



CHAPTER 9

Reader/Writer lock

- 9.1 Acquiring a read lock from a `ReaderWriterLock` 161
- 9.2 Acquiring a writer lock from a `ReaderWriterLock` 166
- 9.3 `ReleaseLock` and `RestoreLock` 179
- 9.4 Summary 181

`ReaderWriterLock` is a synchronization mechanism allowing access to data. It allows multiple threads to read the data simultaneously, but only one thread at a time to update it. While a thread is updating, no other thread can read the data. The name is misleading. It may cause you to think there are two locks; in reality there is a single lock that restricts both reading and writing.

Think of how a conversation in a group generally goes. One person talks while the others listen. Think of how inefficient a conversation would be if only one person could talk to one person in a group at a given time. This is the very reason that conference calls are used. In business, it is often beneficial to have a single conference call, involving all of the parties at once, rather than have multiple person-to-person calls. A `ReaderWriterLock` allows multiple threads to read data at the same time. The only restriction is that a thread cannot modify the data while someone is reading it.

The majority of data accesses are reads, but occasionally a thread needs to change a value. This is problematic in that one thread may modify a data element while another one is accessing it. To combat this, the choices are to protect the element with a synchronization lock, such as `lock` and `SyncLock`, or to use `ReaderWriterLock`.

This chapter uses a simulated auction to demonstrate this concept. To test our synchronization system we can utilize multiple threads. Each thread will have a list of items it is instructed to acquire, along with an allotment of bidding points. Since an auction involves many reads to data and a few writes, it is ideal for demonstrating the concepts of a reader/writer lock.

The .NET implementation of `ReaderWriterLock` is efficient enough for highly granular use. In our example, each auction item has its own `ReaderWriterLock`, allowing for a higher level concurrency and ensuring fairness in lock allocation between threads. When a thread requests a write lock, no other threads will be granted a read lock until the write lock request is satisfied.

The `ReaderWriterLock` is a very useful construct. Most environments force developers to write their own or purchase a third-party tool, but the .NET platform makes this construct available for general use. `ReaderWriterLocks` are a powerful tool for selectively guarding data.

9.1 ACQUIRING A READ LOCK FROM A READERWRITERLOCK

The read portion of the `ReaderWriterLock` is the means that a thread uses to indicate that it is reading the protected data. This is needed because the determination of whether a thread can write to the protected data is based on the presence of one or more threads reading it. It doesn't make much sense for a read lock to be used without a write lock. If no thread is changing the data, there isn't much need in restricting access to it, since the data must be constant in nature. Figure 9.1 presents the logical structure of a `ReaderWriterLock`.

Remember that all access to a data element must be restricted to effectively protect the data. If there are ten ways to examine and three ways to update a data element, but only nine of the possible reading paths are protected, the concurrency issues that the `ReaderWriterLock` is supposed to be avoiding will still occur.

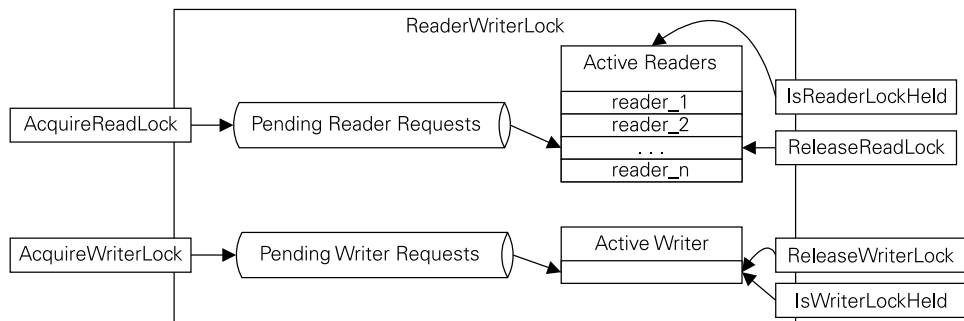


Figure 9.1 Logical structure of the `ReaderWriterLock`

9.1.1 Acquiring and releasing a reader lock

Suppose that you wanted to control access to data elements so that multiple consumers of that data could read it concurrently without data corruption. One way to do this is to use a `ReaderWriterLock`.

Reader-WriterLock A `ReaderWriterLock` is a synchronization mechanism that allows concurrent data reading but restricts data writing to occur only when no readers are present.

A `ReaderWriterLock` selectively allows access to data. It allows multiple threads to acquire a reader lock, which is acquired when the thread will be performing only read operations. Nothing keeps an errant thread from acquiring a reader lock and performing write operations. Care should be taken to ensure that only read operations occur in a region guarded by a read lock.

The power of a `ReaderWriterLock` is that it allows read operations to be logically separated from write operations. Since multiple read operations do not result in data corruption, there is no reason that multiple threads cannot simultaneously read a variable without ill effects.

To acquire a read lock, we invoke the `AcquireReadLock` method on the instance of the lock we wish to acquire. `AcquireReadLock` accepts a timeout value as its only parameter. As with many other synchronization methods, the timeout can either be an integer specifying the number of milliseconds to wait or an instance of the `TimeSpan` class.

In listing 9.1, we pass in the constant `Timeout.Infinite` to indicate we wish to wait indefinitely until we are able to acquire a read lock. We are assured that when the method returns we have acquired a read lock. As a general rule, using `Timeout.Infinite` is a bad idea. A better approach is to supply a timeout value because it removes the possibility of a `ReaderWriterLock`-related deadlock. To keep this chapter's examples simple, we use `Timeout.Infinite`.

