8 Monitors and Blocking Synchronization

8.1 Introduction

Monitors are a structured way of combining synchronization and data. A class encapsulates both data and methods in the same way that a monitor combines data, methods, and synchronization in a single modular package.

Here is why modular synchronization is important. Let us imagine our application has two threads, a producer and a consumer, that communicate through a shared FIFO queue. We could have the threads share two objects: an unsynchronized queue, and a lock to protect the queue. The producer looks 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. Even worse, 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 each 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

Just 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 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 cannot immediately acquire a lock, it can either *spin*, repeatedly testing whether the desired event has happened, or it can *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.

Pragma 8.2.1. Most of the locks in this book follow the interface shown in Fig. 8.1. Here is an explanation 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 8.2.2.)
- The `unlock()` method releases the lock.
- The `newCondition()` method is a *factory* that creates and returns a `Condition` object associated with the lock (explained below.)
- 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 Elsewhere 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.

```
public interface Lock {
  void lock();
  void lockInterruptibly() throws InterruptedException;
  boolean tryLock();
  boolean tryLock(long time, TimeUnit unit);
  Condition newCondition();
  void unlock();
}
```

Figure 8.1 The Lock Interface.

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 is a very good idea to release the lock on the queue, because otherwise the other thread will never be able to enqueue the anticipated item. After the waiting thread has released the lock, it needs a way to be notified when to reacquire the lock and try again.

In the Java concurrency package (and in related packages such as Pthreads), the ability to release a lock temporarily is provided by a `Condition` object associated with a lock. Fig. 8.2 shows the use of 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.

Pragma 8.2.2. Threads in Java can be *interrupted* by other threads. If a thread is interrupted during a call to a `Condition`'s `await()` method, then the call throws `InterruptedException`. The proper response to an interrupt is application-dependent. (It is not good programming practice simply to ignore interrupts).
Fig. 8.2 shows a schematic example

```
Condition condition = mutex.newCondition();
...
mutex.lock()
try {
  while (!property) { // not happy
     condition.await(); // wait for property
  }   catch (InterruptedException e) {
          ... // application-dependent response
          }
  ... // happy: property must hold
}
```

Figure 8.2 How to use Condition objects.

To avoid clutter, we usually omit `InterruptedException` handlers from example code, even though they would be required in actual code.

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 at the time 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 still does not hold, 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, while calling `signalAll()` wakes up all waiting threads. Fig. 8.4 describes a schematic execution of a monitor lock.

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. As discussed in Section 8.4, this lock is *reentrant*: a thread that is holding the lock can acquire it again without blocking.

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

```
public interface Condition {
  void await() throws InterruptedException;
  boolean await(long time, TimeUnit unit)
    throws InterruptedException;
  boolean awaitUntil(Date deadline)
    throws InterruptedException;
  long awaitNanos(long nanosTimeout)
    throws InterruptedException;
  void awaitUninterruptibly();
  void signal();      // wake up one waiting thread
  void signalAll();   // wake up all waiting threads
}
```

Figure 8.3 The `Condition` interface: `await()` and its variants release the lock, give up the processor, and later awaken and reacquire the lock. The `signal()` and `signalAll()` methods awaken one or more waiting threads.

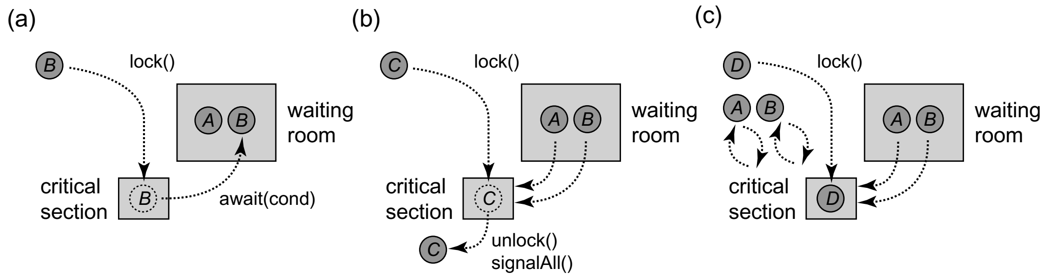

Figure 8.4 A schematic representation of a monitor execution. In Part (a) thread *A* has acquired the monitor lock, called `await()` on a condition, released the lock, and is now in the waiting room. Thread *B* then goes through the same sequence of steps, entering the critical section, calling `await()` 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 *C* leaves the critical section. Notice that if *C* would have issued a `signal()` instead of a `signalAll()`, only one of *A* or *B* would have left the waiting room, and the other would have continued to wait.

