期中作业

2150276 沈卓成

1 作业要求

利用 C#实现一个读写锁
（1）支持 同时 多个线程进行并发的读访问
（2）可以支持多个线程进行写 请求
（3）要求 不能出现写饥饿现象
（4）可以使用 Monitor(Enter、Exit)、Event、Semphore等基本同步原语，不要使用更高级的原语

2 实现思路

该读写锁可重入且具有非公平性，旨在解决并发读写和读写同步的问题。通过采用非同步锁实现，可以优化读取操作的效率。为了防止饥饿状态，可以设置一个阅读者数量上限来预防。该读写锁也允许一个线程多次获得同一个锁，但每次获得锁都需要相应地释放，否则将导致死锁或是死循环等问题。该读写锁中需要管理读写线程队列的调度，线程进入队列的顺序按照如下规则：多线程启动时，它们进入队列的时间由系统安排，没有一开始就定好的顺序。调度是对已经入队的线程执行阻塞和唤醒等操作。

该项目的实现使用C#中的lock基本原语，并不使用其他高级原语。

3 主要数据结构与算法

3.1 LockNode

Code：

```
using System;
using System.Threading;

namespace ReadWriteLock
{
    public class LockNode
{
    // Shared node instance
    public static readonly LockNode NODE_SHARED = new LockNode();
    // Exclusive node instance
    public static readonly LockNode NODE_EXCLUSIVE = null;
    // Threshold for read chain length
    public static int ReadThreshold = 3;

    // Node states
    public static readonly int STATE_CANCELLED = 1;
    public static readonly int STATE_RUNNING = -1;
    public static readonly int STATE_WAITING = -2;
    public static readonly int STATE_SIGNAL = -3;

    // Node waiting status
    public int status;
    // Predecessor node
    public LockNode previous;
    // Successor node
```

```
    public LockNode next;
    // Thread that owns the node
    public Thread currentThread;

    // The following are applicable only for read nodes
    // Head of the read chain
    public LockNode headReader;
    // Next read node in the chain
    public LockNode subsequentReader;
    // Length of the read chain
    public int countReaders = 1;

    // Node type (shared or exclusive)
    public LockNode nodeType;

    // Constructor for creating a node
    public LockNode()
    {
    }

    // Constructor for shared or exclusive node
    public LockNode(Thread threadToHold, LockNode type)
    {
        this.nodeType = type;
        this.currentThread = threadToHold;
    }

    // Method to check if node is shared
    public bool isNodeShared()
    {
        return nodeType == NODE_SHARED;
    }

    // Method to get status description
    public static string DescribeStatus(int statusCode)
    {
        switch (statusCode)
        {
            case 1:
                return "CANCELLED";
            case -1:
                return "RUNNING";
            case -2:
                return "WAITING";
            case -3:
                return "SIGNAL";
            default:
                return "UNKNOWN";
        }
    }
}
}
```