Once a thread has acquired a read lock, it can perform any reads that are required, and once those reads are complete it should release the lock using the `ReleaseReadLock` method. The number of calls to `ReleaseReadLock` must match the number of calls to `AcquireReadLock`. If a thread fails to release the lock the same number of times it acquires the lock a write lock will not be granted to other threads. This will lead to deadlock if `Timeout.Infinite` is being used, as well as stopping the granting of any write locks.

TIP When the number of releases is greater than the number of acquires, a `System.ApplicationException` is thrown.

If a thread attempts to release a lock that it does not own, an exception is generated with the message "Attempt to release mutex not owned by caller." This might make you think that the `ReaderWriterLock` is implemented using the `Mutex` synchronization primitive we covered in section 8.5; however, it is not. This is a case of a somewhat

misleading error message. If this exception is encountered, it indicates that the number of releases is greater than the number of acquires.

Listing 9.1 Acquiring and releasing a read lock (VB.NET)

```
Private ItemLock As ReaderWriterLock
. . .
Public ReadOnly Property CurrentPrice() As Decimal
Get
    ItemLock.AcquireReaderLock(Timeout.Infinite)
    Try
        Return TheCurrentPrice
    Finally
        ItemLock.ReleaseReaderLock()
    End Try
End Get
End Property
```

← Declares the ReaderWriterLock

← Waits indefinitely for a reader lock

← Releases the reader lock

In Listing 9.1 notice that the `ReleaseReadLock` is located in a `Finally` statement. This ensures that if an exception is generated while the thread owns the read lock it will correctly be released. This is a good example of how `Finally` clauses should be used with exception handling. Figure 9.2 shows how multiple threads can access shared data using a `ReaderWriterLock`. Note that the shared data is not actually contained within the `ReaderWriterLock` but is guarded by it.

Each thread acquires the lock, accesses the data, and releases the lock. Since both threads are reading the data, there is no restriction on when the threads can access the data.

In this section we waited indefinitely to acquire the read lock. In the next section we'll discuss how to wait for a predetermined period of time. Once that time has expired, we need a means of determining if we have acquired the lock. Since `AcquireReaderLock` does not return a value, we must use the `IsReaderLockHeld` property that we discuss in the next section.

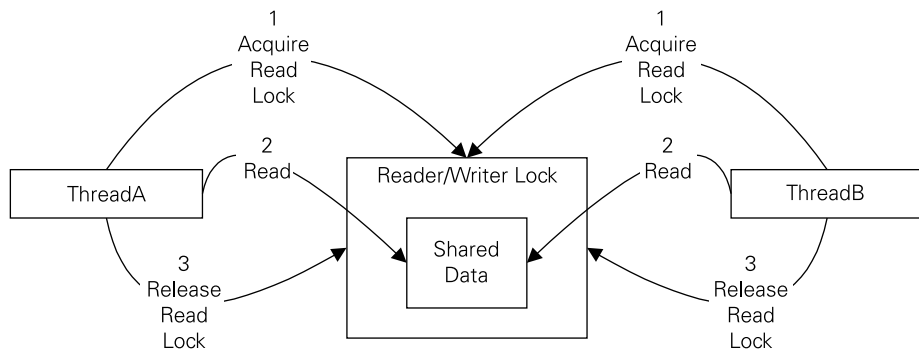


Figure 9.2 Two threads can read the shared data using the `ReaderWriterLock` to protect its value.

9.1.2 IsReaderLockHeld

Suppose you wanted to wait a certain amount of time for a lock to be acquired. The parameter to the `AcquireReaderLock` method specifies how long to wait for a reader lock to become available. As we saw in the previous section, it can either be an integer specifying the number of milliseconds to wait or a `TimeSpan` object. If the lock is not acquired in the specified time, an `ApplicationException` is raised. Listing 9.2 demonstrates one way of acquiring a reader lock and utilizing a timeout.

Listing 9.2 An improved way of acquiring a read lock (C#)

```
. . .
public decimal CurrentPrice
{
    get
    {
        do
        {
            try
            {
                ItemLock.AcquireReaderLock(1000); ← Attempts to acquire
                                                    a reader lock
            }
            catch(System.ApplicationException ex)
            {
                System.Diagnostics.Debug.WriteLine(ex.Message);
            }
        } while (!ItemLock.IsReaderLockHeld); ← Determines if the
                                                lock was acquired
        try
        {
            return TheCurrentPrice;
        }
        finally
        {
            ItemLock.ReleaseReaderLock(); ← Releases
                                          the lock
        }
    }
}
. . .
```

The code loops until it acquires the reader lock; if it takes more than one second to acquire the lock, an `ApplicationException` is raised. The property `IsReaderLockHeld` returns a Boolean value true if the current thread has a reader lock to the data, false if it does not.

IsReaderLockHeld `IsReaderLockHeld` is a property of the `ReaderWriterLock` class that indicates if the thread on which the executing code inspects the property currently owns a reader lock.

Another use for the `IsReaderLockHeld` property is to determine if invoking the `ReleaseReaderLock` method results in the lock being freed:

```
try
{
    return TheCurrentPrice;
}
finally
{
    ItemLock.ReleaseReaderLock();
    if (ItemLock.IsReaderLockHeld)
    {
        throw new Exception("Reader Lock still held after release");
    }
}
```

This can help detect situations where the number of releases is less than the number of acquires. The closer the error-detecting code is to the error, the easier it is to detect the error. If the error were not detected here, the mistake would likely manifest itself by having no other thread able to access the read lock. This would make the program hang. These sorts of issues are much more difficult to resolve without error-detecting instructions.