This combination of methods, mutual exclusion locks, and condition objects is called a *monitor*.

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 non-empty? 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, removes the first item, but *B*, victim of a lost wakeup, waits forever even though there is an item in the buffer to be consumed.

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

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

```
class LockedQueue<T> {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final T[] items;
  int tail, head, count;
  public LockedQueue(int capacity) {
    items = (T[])new Object[100];
  }
  public void enq(T x) {
    lock.lock();
    try {
      while (count == items.length)
        notFull.await();
      items[tail] = x;
      if (++tail == items.length)
        tail = 0;
      ++count;
      notEmpty.signal();
    } finally {
      lock.unlock();
    }
  }
  public T deq() {
    lock.lock();
    try {
      while (count == 0)
        notEmpty.await();
      T x = items[head];
      if (++head == items.length)
        head = 0;
      --count;
      notFull.signal();
      return x;
    } finally {
      lock.unlock();
    }
  }
}
```

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.

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

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

```
public void enq(T x) {
  lock.lock();
  try {
    while (count == items.length)
      notFull.await();
    items[tail] = x;
    if (++tail == items.length)
      tail = 0;
    ++count;
    if (count == 1) {  // Wrong!
      notEmpty.signal();
    }
  } finally {
    lock.unlock();
  }
}
```

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, called *readers*, return information about the object's state without modifying the object, while only a small number of calls, called *writers*, actually modify the object.

There is no need for readers to 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.

8.3.1 Simple Readers–Writers Lock

We consider a sequence of increasingly sophisticated reader–writer lock implementations. The `SimpleReadWriteLock` class appears in Figs. 8.7–8.9. This

```java
public class SimpleReadWriteLock implements ReadWriteLock {
  int readers;
  boolean writer;
  Lock lock;
  Condition condition;
  Lock readLock, writeLock;
  public SimpleReadWriteLock() {
    writer = false;
    readers = 0;
    lock = new ReentrantLock();
    readLock = new ReadLock();
    writeLock = new WriteLock();
    condition = lock.newCondition();
  }
  public Lock readLock() {
    return readLock;
  }
  public Lock writeLock() {
    return writeLock;
  }
```

Figure 8.7 The `SimpleReadWriteLock` class: fields and public methods.

```java
  class ReadLock implements Lock {
    public void lock() {
      lock.lock();
      try {
        while (writer) {
          condition.await();
        }
        readers++;
      } finally {
        lock.unlock();
      }
    }
    public void unlock() {
      lock.lock();
      try {
        readers--;
        if (readers == 0)
          condition.signalAll();
      } finally {
        lock.unlock();
      }
    }
  }
```

Figure 8.8 The `SimpleReadWriteLock` class: the inner read lock.

```
  protected class WriteLock implements Lock {
    public void lock() {
      lock.lock();
      try {
        while (readers > 0) {
          condition.await();
        }
        writer = true;
      } finally {
        lock.unlock();
      }
    }
    public void unlock() {
      writer = false;
      condition.signalAll();
    }
  }
}
```

Figure 8.9 The SimpleReadWriteLock class: inner write lock.

class uses a counter to keep track of the number of readers that have acquired the lock, and a Boolean field indicating whether a writer has acquired the lock. To define the associated read–write locks, this code uses *inner classes*, a Java feature that allows one object (the SimpleReadWriteLock lock) to create other objects (the read–write locks) that share the first object's private fields. Both the readLock() and the writeLock() methods return objects that implement the Lock interface. These objects communicate via the writeLock() object's fields. Because the read–write lock methods must synchronize with one another, they both synchronize on the mutex and condition fields of their common SimpleReadWriteLock object.

8.3.2 Fair Readers–Writers Lock

Even though the SimpleReadWriteLock algorithm is correct, it is still not quite satisfactory. If readers are much more frequent than writers, as is usually the case, then writers could be locked out for a long time by a continual stream of readers. The FifoReadWriteLock class, shown in Figs. 8.10–8.12, shows one way to give writers priority. This class ensures that once a writer calls the write lock's lock() method, then no more readers will be able to 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 will 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. (We are, of course, ignoring potential integer overflow and wraparound problems.) The class has a

```
public class FifoReadWriteLock implements ReadWriteLock {
  int readAcquires, readReleases;
  boolean writer;
  Lock lock;
  Condition condition;
  Lock readLock, writeLock;
  public FifoReadWriteLock() {
    readAcquires = readReleases = 0;
    writer   = false;
    lock  = new ReentrantLock();
    condition = lock.newCondition();
    readLock = new ReadLock();
    writeLock = new WriteLock();
  }
  public Lock readLock() {
    return readLock;
  }
  public Lock writeLock() {
    return writeLock;
  }
  ...
}
```