这段代码定义了一个名为`LockNode`的类，主要用于表示锁机制中的节点。这个类通过不同的变量和方法，实现了对锁节点的管理和状态追踪。

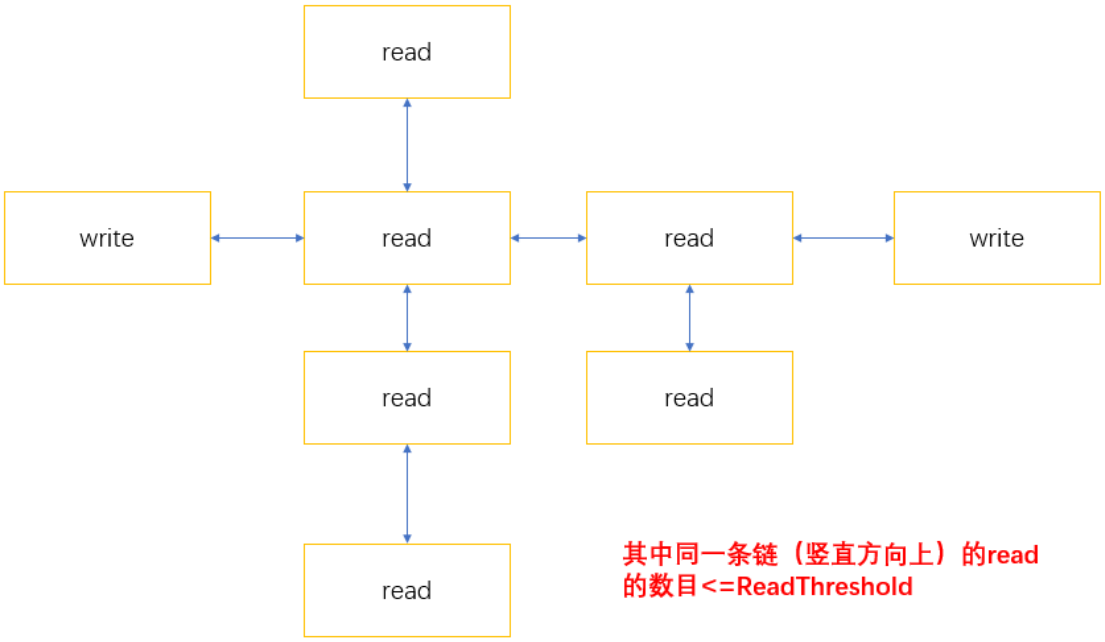
变量和常量

- `NODE_SHARED` 和 `NODE_EXCLUSIVE`：这两个静态只读变量分别代表共享节点和独占节点的实例。共享节点意味着多个线程可以同时访问相同的资源，而独占节点则表示只有单个线程可以访问资源。
- `ReadThreshold`：一个静态整型变量，定义了读链的长度阈值。读链是一种数据结构，用来管理那些尝试读取共享资源的线程。
- `STATE_CANCELLED`、`STATE_RUNNING`、`STATE_WAITING` 和 `STATE_SIGNAL`：这些静态只读整型变量定义了节点可能的状态，包括被取消、正在运行、等待和信号状态。
- `status`：整型变量，表示节点的当前等待状态。
- `previous` 和 `next`：`LockNode` 类型的变量，分别指向当前节点的前驱节点和后继节点。
- `currentThread`：`Thread` 类型的变量，表示持有当前节点的线程。
- `headReader`、`subsequentReader` 和 `countReaders`：这些变量特定于读节点，分别表示读链的头节点、后继读节点和读链长度。
- `nodeType`：`LockNode` 类型的变量，表示节点的类型（共享或独占）。

方法

- 构造函数：类中定义了两个构造函数。第一个是默认的构造函数，没有参数。第二个构造函数接受两个参数：`threadToHold`（表示将持有该节点的线程）和 `type`（节点类型，共享或独占）。
- `isNodeShared`：这是一个布尔方法，用于检查当前节点是否为共享节点。
- `DescribeStatus`：这是一个静态方法，接受一个整型参数 `statusCode`，根据传入的状态码返回相应的状态描述字符串。这个方法使用了 `switch` 语句来匹配不同的状态码。

整个队列的结构如下：



3.2 关键辅助方法

3.2.1 SignalNext

Code:

```
private void SignalNext()
        {
            LockNode node = GetNextNode();
```

```
            if (node == null)
            {
                listTail = null;
                currentOwner = null;
                entryCount = 0;
                return;
            }
            else
            {
                node.previous = listHead;
                entryCount = node.countReaders;
                while (node != null)
                {
                    node.status = LockNode.STATE_SIGNAL;
                    node = node.subsequentReader;
                    if (node != null)
                        node.previous = listHead;
                }
            }
        }
```