TIP Use the `IsReaderLockHeld` property to determine if a lock is held before `AcquireReaderLock` and after `ReleaseReaderLock`. This helps track down the number of acquires not matching the number of releases. Since there is a performance penalty, this sort of checking should only be performed during development. Release builds should not include this check.

When faced with an inconsistent or undesirable outcome, the first step should be to include robust error-detecting and -handling code. This is an area where exceptions and assertions can play a key role. Additionally, it is a good idea to determine if a thread already has a reader lock before the acquire method is called. The following code demonstrates a more defensive way of dealing with the acquire method:

```
. . .
if (ItemLock.IsReaderLockHeld)
{
    throw new Exception("Reader Lock held before acquire");
}
do
{
    try
    {
        ItemLock.AcquireReaderLock(1000);
    }
    . . .
```

The concept here is to make sure that the conditions of a thread are in the state you think they are. If not, throw an exception to help track down the error.

9.2 ACQUIRING A WRITER LOCK FROM A READERWRITERLOCK

In the previous section we discussed the reader portion of `ReaderWriterLock`. Now we turn to the write portion. The purpose of a write lock is to ensure that no threads are reading data while it is being updated.

9.2.1 Acquire, release, and `IsLockHeld`

The goal of a write lock is to enable multiple threads to read shared data while restricting write access in a way that ensures data corruption does not occur. We have already covered the read lock. Multiple threads can safely read data at the same time. Only one thread can be modifying data at one time. While a thread is modifying the shared data, no other thread can access the data without the risk of data corruption. In terms of our simulated auction, a write lock allows a new bid to be accepted. Listing 9.3 demonstrates the bidding process.

Listing 9.3 A bid must be higher than the current price (VB.NET).

```
Public Sub Bid(ByVal Amount As Decimal, ByVal BiddersName As String)
    If ItemLock.IsWriterLockHeld Then
        Throw New Exception("Writer lock held before acquire")
    End If
    Try
        Do
            Try
                ItemLock.AcquireWriterLock(TimeoutValue)
            Catch Ex As System.ApplicationException
                System.Diagnostics.Debug.WriteLine(Ex.Message)
            End Try
            Loop While Not ItemLock.IsWriterLockHeld
            If AuctionComplete Then
                Throw New Exception("Auction has ended")
            End If
            If (Amount > TheCurrentPrice) Then
                TheCurrentPrice = Amount
                TheBiddersName = BiddersName
            Else
                Throw New Exception("Bid not higher than current price")
            End If
        Finally
            ItemLock.ReleaseWriterLock()
            If (ItemLock.IsWriterLockHeld) Then
                Throw New Exception("Writer Lock still held after release")
            End If
        End Try
    End Sub
```

← If a write lock is held, throw an exception

← Try to acquire the lock

← Loop until the writer lock is acquired

← Once the update is complete, release the writer lock

As you can see in listing 9.3, it is similar to `AcquireReadLock` in that it accepts a timeout parameter. The `AcquireWriteLock` method is, obviously, used to acquire a write lock.

Acquire-WriterLock `AcquireWriterLock` is a method on the `ReaderWriterLock` class that allows a thread to request ownership of a write lock. It accepts a timeout parameter and throws an `ApplicationException` if the lock cannot be acquired in the specified time period.

If the write lock cannot be acquired within the specified duration, an exception is raised. Figure 9.3 shows the relationship between a read lock and a write lock.

At any given point a thread cannot have a write lock and some other thread have a read lock on the same instance of the `ReaderWriterLock` class. When a thread wishes to acquire a write lock, it calls `AcquireWriteLock`. It then must wait until all threads that currently have read locks release them. Once all threads have released the read locks, the requesting thread is granted its write lock. While that thread has a write lock, no other threads will be able to acquire a read or write lock.

