The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 8

July 14, 2009

Exercise93. 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.

Solution The simplest approach is to eliminate the superclass lock and condition fields, and replace each call to lock() and await() with calls to the superclass's lock and wait methods. See Fig. **??**.

Exercise94. 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.

Solution First, to be a read-write lock, the ReentrantReadWriteLock class must keep track of two kinds of information: is there a writer, and are there readers?

Second, to be reentrant, the class must keep track of the identity of the writer, so that writer will be granted the lock if it re-requests it. For readers, however, there is no need to keep track of identies. Their number is enough. A thread rerequesting a read lock simply reincrements the reader count.

Supose we were to allow readers to upgrade. If a reader finds there are no writers, and only a single reader (itself) then it can increment the writers field and zero out the readers field.

What if it finds no writers, but a reader count of 2? If these readers are distinct, then it cannot upgrade, but if these readers represent two lock calls by the same thread, then it can upgrade. Without keeping track of reader identies, it is impossible to distinguish between these two situations.

In short, we would need to keep track of readers' *identities*, not just their numbers. Keeping track of identities is much less efficient: we need a hash table or something similar. If upgrades are rare, this places a sustantial cost on the common case.

Exercise 95. A *savings account* object holds a nonn egative 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.

```
public class SimpleReadWriteLockMonitor implements ReadWriteLock {
  int readers;
  boolean writer;
  Lock readLock, writeLock;
  public SimpleReadWriteLockMonitor() {
    writer = false;
    readers = 0;
    readLock = new ReadLock();
    writeLock = new WriteLock();
  }
  ...
  class ReadLock implements Lock {
    public void lock() {
      synchronized (SimpleReadWriteLockMonitor.this) {
        while (writer) {
          try {
            SimpleReadWriteLockMonitor.this.wait();
          } catch (InterruptedException e) {
          }
        }
        readers++;
      }
    }
    public void unlock() {
      synchronized (SimpleReadWriteLockMonitor.this) {
        readers--;
        if (readers == 0) {
          SimpleReadWriteLockMonitor.this.notifyAll();
        }
      }
    }
  ...
  }
  protected class WriteLock implements Lock {
    public void lock() {
      synchronized (SimpleReadWriteLockMonitor.this) {
        while (readers > 0 || writer) {
          try {
            SimpleReadWriteLockMonitor.this.wait();
          } catch (InterruptedException e) {
          }
        }
        writer = true;
      }
    }
    public void unlock() {
      writer = false;
      SimpleReadWriteLockMonitor.this.notifyAll();
    }
  ...
}
```

Figure 1: The SimpleReadWriteLock class using Java primitives

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?

Solution Fig **??** shows one solution. The

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(); and
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.

```
public class SavingsAccount {
  int balance;
  int preferredWaiting;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  void withdraw(boolean preferred, int amount) throws InterruptedException {
    lock.lock();
    try {
      if (preferred) {
        preferredWaiting++;
        while (preferredWaiting > 1) {
          condition.await();
        }
        preferredWaiting--;
      } else {
        while (preferredWaiting > 0) {
          condition.await();
        }
      }
      balance -= amount;
      lock.unlock();
      condition.signalAll();
    }
  }
  void deposit(int amount) {
    lock.lock();
    try {
      balance += amount;
    } finally {
      lock.unlock();
    }
  }
}
```

Figure 2: The SavingsAccount class

Solution Fig. **??** shows a solution using locks and condtion variables, and Fig. **??** shows a solution using Java monitors. In both implementations, the class keeps track of the number of males or females in the bathroom. A male can enter when the number of females falls to zero, and vice-versa. To ensure fairness, the turn field gives preference to the other gender when one person

```
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 3: The Rooms class.

```
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 4: The CountDownLatch class: an example usage.

```
public class Bathroom {
  enum Sex { Male, Female }
  int males = 0;
  int females = 0;
  Sex turn = Sex.Male;
  Lock lock = new ReentrantLock();
  Condition condition = lock.newCondition();
  void enterMale() throws InterruptedException {
    lock.lock();
    try {
      while (turn == Sex.Female && females > 0) {
        condition.await();
      }
      males++;
    } finally {
      lock.unlock();
    }
  }
  void leaveMale() throws InterruptedException {
    lock.lock();
    try {
      males--;
      turn = Sex.Female;
    } finally {
      condition.signalAll();
      lock.unlock();
    }
  }
 ...
}
```

Figure 5: The Bathroom class

```
   ...
  synchronized void enterMale() throws InterruptedException {
    while (turn == Sex.Female && females > 0) {
      wait();
    }
    males++;
  }
  synchronized void leaveMale() throws InterruptedException {
    males--;
    turn = Sex.Female;
    notifyAll();
  }
   ...
```

Figure 6: The Bathroom class using Java Monitors

leaves the bathroom.

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. **??** 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 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.

Solution Here are two distinct implementations. The TicketRooms class (Fig. 7), is organized around the rooms, while the QueueRooms class ((Fig. 7), is organized around the threads.