这段代码定义了一个`SignalNext`方法，它的目的是在一个读写锁的实现中向等待的线程发送信号，以便它们可以开始执行它们的读或写操作。

1. **获取下一个节点**: 首先，通过调用`GetNextNode()`方法获取链表中的下一个待处理的`LockNode`节点。
2. **检查节点是否为空**: 如果`node`为`null`，这意味着没有更多的节点等待处理。此时，将`listTail`和`currentOwner`设置为`null`，并将`entryCount`重置为0，然后返回。这表示锁当前没有被任何线程占用。
3. **处理非空节点**: 如果`node`不为`null`，则进行以下操作：
    - 将该节点的`previous`指向链表的头节点`listHead`。这可能是为了重新组织链表，准备将控制权转移给这个节点。
    - 将`entryCount`更新为该节点的`countReaders`值，反映等待在这个节点上的读操作的数量。
    - 进入一个循环，遍历所有的后续读者（通过`node.subsequentReader`访问）。对于每个这样的节点，设置其状态为`STATE_SIGNAL`，这表示它现在可以开始执行它的操作（读或写）。如果存在后续读者，还会将它们的`previous`指向`listHead`。

这个方法是读写锁同步机制的一部分，用于在一个线程完成其操作（读或写）并释放锁之后，唤醒等待中的线程，使它们可以开始它们的操作。通过将节点状态设置为`STATE_SIGNAL`，它告知这些节点它们现在可以安全地访问共享资源。这是确保在多线程环境下维持数据一致性和线程安全的关键机制之一。

3.2.2 GetReaderNode

Code:

```
private LockNode GetReaderNode(LockNode node)
        {
            LockNode subsequentReader = node.headReader;
            while (subsequentReader != null)
            {
                if
(subsequentReader.currentThread.Name.Equals(Thread.CurrentThread.Name))
                    return subsequentReader;
                else
                    subsequentReader = subsequentReader.subsequentReader;
            }
            return null;
        }
```

1. **初始化**: 首先，它获取 `node` 的 `headReader` 属性，这个属性指向当前节点的读者链的头部。这个头部是一个 `LockNode` 类型的对象，表示第一个等待读取的线程。
2. **遍历读者**: 然后它进入一个循环，遍历所有的读者节点。这是通过 `while (subsequentReader != null)` 实现的，意味着它将一直循环，直到没有更多后续的读者节点。
3. **匹配当前线程**: 在循环内部，它检查每个读者节点的 `currentThread` 属性（代表当前正在该节点上等待的线程）是否与当前执行代码的线程名字相同。这是通过 `subsequentReader.currentThread.Name.Equals(Thread.CurrentThread.Name)` 实现的。如果找到匹配的线程，它返回相应的读者节点。
4. **获取下一个读者**: 如果当前检查的节点不是当前线程，则通过 `subsequentReader = subsequentReader.subsequentReader` 移动到读者链中的下一个节点。
5. **返回结果**: 如果在链表中找到了匹配的节点，方法返回这个节点。如果没有找到匹配的节点（也就是没有读者节点的线程名与当前线程的名字相同），则返回 `null`。

这个方法在锁机制中很有用，特别是在需要确定当前线程是否已经在读者链中有一个节点，或者需要获取当前线程的读者节点以进行进一步处理时。这个方法用于在一个由 `LockNode` 组成的链表中找到与当前执行线程相对应的节点。

3.2.3 IsReadChainDone

Code:

```
private bool IsReadChainDone(LockNode node)
        {
            LockNode subsequentReader = node.headReader;
            while (subsequentReader != null)
            {
                if (subsequentReader.status != LockNode.STATE_CANCELLED)
                    return false;
                else
                    subsequentReader = subsequentReader.subsequentReader;
            }
            return true;
        }
```

这段代码定义了一个名为 `IsReadChainDone` 的方法，它用于确定一个读锁节点（`LockNode`）链表中的所有读者是否都已完成其操作。这个方法接受一个 `LockNode` 类型的参数 `node`，它通常代表锁链表中的一个节点。方法的逻辑如下：

1. **初始化**: 首先，方法获取 `node` 的 `headReader` 属性，这个属性指向当前节点的读者链的头部节点。头部节点是一个 `LockNode` 对象，代表等待读取的第一个线程。