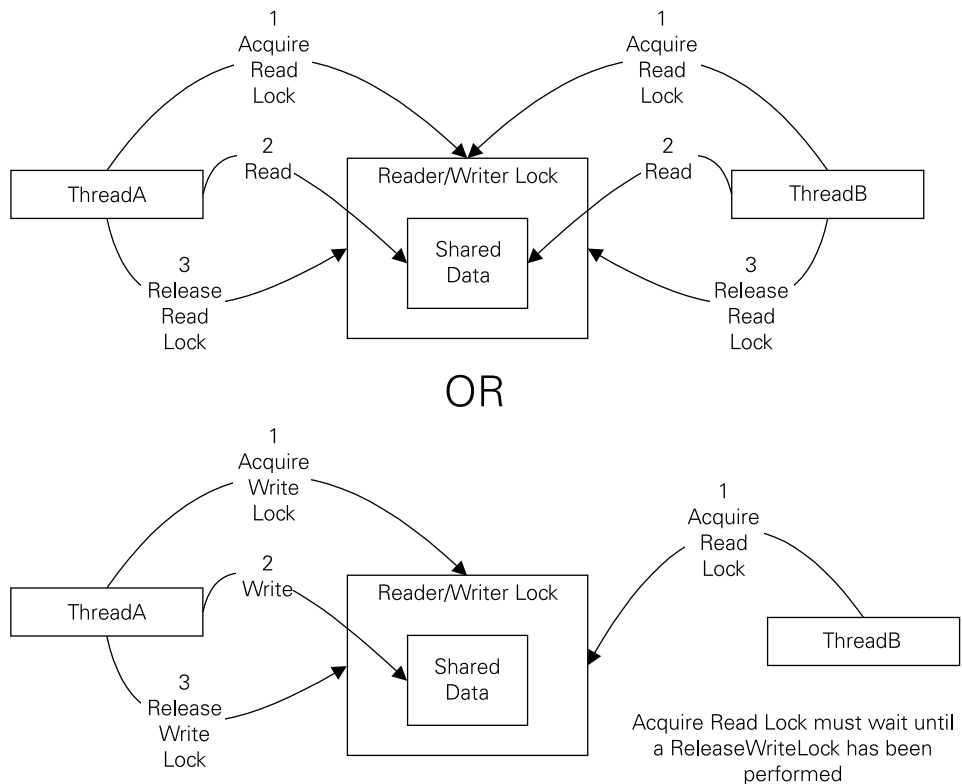


Figure 9.3 When a write lock has been granted, no thread will be granted a read or write lock until it is released.

IsWriterLockHeld `IsWriterLockHeld` is a property of the `ReaderWriterLock` class that allows a thread to determine if it has acquired a write lock on an instance of the `ReaderWriterLock` class. If the thread currently owns a write lock, `true` is returned.

To release a write lock, a thread uses the `ReleaseWriterLock` method of the `ReaderWriterLock` class. If the thread does not own the lock, an `ApplicationException` is raised with the message “Attempt to release mutex not owned by caller.” Once the thread has released its write lock, other threads are able to acquire their desired locks. This ensures that the data a thread is viewing doesn’t change while it is looking at it. Care should be taken to ensure that a thread does not modify shared data unless it currently owns a write lock.

9.2.2 UpgradeToWriterLock

There are times when it’s unclear if the lock required will be a reader or a writer. For example, in the auction simulation, in order to determine if a new bid is higher than the existing bid we must first look at what the current bid is (listing 9.4). Once we’ve examined the current bid, we can see if the new bid is higher.

Listing 9.4 Checks to see if the new bid is higher than the existing (C#)

```
public void Bid(decimal Amount, string BiddersName)
{
    if (ItemLock.IsWriterLockHeld)
    {
        throw new Exception("Writer Lock held before acquire");
    }
    if (ItemLock.IsReaderLockHeld)
    {
        throw new Exception("Reader Lock held before acquire");
    }
    ItemLock.AcquireReaderLock(Timeout.Infinite); ← Initially acquire a reader lock
    try
    {
        if (DateTime.Now > TheAuctionEnds)
        {
            throw new Exception("Auction has ended");
        }
        if (Amount > TheCurrentPrice) ← See if we need to acquire a writer lock
        {
            ItemLock.UpgradeToWriterLock(60000); ← Upgrade to a writer lock
            if (!ItemLock.IsWriterLockHeld)
            {
                throw new Exception("Writer Lock not held after upgrade");
            }
            if (Amount > TheCurrentPrice) ← Check to see if we're still the highest bidder
            {
                TheCurrentPrice = Amount;
                TheBiddersName=BiddersName;
            }
        }
    }
}
```

```

    }
    else
    {
        throw new Exception("Bid not higher than current price");
    }
}
else
{
    throw new Exception("Bid not higher than current price");
}
}
finally
{
    ItemLock.ReleaseReaderLock();
    if (ItemLock.IsWriterLockHeld)
    {
        throw new Exception("Writer Lock still held after release");
    }
    if (ItemLock.IsReaderLockHeld)
    {
        throw new Exception("Reader Lock still held after release");
    }
}
}
}

```

**ReleaseReaderLock
releases both
Reader and Writer**

In listing 9.4, it's unclear if a writer lock is needed until the bid amount is compared to the current price. In the last section, we dealt with this by acquiring a write lock. A more optimal solution is to acquire a read lock and determine if a write lock is required. If it is, we call `UpgradeToWriterLock`.

The advantage is that we only require a write lock when it is needed. Since write locks keep all reader locks from accessing data, using them unnecessarily results in reduced performance. Be careful when upgrading from a read to a write lock. There is a relatively small chance that during the transition from read to write some other pending write request may change the value. Figure 9.4 presents a graphical version of the logic involved.

