore allows at most $c$ threads, where $c$ is its capacity.

The `Semaphore` class of Fig. 8.12 provides two methods: A thread calls `acquire()` to request permission to enter the critical section, and `release()` to announce that it is leaving the critical section. The `Semaphore` itself is just a counter: It keeps track of the number of threads that have been granted permission to enter. If a new `acquire()` call is about to exceed the capacity, the calling thread is suspended until there is room.

```
1  public class SimpleReentrantLock implements Lock{
2    Lock lock;
3    Condition condition;
4    int owner, holdCount;
5    public SimpleReentrantLock() {
6      lock = new SimpleLock();
7      condition = lock.newCondition();
8      owner = 0;
9      holdCount = 0;
10   }
11   public void lock() {
12     int me = ThreadID.get();
13     lock.lock();
14     try {
15       if (owner == me) {
16         holdCount++;
17         return;
18       }
19       while (holdCount != 0) {
20        condition.await();
21       }
22       owner = me;
23       holdCount = 1;
24    } finally {
25      lock.unlock();
26    }
27   }
28   public void unlock() {
29     lock.lock();
30     try {
31       if (holdCount == 0 || owner != ThreadID.get())
32         throw new IllegalMonitorStateException();
33       holdCount--;
34       if (holdCount == 0) {
35         condition.signal();
36       }
37     } finally {
38       lock.unlock();
39     }
40   }
41   ...
42 }
```

**FIGURE 8.11**

The `SimpleReentrantLock` class: `lock()` and `unlock()` methods.

```
1  public class Semaphore {
2    final int capacity;
3    int state;
4    Lock lock;
5    Condition condition;
6    public Semaphore(int c) {
7      capacity = c;
8      state = 0;
9      lock = new ReentrantLock();
10     condition = lock.newCondition();
11   }
12   public void acquire() {
13     lock.lock();
14     try {
15       while (state == capacity) {
16         condition.await();
17       }
18       state++;
19     } finally {
20       lock.unlock();
21     }
22   }
23   public void release() {
24     lock.lock();
25     try {
26       state--;
27       condition.signalAll();
28     } finally {
29       lock.unlock();
30     }
31   }
32 }
```

**FIGURE 8.12**

Semaphore implementation.

When a thread calls `release()` after leaving the critical section, it signals to notify any waiting thread that there is now room.

## 8.6 Chapter notes

Monitors were invented by Per Brinch-Hansen [57] and Tony Hoare [77]. Semaphores were invented by Edsger Dijkstra [38]. McKenney [122] surveys different kinds of locking protocols.

## 8.7 Exercises

**Exercise 8.1.** Reimplement the `SimpleReadWriteLock` class using Java **`synchronized`**, `wait()`, `notify()`, and `notifyAll()` constructs in place of explicit locks and conditions.

Hint: You must figure out how methods of the inner read and write lock classes can lock the outer `SimpleReadWriteLock` object.

**Exercise 8.2.** Design a "nested" readers–writers lock in which a thread must first grab the read lock in order to grab the write lock, and releasing the write lock does not release the read lock. In order for a reader to become a writer with exclusive write access, every other reader must either unlock the read lock or also attempt to lock the write lock. Show that your implementation is correct and has a reasonable fairness guarantee between readers and writers.

**Exercise 8.3.** Read–write locks are fundamentally asymmetric in that many readers can enter at once but only one writer can enter. Design a symmetric locking protocol for two types of threads: RED and BLUE. For correctness, never allow a RED and BLUE thread to enter simultaneously. For progress, allow for multiple RED threads or multiple BLUE threads to enter at once, and have a symmetric fairness mechanism for draining RED threads to allow waiting BLUE threads to enter, and vice versa. Show that your implementation is correct, and describe the exact fairness property it guarantees and why you chose to use it.

**Exercise 8.4.** The `ReentrantReadWriteLock` class provided by the java.util.concurrent.locks package does not allow a thread holding the lock in read mode to then access that lock in write mode (the thread will block). Justify this design decision by sketching what it would take to permit such lock upgrades.

**Exercise 8.5.** A *savings account* object holds a nonnegative balance, and provides `deposit(`$k$`)` and `withdraw(`$k$`)` methods, where `deposit(`$k$`)` adds $k$ to the balance, and `withdraw(`$k$`)` subtracts $k$, if the balance is at least $k$, and otherwise blocks until the balance becomes $k$ or greater.

1. Implement this savings account using locks and conditions.
2. Now suppose there are two kinds of withdrawals: *ordinary* and *preferred*. Devise an implementation that ensures that no ordinary withdrawal occurs if there is a preferred withdrawal waiting to occur.
3. Now add a `transfer()` method that transfers a sum from one account to another:

```
void transfer(int k, Account reserve) {
  lock.lock();
  try {
    reserve.withdraw(k);
    deposit(k);
  } finally {
    lock.unlock();
  }
}
```