Figure 8.10 The `FifoReadWriteLock` class: fields and public methods.

```
  private class ReadLock implements Lock {
    public void lock() {
      lock.lock();
      try {
        readAcquires++;
        while (writer) {
          condition.await();
        }
      } finally {
        lock.unlock();
      }
    }
    public void unlock() {
      lock.lock();
      try {
        readReleases++;
        if (readAcquires == readReleases)
          condition.signalAll();
      } finally {
        lock.unlock();
      }
    }
  }
```

Figure 8.11 The `FifoReadWriteLock` class: inner read lock class.

```
 private class WriteLock implements Lock {
  public void lock() {
   lock.lock();
  try {
    while (readAcquires != readReleases)
     condition.await();
    writer = true;
   } finally {
    lock.unlock();
   }
  }
  public void unlock() {
   writer = false;
  }
 }
}
```

Figure 8.12 The `FifoReadWriteLock` class: inner write lock class.

private `lock` field, held by readers for short durations: they acquire the lock, increment the `readAcquires` field, and release the lock. Writers hold this lock from the time they try to acquire the write lock to the time they release it. This locking protocol ensures that once a writer has acquired the `lock`, no additional reader can increment `readAcquires`, so no additional reader can acquire the read lock, and eventually all readers currently holding the read lock will release it, allowing the writer to proceed.

How are waiting writers notified when the last reader releases its lock? When a writer tries to acquire the write lock, it acquires the `FifoReadWriteLock` object's `lock`. A reader releasing the read lock also acquires that `lock`, and calls the associated condition's `signal()` method if all readers have released their locks.

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 non-reentrant lock. This exercise is intended to illustrate how to use locks and conditions. In practice, the java.util.concurrent.locks package provides reentrant lock classes, so there is no need to write our own.

```
public class SimpleReentrantLock implements Lock{
  Lock lock;
  Condition condition;
  int owner, holdCount;
  public SimpleReentrantLock() {
    lock = new SimpleLock();
    condition = lock.newCondition();
    owner = 0;
    holdCount = 0;
  }
  public void lock() {
    int me = ThreadID.get();
    lock.lock();
    if (owner == me) {
      holdCount++;
      return;
    }
    while (holdCount != 0) {
      condition.await();
    }
    owner = me;
    holdCount = 1;
  }
  public void unlock() {
    lock.lock();
    try {
      if (holdCount == 0 || owner != ThreadID.get())
        throw new IllegalMonitorStateException();
      holdCount--;
      if (holdCount == 0) {
        condition.signal();
      }
    } finally {
      lock.unlock();
    }
  }

  public Condition newCondition() {
    throw new UnsupportedOperationException("Not supported yet.");
  }
 ...
}
```

Figure 8.13 The `SimpleReentrantLock` class: `lock()` and `unlock()` methods.

Fig. 8.13 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. In Fig. 8.13, we initialze 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 14). Otherwise, if the hold count is not zero, the lock is held by another thread, and the caller releases the lock and waits until the condition is signaled (Line 19). When the caller awakens, it must still check that the hold count is zero. When the hold count is established to be zero, the calling thread makes itself the owner and sets the hold count to 1.

The `unlock()` method acquires the internal lock (Line 25). It throws an exception if either the lock is free, or the caller is not the owner (Line 27). 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 31).

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. A `Semaphore` is a generalization of mutual exclusion locks. Each `Semaphore` has a *capacity*, denoted by c for brevity. Instead of allowing only one thread at a time into the critical section, a `Semaphore` allows at most c threads, where the capacity c is determined when the `Semaphore` is initialized. As discussed in the chapter notes, semaphores were one of the earliest forms of synchronization.

The `Semaphore` class of Fig. 8.14 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 c, the calling thread is suspended until there is room. When a thread leaves the critical section, it calls `release()` to notify a waiting thread that there is now room.

8.6 Chapter Notes

Monitors were invented by Per Brinch-Hansen [52] and Tony Hoare [71]. Semaphores were invented by Edsger Dijkstra [33]. McKenney [113] surveys different kinds of locking protocols.

```
public class Semaphore {
  final int capacity;
  int state;
  Lock lock;
  Condition condition;
  public Semaphore(int c) {
    capacity = c;
    state = 0;
    lock = new ReentrantLock();
    condition = lock.newCondition();
  }
  public void acquire() {
    lock.lock();
    try {
      while (state == capacity) {
        condition.await();
      }
      state++;
    } finally {
      lock.unlock();
    }
  }
  public void release() {
    lock.lock();
    try {
      state--;
      condition.signalAll();
    } finally {
      lock.unlock();
    }
  }
}
```

Figure 8.14 Semaphore implementation.