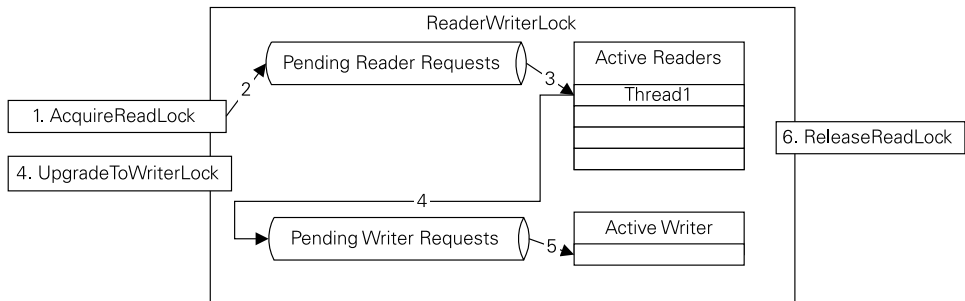


Figure 9.4 Acquiring a read lock

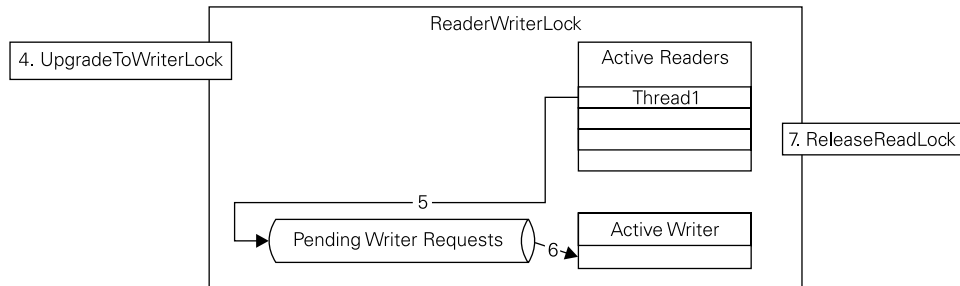


Figure 9.5 The steps involved in an UpgradeToWriterLock

The process begins by the calling thread requesting a read lock (step 1). This causes an entry to be added to the pending reader requests queue (step 2). Once that lock is granted (step 3), the calling thread can then request the upgrade to the writer lock. Figure 9.5 shows the steps involved in the upgrade.

When the calling thread calls `UpgradeToWriterLock` (step 4), the thread is removed from the list of active readers and placed in the pending writer requests queue (step 5). If a request from a different thread is already in the pending writer requests queue, it will be allowed to gain a write lock before the thread that requested the upgrade. The reason is the write lock requests are serviced in the order they are received, without any sort of priority associated with them. Once the requesting thread has been granted the write lock, it is moved to the active writer location (step 6). Listing 9.5 contains a class that can be used to see how a value can change during `UpgradeToWriterLock`.

Listing 9.5 A value can change during an UpgradeToWriterLock

```
Imports System.Threading
Public Class SimpleExample
    Dim rwLock As ReaderWriterLock
    Dim protectedValue As String
    Dim pauseThreadTwo As ManualResetEvent
    Dim ThreadOne As Thread
    Dim ThreadTwo As Thread

    Public Sub New()
        protectedValue = "Initial Value"
        rwLock = New ReaderWriterLock()
        pauseThreadTwo = New ManualResetEvent(False)
    End Sub

    Public Sub Test()
        ThreadOne = New Thread(AddressOf MethodOne)
        ThreadOne.Start()

        ThreadTwo = New Thread(AddressOf MethodTwo)
        ThreadTwo.Start()
    End Sub
```

```

Private Sub MethodOne()
    rwLock.AcquireReaderLock(1000)
    Dim seqNum As Integer = rwLock.WriterSeqNum
    Dim readValue As String = protectedValue
    pauseThreadTwo.Set()
    Thread.Sleep(1000)
    rwLock.UpgradeToWriterLock(10000)
    If (protectedValue <> readValue) Then
        Dim feedback As String
        feedback = "Value Changed:\""
        feedback += readValue
        feedback += "\" != \""
        feedback += protectedValue + "\""
        Console.WriteLine(feedback)
    End If
    rwLock.ReleaseReaderLock()
End Sub

Private Sub MethodTwo()
    pauseThreadTwo.WaitOne()
    rwLock.AcquireWriterLock(10000)
    protectedValue = "Set in Method Two"
    rwLock.ReleaseWriterLock()
End Sub

Public Sub WaitForFinished()
    ThreadOne.Join()
    ThreadTwo.Join()
End Sub

End Class

```

The output produced by listing 9.5 is:

```
Value Changed:"Initial Value" != "Set in Method Two"
```

Why not just acquire the write lock while holding the read lock? Consider the example in listing 9.6.

Listing 9.6 ReaderWriterLock Deadlock example (C#)

```

ReaderWriterLock RWLock=new ReaderWriterLock();
RWLock.AcquireReaderLock(Timeout.Infinite);
// Read some value
RWLock.AcquireWriterLock(Timeout.Infinite);
// The above instruction will not return
// Write some value
RWLock.ReleaseWriterLock();
RWLock.ReleaseReaderLock();

```

The problem is `AcquireWriteLock` will not return until it has successfully acquired a write lock and a write lock will not be granted until all read locks are released. Since the current thread has a read lock, it will never be able to acquire a write lock. This is a form of deadlock. It is unusual in that only one thread is required to form this deadlock.

Acquiring a Writer Lock To acquire a writer lock, all threads with a reader lock, including the thread requesting the write lock, must release them. `UpgradeToWriterLock` is an alternative to releasing the reader lock.

Since `AcquireWriterLock` does not consider which thread owns any outstanding reader locks, the same thread that is attempting to gain a write lock owns a read lock and will not be able to acquire the write lock.

Upgrade-To-Writer-Lock `UpgradeToWriterLock` is a method on the `ReaderWriterLock` class that allows a thread that has a read lock to convert it to a write lock, without first releasing the read lock.

An alternative might be to release the read lock before attempting to acquire the write lock. Listing 9.7 shows how this might be done.

Listing 9.7 Releasing the read lock and then acquiring a write lock (C#)

```
ReaderWriterLock RWLock=new ReaderWriterLock();
RWLock.AcquireReaderLock(Timeout.Infinite);
// Read some value
RWLock.ReleaseReaderLock();
// A different thread may change what was
// previously read during the read lock, this
// will likely result in a race condition.
RWLock.AcquireWriterLock(Timeout.Infinite);
// Change some value
RWLock.ReleaseWriterLock();
```

When `ReleaseReaderLock` is called, the read lock is released. There is no way to regain that lock; instead, a new lock will need to be acquired. The next section discusses a way of going from a read lock, to a write lock, and then back to a read lock.

