44

READ-WRITE LOCK

This pattern was previously described in Grand98 and is based on the material that appeared in Lea97.

DESCRIPTION

During the discussion of the Monitor and the Critical Section patterns earlier, we saw that when multiple threads in an application simultaneously access a resource it could result in unpredictable behavior. Hence the resource must be protected so that only one thread at a time is allowed to access the resource. Though this may be required in most cases, it may lead to unwanted CPU overhead when some of the threads accessing the resource are interested only in reading the values or state of the resource but not in changing it. In such cases, it can be inefficient to prevent a thread from accessing the resource solely to read its values while a different thread is currently reading the same resource values. Because a read operation does not alter the values of the resource, multiple threads can safely be allowed to access the resource at the same time if all of these threads are interested only in reading the resource values. This kind of design improves the overall application responsiveness with reduced CPU overhead. That means, when a thread obtains a lock to simply read the values of a resource, it should not prevent other threads from accessing the resource to read its values. In other words, a read lock should be shared. If a thread is allowed to read a resource's data while a different thread is updating the same resource, the thread that is reading the data may receive an inconsistent view. Allowing more than one thread to update the values of a resource could also result in unpredictable results.

While some threads are interested only in reading the resource values, some other threads may access the resource to read and update its values. To eliminate concurrency problems, when such a thread needs to access the resource to update its values, it must get a write lock on the object representing the resource. A write lock is an exclusive lock on the object and prevents all other threads from accessing the resource at the same time. Further, if a read and a write lock are requested on an object at the same time, the write lock request should be granted first. The write lock is issued only if there are no threads currently holding a read lock on the same object.

Table 44.1 summarizes the criteria for issuing a read-write lock.

Table 44.1 Rules for Issuing Read-Write Locks

Lock	*Rules*
Read Lock	A read lock should be issued if there is no currently issued write lock and there are no threads waiting for the write lock.
Write Lock	A write lock should be issued if no thread is currently issued a (read or write) lock on the object.

In Java, there is no readily available feature for implementing read-write locks. But a custom class can be built (Listing 44.1) with the responsibility of issuing read-write locks on an object to different threads in an application.

Design Highlights of the `ReadWriteLock` Class

Lock Statistics

The `ReadWriteLock` maintains different lock statistics in a set of instance variables as follows:

- `totalReadLocksGiven` — To store the number of read locks already issued on the object.
- `writeLockIssued` — To indicate if a write lock has been issued or not.
- `threadsWaitingForWriteLocks` — To keep track of the number of threads currently waiting for a write lock.

These values are in turn used by the lock issuing methods — `getReadLock` and `getWriteLock`.

Lock Methods

The `ReadWriteLock` offers two methods — `getReadLock` and `getWriteLock` — which can be used by client objects to get read and write locks on an object, respectively. As part of its implementation of these two methods, the `ReadWriteLock` issues read-write locks as per the rules listed in Table 44.1.

Lock Release

A client object that currently holds a read-write lock can release the lock by invoking the `done` method. The `done` method updates appropriate lock statistics and allows the lock to be issued to any waiting thread as per the rules listed in Table 44.1.

The `ReadWriteLock` class is a generic implementation for issuing read-write locks and can be readily used in any application.

Listing 44.1 Generic `ReadWriteLock` Implementation

```
public class ReadWriteLock {
  private Object lockObj;
  private int totalReadLocksGiven;
  private boolean writeLockIssued;
  private int threadsWaitingForWriteLock;
  public ReadWriteLock() {
    lockObj = new Object();
    writeLockIssued = false;
  }
  /*
    A read lock can be issued if
      there is no currently issued
      write lock and
      there is no thread(s) currently waiting for the
      write lock
    */
  public void getReadLock() {
    synchronized (lockObj) {
      while ((writeLockIssued) ||
        (threadsWaitingForWriteLock != 0)) {
      try {
        lockObj.wait();
      } catch (InterruptedException e) {
        //
      }
    }
    //System.out.println(" Read Lock Issued");
    totalReadLocksGiven++;
  }
}
/*
  A write lock can be issued if
    there is no currently issued
    read or write lock
  */
```

(continued)