8.7 Exercises

Exercise 93. Reimplement the `SimpleReadWriteLock` class using Java **`synchronized`**, `wait()`, `notify()`, and `notifyAll()` constructs in place of explict locks and conditions.

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

Exercise 94. 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 95. 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 let us 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();}
   }
   ```

   We are given a set of 10 accounts, whose balances are unknown. At 1:00, each of n threads tries to transfer \$100 from another account into its own account. At 2:00, a Boss thread deposits \$1000 to each account. Is every transfer method called at 1:00 certain to return?

Exercise 96. In the *shared bathroom problem*, there are two classes of threads, called *male* and *female*. There is a single *bathroom* resource that must be used in the following way:

1. Mutual exclusion: persons of opposite sex may not occupy the bathroom simultaneously,
2. Starvation-freedom: everyone who needs to use the bathroom eventually enters.

The protocol is implemented via the following four procedures: `enterMale()` delays the caller until it is ok for a male to enter the bathroom, `leaveMale()` is called when a male leaves the bathroom, while `enterFemale()` and `leaveFemale()` do the same for females. For example,

```
enterMale();
teeth.brush(toothpaste);
leaveMale();
```

1. Implement this class using locks and condition variables.
2. Implement this class using **`synchronized`**, `wait()`, `notify()`, and `notifyAll()`.

For each implementation, explain why it satisfies mutual exclusion and starvation-freedom.

Exercise 97. The Rooms class manages a collection of *rooms*, indexed from 0 to *m* (where *m* is an argument to the constructor). Threads can enter or exit any room in that range. Each room can hold an arbitrary number of threads simultaneously, but only one room can be occupied at a time. For example, if there are two rooms, indexed 0 and 1, then any number of threads might enter the room 0, but no thread can enter the room 1 while room 0 is occupied. Fig. 8.15 shows an outline of the Rooms class.

Each room can be assigned an *exit handler*: calling setHandler(i, h) sets the exit handler for room i to handler h. The exit handler is called by the last thread to

```java
public class Rooms {
  public interface Handler {
    void onEmpty();
  }
  public Rooms(int m) { ... };
  void enter(int i) { ... };
  boolean exit() { ... };
  public void setExitHandler(int i, Rooms.Handler h) { ... };
}
```

Figure 8.15 The Rooms class.

```java
class Driver {
  void main() {
    CountDownLatch startSignal = new CountDownLatch(1);
    CountDownLatch doneSignal = new CountDownLatch(n);
    for (int i = 0; i < n; ++i) // start threads
      new Thread(new Worker(startSignal, doneSignal)).start();
    doSomethingElse();          // get ready for threads
    startSignal.countDown();    // unleash threads
    doSomethingElse();          // biding my time ...
    doneSignal.await();         // wait for threads to finish
  }
  class Worker implements Runnable {
    private final CountDownLatch startSignal, doneSignal;
    Worker(CountDownLatch myStartSignal, CountDownLatch myDoneSignal) {
      startSignal = myStartSignal;
      doneSignal = myDoneSignal;
    }
    public void run() {
      startSignal.await();    // wait for driver's OK to start
      doWork();
      doneSignal.countDown(); // notify driver we're done
    }
    ...
  }
}
```

Figure 8.16 The CountDownLatch class: an example usage.

leave a room, but before any threads subsequently enter any room. This method is called once and while it is running, no threads are in any rooms.

Implement the `Rooms` class. Make sure that:

- If some thread is in room i, then no thread is in room $j \neq i$.
- The last thread to leave a room calls the room's exit handler, and no threads are in any room while that handler is running.
- Your implementation must be *fair*: any thread that tries to enter a room eventually succeeds. Naturally, you may assume that every thread that enters a room eventually leaves.

Exercise 98. Consider an application with distinct sets of *active* and *passive* threads, where we want to block the passive threads until all active threads give permission for the passive threads to proceed.

A `CountDownLatch` encapsulates a counter, initialized to be n, the number of active threads. When an active method is ready for the passive threads to run, it calls `countDown()`, which decrements the counter. Each passive thread calls `await`(), which blocks the thread until the counter reaches zero. (See Fig. 8.16.)

Provide a `CountDownLatch` implementation. Do not worry about reusing the `CountDownLatch` object.

Exercise 99. This exercise is a follow-up to Exercise 98. Provide a `CountDownLatch` implementation where the `CountDownLatch` object can be reused.