9.2.3 DowngradeFromWriterLock

We know how to convert from a read to a write lock. Suppose we want to do the opposite? `UpgradeToWriterLock` returns `LockCookie`, which can be used with the `DowngradeFromWriterLock` method to change from a writer lock to a reader lock. There is no possibility of change between the time `DowngradeFromWriterLock` is called and the read lock is granted because when moving from a writer to a reader there is no chance that some other thread is already a reader, or can become one.

This is not true when moving from a reader to a writer. In order to handle possible race conditions, `UpgradeToWriterLock` uses the writer request queue. If a thread

requests a write lock, it is given the same priority as a thread that is converting from a reader lock. If the reader request queue was a priority queue, the threads that had obtained a read lock could potentially starve the threads that attempted a write lock request directly.

Listing 9.8 demonstrates downgrading from a writer to a reader lock. Note that this can only be performed if the thread originally obtained a read lock and used the `UpgradeToWriterLock` method. The cookie returned by `UpgradeToWriterLock` can only be used with `DowngradeFromWriterLock`.

Listing 9.8 Using the `DowngradeFromWriterLock` method

```
using System;
using System.Threading;
namespace Manning.Dennis
{
    public class DataUD:ThreadedTesterBase
    {
        ManualResetEvent[] interactEvents;

        public DataUD(ref Data pd,string n,string v)
            :base(ref pd,n,v)
        {
            interactEvents =new ManualResetEvent[4];
            for (int i=0;i< interactEvents .Length;i++)
            {
                interactEvents[i]=new ManualResetEvent(false);
            }
        }

        public void Interact(ActionsEnum index)
        {
            interactEvents[(int)index].Set();
            // Give the associated thread time to do its thing
            Thread.Sleep(1000);
        }

        public enum ActionsEnum
        {
            UpgradeToWrite=0,
            DowngradeToRead=1,
            ReleaseRead=2
        }

        protected override void ThreadMethod()
        {
            LockCookie cookie;
            Message("Enter");
            acquireEvent.WaitOne();
            Message("Starting Wait for Read Lock");
            protectedData.rwLock.AcquireReaderLock(Timeout.Infinite);

            Message("+++ UD- Acquired Read Lock");
            string s = protectedData.Value;
```

```

interactEvents[(int)ActionsEnum.UpgradeToWrite].WaitOne();
Message("^^^ UD- Upgrading Read Lock");
cookie=protectedData.rwLock.UpgradeToWriterLock(Timeout.Infinite);
protectedData.Value= valueToWrite;
interactEvents[(int)ActionsEnum.DowngradeToRead].WaitOne();
Message("vvv UD- Downgrading Read Lock");
protectedData.rwLock.DowngradeFromWriterLock(ref cookie);
string s2 = protectedData.Value;
interactEvents[(int)ActionsEnum.ReleaseRead].WaitOne();
Message("??? UD- Releasing Read Lock");
protectedData.rwLock.ReleaseReaderLock();
Message("---Released Read Lock");
}
}
}

```

Convert the read lock to a write lock ←
Change back to a read lock ←

One of the biggest advantages of the `DowngradeFromWriterLock` method is that it will not block. This means that it will immediately return granting the thread a read lock because there cannot possibly be a read lock at the point a write lock has been granted. Additionally, at the point the write lock is released, all pending read locks will also be released.

Listing 9.8 uses a base class that reduces the complexity of the `DataUD` class. Other classes use this base class. Listing 9.9 contains the base class code.

Listing 9.9 The base class that listing 9.8 relies on

```

using System;
using System.Threading;
namespace Manning.Dennis
{
    public abstract class ThreadedTesterBase
    {
        protected string valueToWrite;
        protected bool acquireCalled;
        protected bool interactCalled;
        protected ManualResetEvent acquireEvent;
        protected ManualResetEvent interactEvent;
        protected Data protectedData;
        protected Thread workerThread;
        protected string name;
        protected void Message(string msg)
        {
            protectedData.Message(msg);
        }
        public void Acquire()
        {
            acquireCalled = true;
            acquireEvent.Set();
        }
    }
}

```

```

        // Give the associated thread time to do its thing
        Thread.Sleep(1000);
    }
    public void Interact()
    {
        if (!acquireCalled)
        {
            throw new Exception("Call Acquire first");
        }
        interactCalled = true;
        interactEvent.Set();
        // Give the associated thread time to do its thing
        Thread.Sleep(1000);
    }
    protected ThreadedTesterBase(ref Data pd,string name,string valueToWrite)
    {
        this.valueToWrite = valueToWrite;
        acquireCalled = false;
        interactCalled = false;
        acquireEvent = new ManualResetEvent(false);
        interactEvent = new ManualResetEvent(false);
        this.protectedData = pd;
        workerThread = new Thread(new ThreadStart(ThreadMethod));
        workerThread.Name = name;
        this.name = name;

        workerThread.Start();
    }
    protected abstract void ThreadMethod();

    public void WaitForFinish()
    {
        workerThread.Join();
        // Give the associated thread time to do its thing
        Thread.Sleep(0);
    }
}
}
}

```

This base class simplifies the creation of threads used during the testing process. Listing 9.10 contains code that drives the example.