Listing 44.1 Generic `ReadWriteLock` Implementation (Continued)

```
public void getWriteLock() {
  synchronized (lockObj) {
    threadsWaitingForWriteLock++;
    while ((totalReadLocksGiven != 0) ||
        (writeLockIssued)) {
      try {
        lockObj.wait();
      } catch (InterruptedException e) {
        //
      }
    }
    //System.out.println(" Write Lock Issued");
    threadsWaitingForWriteLock -- ;
    writeLockIssued = true;
  }
}
//used for releasing locks
public void done() {
  synchronized (lockObj) {
    //check for errors
    if ((totalReadLocksGiven == 0) &&
        (!writeLockIssued)) {
      System.out.println(
        " Error: Invalid call to release the lock");
      return;
    }
    if (writeLockIssued)
      writeLockIssued = false;
    else
      totalReadLocksGiven -- ;
      lockObj.notifyAll();
    }
  }
}
```

EXAMPLE

Applying the Read-Write Lock pattern, let us design an application to allow members of a library to:

- View details of different library items
- Check out an item if it is currently available

The application must ensure that multiple members are allowed to view an item status at the same time, but only one member is allowed to check out an item at a time. In other words, the application must support multiple simultaneous member transactions without producing unpredictable results.

The overall application design becomes much simpler using the `ReadWriteLock` class designed earlier. The representation of a library item can be designed in the form of an `Item` class (Listing 44.2) with methods to allow members to check the status of an item and to check in or check out an item.

Because the status check of an item does not involve changes to its status, the `getStatus` method acquires a read lock. This allows more than one thread to invoke the `getStatus` method to check the status of an item.

In contrast, both the `checkIn` and `checkOut` methods involve changes to the item status and hence acquire a write lock before changing the item status. This ensures that only one thread is allowed to alter the item status even though more than one thread invokes the `checkIn/checkOut` method at the same time. The `Item` class makes use of the services of a `ReadWriteLock` object to acquire an appropriate lock.

By using the exclusive write lock only when needed, the `Item` class allows multiple threads to access an item in a more controlled manner without the overhead of any unwanted waiting and eliminates the scope for unpredictable behavior at the same time.

The representation of a member transaction can be designed as a Java Thread (Listing 44.3) to facilitate the reflection of the real world scenario of different members accessing an item simultaneously.

The `MemberTransaction` class is designed in its simplest form and can be configured with an operation to check an item status or to check in or check out an item when it is instantiated.

To simulate a real world scenario, a test program `RWTest` can be designed to create multiple `MemberTransaction` objects to perform different operations to read the status of an item or check in or check out an item.

```
public class RWTest {
  public static void main(String[] args) {
    Item item = new Item("CompScience-I");
    new MemberTransaction("Member1", item, "StatusCheck");
    new MemberTransaction("Member2", item, "StatusCheck");
    new MemberTransaction("Member3", item, "CheckOut");
    new MemberTransaction("Member4", item, "CheckOut");
```

```
    new MemberTransaction("Member5", item, "CheckOut");
    new MemberTransaction("Member6", item, "StatusCheck");
  }
}
```

When the `RWTest` is executed, the order in which different read-write locks are issued will be displayed.

Listing 44.2 `Item` Class

```
public class Item {
  private String name;
  private ReadWriteLock rwLock;
  private String status;
  public Item(String n) {
    name = n;
    rwLock = new ReadWriteLock();
    status = "N";
  }
  public void checkOut(String member) {
    rwLock.getWriteLock();
    status = "Y";
    System.out.println(member +
                       " has been issued a write lock-ChkOut");
    rwLock.done();
  }
  public String getStatus(String member) {
    rwLock.getReadLock();
    System.out.println(member +
                       " has been issued a read lock");
    rwLock.done();
    return status;
  }
  public void checkIn(String member) {
    rwLock.getWriteLock();
    status = "N";
    System.out.println(member +
                       " has been issued a write lock-ChkIn");
    rwLock.done();
  }
}
```

Listing 44.3 `MemberTransaction` Class

```
public class MemberTransaction extends Thread {
  private String name;
  private Item item;
  private String operation;
  public MemberTransaction(String n, Item i, String p) {
    name = n;
    item = i;
    operation = p;
    start();
  }
  public void run() {
    //all members first read the status
    item.getStatus(name);
    if (operation.equals("CheckOut")) {
      System.out.println("\n" + name +
                         " is ready to checkout the item.");
      item.checkOut(name);
      try {
        sleep(1);
      } catch (InterruptedException e) {
        //
      }
      item.checkIn(name);
    }
  }
}
```

PRACTICE QUESTIONS

1. Design an application to allow different customers to buy airline tickets. Apply the Read-Write Lock pattern to ensure that multiple customers are allowed to check the seat availability on the same flight, but only one customer is allowed to buy the ticket at a time.
2. Design an application to allow different customers to bid on auctioned items. Apply the Read-Write Lock pattern to ensure that multiple customers are allowed to check the current bid but no two customers are allowed to alter the bid amount at the same time.