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. .)

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

Solution Here is a straightforward CountDownLatch implementation. The lock and condition fields provide synchronization, and the counter field counts the number of missing permissions. The await() method acquires the lock (Line **??**), and waits until the counter field reaches zero (Lines **??** - **??**). The signal() method simply decrements the counter field (Line **??**), and awakens the passive threads (Line **??**).

```
public class TicketRooms implements Rooms {
  int numRooms;               // how many rooms
  AtomicInteger activeRoom; // index of active room or NONE
  AtomicIntegerArray wait;   // how many enter requests
  AtomicIntegerArray grant; // how many enter grants
  AtomicIntegerArray done;   // how many exits
  Rooms.Handler[] handler;   // per−room exit handler
  private static final int NONE = −1; // no room is free
  public TicketRooms(int m) {
    numRooms = m;
    wait   = new AtomicIntegerArray(m);
    grant  = new AtomicIntegerArray(m);
    done   = new AtomicIntegerArray(m);
    handler  = new Rooms.Handler[m];
    activeRoom = new AtomicInteger(NONE);
  }
  public void enter(int i) {
    int myTicket = wait.getAndIncrement(i) + 1; // get ticket
    while (myTicket > grant.get(i)) {              // spin until granted
      if (activeRoom.get() == NONE) {           // if no active room
        if (activeRoom.compareAndSet(NONE,i)) { // make my room active
          grant.set(i, wait.get(i));
// let all with tickets enter
          return;
        }
      }
    }
  }
  public boolean exit() {
    int room = activeRoom.get();           // preparing to exit
    int myDone = done.getAndIncrement(room) + 1; // increment done counter
    if (myDone == grant.get(room)) {     // Am I last?
      if (handler[room] != null)           // if handler defined,
        handler[room].onEmpty();           // call it
      for (int k=0; k < numRooms; k++) {
        room = (room + 1) % numRooms; // round robin through rooms
        if (wait.get(room) > grant.get(room)) { // someone waiting?
          activeRoom.set(room);                 // make this room active
          grant.set(room, wait.get(room));  // admit threads with tickets
          return true;                          // I am last to leave
        }
      }
      activeRoom.set(NONE); // no waiters, no active reoom
      return true;                  // I am last to leave
    }
    return false;                   // I am not last to leave
  }
  public void setExitHandler(int i, Rooms.Handler h) {
    handler[i] = h;
  }
}
```

Figure 7: Ticket-Based Rooms implementation

```
public class QueueRooms implements Rooms {
  int numRooms;          // number of rooms
  int inside;            // number of threads in room
  int activeRoom;        // index of active room
  Queue<ThreadInfo> queue; // queue of waiting threads
  Rooms.Handler[] handler;
  private static final int NONE = -1; // no room is free
  public QueueRooms(int n) {
    numRooms = n;
    inside = 0;
    activeRoom = NONE;
    queue = new LinkedList<ThreadInfo>();
    handler = new Rooms.Handler[n];
  }
  public void enter(int room) {
    ThreadInfo info = new ThreadInfo(room);
    synchronized(this) {
      if (queue.isEmpty() && inside == 0) {
        activeRoom = room;
        inside = 1;
        return;
      } else {
        queue.add(info);
      }
    }
    while (info.wait) {}; // spin
  }
  public synchronized boolean exit() {
    inside--;
    if (inside == 0) {              // last one out
      if (handler[activeRoom] != null)     // call exit handler
        handler[activeRoom].onEmpty();
      if (!queue.isEmpty()) {              // someone waiting?
        ThreadInfo info = queue.remove(); // dequeue first
        info.wait = false;                 // wake up first
        activeRoom = info.room;
        Queue<ThreadInfo> newQueue = new LinkedList<ThreadInfo>(); // wake up same room
        while (!queue.isEmpty()) {
          info = queue.remove();
          if (info.room == activeRoom)
            info.wait = false; // wake up thread
          else
            newQueue.add(info); // keep thread asleep
        }
        queue = newQueue;
      }
      return true;
    }
    return false;
  }
  public void setExitHandler(int i, Rooms.Handler h) {
    handler[i] = h;
  }
  class ThreadInfo {
    public boolean wait;
    public int room;
```

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

Solution See Fig .

```
public class CountDownLatch {
  int counter;
  Lock lock;
  Condition condition;
  public CountDownLatch(int count) {
    if (count < 0)
      throw new IllegalArgumentException("count < 0");
    counter = count;
    lock = new ReentrantLock();
    condition = lock.newCondition();
  }
  public void await() throws InterruptedException {
    lock.lock();
    try {
      while (counter > 0)
        condition.await();
    } finally {
      lock.unlock();
    }
  }
  public void countDown() {
    lock.lock();
    try {
      counter--;
      if (counter == 0) {
        condition.signalAll();
      }
    } finally {
      lock.unlock();
    }
  }
}
```

Figure 9: Resuable CountDownLatch