Listing 9.10 Code that demonstrates that a DowngradeFromWriterLock does not block

```

public void UpgradeDowngradeExample()
{
    Data pdata = new Data();
    DataWriter w1;
    DataWriter w2;
    DataUD udl;

```

```

        ud1= new DataUD    (ref pdata,"Upgrader1: {0}", "Upgrader1");
        w1= new DataWriter(ref pdata,"Writer_1 : {0}", "writer_1");
        w2= new DataWriter(ref pdata,"Writer_2 : {0}", "writer_2");
        Thread.Sleep(1000);
        w1.Acquire(); // acquire write lock
        ud1.Acquire();
        ud1.Interact(DataUD.ActionsEnum.UpgradeToWrite);
        w1.Interact(); // set value and release lock
        w2.Acquire(); // acquire write lock
        w2.Interact(); // set value and release lock
        ud1.Interact(DataUD.ActionsEnum.DowngradeToRead);
        ud1.Interact(DataUD.ActionsEnum.ReleaseRead);
        w1.WaitForFinish();
        Console.WriteLine("Enter to exit");
        Console.ReadLine();
    }

```

The `DataWriter` class is contained in listing 9.11.

Listing 9.11 `DataWriter` class

```

using System;
using System.Threading;
namespace Manning.Dennis
{
    public class DataWriter : ThreadedTesterBase
    {
        public DataWriter(ref Data protectedData, string name, string valueToWrite)
            : base(ref protectedData, name, valueToWrite)
        {
        }
        protected override void ThreadMethod()
        {
            Message("Enter");
            acquireEvent.WaitOne();
            Message("Starting Wait for Write Lock");
            protectedData.rwLock.AcquireWriterLock(Timeout.Infinite);
            Message("+++Acquired Writer Lock");
            interactEvent.WaitOne();
            Message("Setting value");
            protectedData.Value=valueToWrite;
            Message("???Releasing Writer Lock");
            protectedData.rwLock.ReleaseWriterLock();
            Message("---Released Writer Lock");
        }
    }
}

```

A caution regarding upgrading and downgrading reader locks: A lock should be short-lived. This will increase concurrency and decrease contention for locks. If a thread goes from being a reader to a writer and back to a reader, and stays in that state for an extended period of time, other threads will not be able to acquire a write lock. In general, locks should not be held the vast majority of the time, and only acquired when needed. The general rule of acquiring late and releasing early applies.

9.2.4 WriterSeqNum and AnyWritersSince

Suppose you wanted to know if any changes had occurred since you acquired and released a reader lock. One way to determine this is to use the `WriterSeqNum` property of the `ReaderWriterLock` object. This property returns a value that can be used with the `AnyWritersSince` method to determine if any writer locks have been released since `WriterSeqNum` was acquired.

Listing 9.12 `WriterSeqNum` can be used to see if data has changed (VB.NET).

```
Public Sub Bid(ByVal Amount As Decimal, ByVal BiddersName As String)
    Dim WriterSeqNum As Integer
    . . .
    ItemLock.AcquireReaderLock(Timeout.Infinite)
    . . .
    If (Amount > TheCurrentPrice) Then
        WriterSeqNum = ItemLock.WriterSeqNum
        ItemLock.ReleaseReaderLock()
        Thread.Sleep(1000) ' Make the changes more obvious
        ItemLock.AcquireWriterLock(Timeout.Infinite)
        If (ItemLock.AnyWritersSince(WriterSeqNum)) Then
            If (Amount > TheCurrentPrice) Then
                TheCurrentPrice = Amount
                TheBiddersName = BiddersName
            Else
                Throw New Exception("Bid not higher than current price ")
            End If
        Else
            TheCurrentPrice = Amount
            TheBiddersName = BiddersName
        End If
    Else
        Throw New Exception("Bid not higher than current price")
    End If
```

Retrieve the writer sequence number and save it

Look for new writers

In listing 9.12 we first acquire a reader lock. To simplify the code we wait indefinitely for the lock. Once the reader lock is acquired we retrieve the writer sequence number—the number of nonnested times a write lock has been acquired and released. It starts at 1 and increases by 1 each time `ReleaseWriterLock` is invoked by a thread that results in that thread no longer owning the write lock.

Writer-SeqNum `WriterSeqNum` is a property of the `ReaderWriterLock` class that returns an integer that indicates the current number of write locks acquired.

Table 9.1 demonstrates how various statements impact the values of `WriterSeqNum` along with the return value of the `AnyWritersSince` method.

Table 9.1 How Statements Impact `WriterSeqNum` Values

Statements	Any Writers Since	Writer Sequence Number	Is Read Lock Held	Is Write Lock Held
<code>Dim WSN As Integer</code>	N/A	N/A	N/A	N/A
<code>Dim RW As ReaderWriterLock = New ReaderWriterLock()</code>	N/A	1	F	F
<code>RW.AcquireReaderLock(Timeout.Infinite)</code>	N/A	1	T	F
<code>WSN = RW.WriterSeqNum</code>	F	1	T	F
<code>RW.ReleaseReaderLock()</code>	F	1	F	F
<code>RW.AcquireWriterLock(Timeout.Infinite)</code>	F	2	F	T
<code>RW.AcquireWriterLock(Timeout.Infinite)</code>	F	2	F	T
<code>RW.ReleaseWriterLock()</code>	F	2	F	T
<code>RW.ReleaseWriterLock()</code>	T	2	F	F
<code>RW.AcquireReaderLock(Timeout.Infinite)</code>	T	2	T	F
<code>WSN = RW.WriterSeqNum</code>	F	2	T	F
<code>RW.ReleaseReaderLock()</code>	F	2	F	F
<code>RW.AcquireWriterLock(Timeout.Infinite)</code>	F	3	F	T
<code>RW.AcquireWriterLock(Timeout.Infinite)</code>	F	3	F	T
<code>RW.ReleaseWriterLock()</code>	F	3	F	T
<code>RW.ReleaseWriterLock()</code>	T	3	F	F

Notice that initially the `WriterSeqNum` is one. At the point `AcquireWriterLock` executes, the value changes to 2. The method `AnyWritersSince` returns false until the second `ReleaseWriterLock` executes. This is due to the nesting of the write locks. Notice that there are two calls to `AcquireWriterLock` and two calls to `ReleaseWriterLock`. The second `ReleaseWriterLock` actually releases the lock and indicates that there has been a writer since the sequence number was acquired.

USAGE `AnyWritersSince` and `WriterSeqNum` allow for an easy way to determine if a value might have changed. It allows for a thread to cache values and increase performance. `AnyWritersSince` changes when `IsWriterLockHeld` changes from true to false. `WriterSeqNum` increases when `IsWriterLockHeld` changes from false to true.

The value of `WriterSeqNum` changes when a thread acquires a write lock for the first time while the return value for `AnyWritersSince` changes when a thread releases the write lock for the last time. Notice the correlation between the change in the return value of `AnyWritersSince` and `IsWriterLockHeld`.

9.3 **RELEASELOCK AND RESTORELOCK**

When reading a book it's nice to be able to stop, save your place, and resume. Often a bookmark is used to keep track of the current location. Similarly, the `ReaderWriterLock` class allows a thread to release its locks and later restore them. `ReleaseLock` is a method on the `ReaderWriterLock` class that allows a thread to release all locks, regardless of the nesting depth, and save the state to a lock cookie. Once the state is stored in the lock cookie, the `RestoreLock` method can be used to put the lock back to the same state it was in before `ReleaseLock` was called. Listing 9.13 demonstrates the use of `ReleaseLock` and `RestoreLock`.

Listing 9.13 The use of `ReleaseLock` and `RestoreLock` (C#)

```
static void TestSimpleReleaseLock()
{
    RW.AcquireWriterLock(Timeout.Infinite);
    LockCookie Lock = RW.ReleaseLock(); ← Saves the current
    . . .                                     lock state
    RW.RestoreLock(ref Lock ); ← Restores the state
    RW.ReleaseWriterLock();                 of the locks
}
. . .
```

It is possible that some other thread has acquired a lock during the period between `ReleaseLock` and the call to `RestoreLock`. To handle this situation the `RestoreLock` method blocks until it can acquire the required locks. Unlike the other `ReaderWriterLock` methods that acquire locks, there is no means to specify a timeout value.

ReleaseLock `ReleaseLock` is a method on the `ReaderWriterLock` that releases all currently held locks and stores the state information to a `LockCookie` structure for later restoration using the `RestoreLock` method.

The value that `ReleaseLock` has over releasing the locks using `ReleaseReaderLock` or `ReleaseWriterLock` is that it can release all locks, regardless of the nesting level, in a single call. If, for instance, a thread determined that it should die, it could call `ReleaseLock`. The alternative would be to know what sort of lock is currently held and the number of times acquire has been called.

RestoreLock `RestoreLock` is a method of the `ReaderWriterLock` class that accepts a reference to a `LockCookie` as its only parameter. `RestoreLock` blocks until it can acquire the required locks.

The following instruction releases all locks that the current thread has on the RW instance of `ReaderWriterLock`:

```
RW.ReleaseLock();
```

Instead of using the `ReleaseLock` method, the following instructions perform roughly the same function:

```
while(RW.IsReaderLockHeld)
{
    RW.ReleaseReaderLock();
}
while (RW.IsWriterLockHeld)
{
    RW.ReleaseWriterLock();
}
```

Since `ReleaseLock` returns a `LockCookie` structure we can save the current lock state for future use. During the period between `ReleaseLock` and `RestoreLock`, other threads have access to the values. This means that the values that are being protected by the `ReaderWriterLock` may have changed before `RestoreLock` is called. To handle this situation we can use the `AnyWritersSince` method we discussed in the previous section.

RestoreLock `RestoreLock` is a method of the `ReaderWriterLock` class that accepts a reference to `LockCookie` as its only parameter. `RestoreLock` blocks until it can acquire the required locks.

Listing 9.14 checks to see if some other thread has acquired a write lock since the `ReleaseLock` statement was executed.

Listing 9.14 The safe way to use `ReleaseLock` and `RestoreLock` (C#)

```
RW.AcquireWriterLock(Timeout.Infinite); ← Acquire a write lock
int SeqNum = RW.WriterSeqNum;
LockCookie Lock = RW.ReleaseLock(); ← Save the current WriterSeqNum
. . .

RW.RestoreLock(ref Lock); ← Restore the write lock
if (RW.AnyWritersSince(SeqNum)) ← Look for new writers
{
    Trace.WriteLine("A thread has written to the data");
}
else
{
    // Data has not changed since ReleaseLock
}
RW.ReleaseWriterLock();
```

This is the safest way to use the release and restore lock methods. Failure to use the `AnyWritersSince` method may result in data values changing without the knowledge of the thread that uses `RestoreLock`. If the functionality of release and restore lock is required, use care to ensure that the `ReaderWriterLock` is not bypassed.

9.4 SUMMARY

We've seen how a `ReaderWriterLock` can be used to allow multiple threads read access to a data element while preserving the integrity of the data. `ReaderWriterLocks` are a powerful construct that fit certain synchronization needs. When the situation is right, using a `ReaderWriterLock` can result in a marked performance increase.

In the next chapter we examine the `ThreadPool` class. `ThreadPools` are collections of threads that are reused to perform some short-lived task. `ThreadPools`, like `ReaderWriterLocks`, can solve certain problems very well.