2. **遍历读者**: 然后，它进入一个循环，遍历所有的读者节点。这是通过 `while (subsequentReader != null)` 实现的，意味着循环会一直进行，直到没有更多的后续读者节点。
3. **检查状态**: 对于每个读者节点，方法检查其 `status` 属性。如果任何一个读者节点的状态不等于 `LockNode.STATE_CANCELLED`，这意味着至少有一个读者节点没有取消，也就是说读链还没有完全处理完毕，因此方法返回 `false`。
4. **获取下一个读者**: 如果当前读者节点已被取消（状态等于 `LockNode.STATE_CANCELLED`），则通过 `subsequentReader = subsequentReader.subsequentReader` 指向后续的读者节点继续遍历。
5. **返回结果**: 如果所有的读者节点都已被取消，即读链已经处理完毕，那么循环会顺利结束，方法返回 `true` 表示读链已经完成。

`IsReadChainDone` 方法的作用是遍历特定节点的读者链表，检查是否所有的读者都已不再活跃（即它们的状态都是已取消）。如果所有读者都已完成或取消，那么这个方法返回 `true`，否则返回 `false`。这种检查通常用于在释放读锁或者在写锁请求时确定是否可以安全地将控制权传递给写线程。

3.2.4 CanReenterShared

Code:

```
private bool CanReenterShared()
        {
            LockNode node = listHead.next;
            if (node == null || !node.isNodeShared())
                return false;
            else
            {
                LockNode reader = node;
                while (reader != null)
                {
                    if (reader.currentThread == Thread.CurrentThread)
                    {
                        return true;
                    }
                    reader = reader.subsequentReader;
                }
            }
            return false;
        }
```

`CanReenterShared` 用于检查当前执行代码的线程是否可以再次进入（重新获得）一个共享锁。在读写锁实现中，这通常是指线程已经持有了一个共享锁，并且想要再次进入读取操作，不需要等待其他线程释放锁。

1. **获取第一个节点**: 方法首先获取链表中的第一个节点（`listHead.next`），这通常是链表中的第一个等待或持有锁的节点。
2. **检查是否可以重入**:
    - 如果链表的第一个节点是空的（`node == null`），或者第一个节点不是共享节点（`!node.isNodeShared()`），则方法返回 `false`，表示当前线程不能再次进入共享锁，因为没有共享锁存在或者第一个锁不是共享的。
    - 如果第一个节点存在，且是一个共享锁节点，方法继续检查链表中的节点。
3. **遍历共享锁节点**: 方法进入一个循环，遍历所有的共享锁节点（读者链）。
    - 如果在读者链中找到一个节点，其 `currentThread` 属性（代表持有该节点的线程）与当前执行代码的线程（`Thread.CurrentThread`）相同，返回 `true`，表示当前线程已经持有共享锁，并且可以再次进入。

- 如果当前节点不是由当前线程持有，方法继续遍历链表中的后续节点（通过 `reader = reader.subsequentReader`）。

4. **返回结果**: 如果方法在读者链中没有找到当前线程持有的节点，则返回 `false`。

3.3 读操作

获取锁：

Code:

```
public void LockShared()
        {
            LockNode reader = null;
            lock (this)
            {
                if (listHead == null)
                {
                    listHead = new LockNode();
                }
            }
            Thread currentThread = Thread.CurrentThread;
            lock (this)
            {
                if (listTail == null)
                {
                    AppendWhenTailEmpty(LockNode.NODE_SHARED);
                    return;
                }
            }
            LockNode current = null;
            lock (this)
            {
                if (CanReenterShared())
                {
                    entryCount += 1;
                    return;
                }
                LockNode node = listHead.next;
                if (node == null)
                {
                    AppendWhenTailEmpty(LockNode.NODE_SHARED);
                    return;
                }
                else
                {
                    if (WriterExists())
                    {
                        reader = new LockNode(currentThread,
LockNode.NODE_SHARED);
                        while (node != null && (!node.isNodeShared() ||
node.status == LockNode.STATE_CANCELLED))
                        {
                            if (node.isNodeShared())
                            {
                                if (!IsReadChainDone(node))
                                    break;
                                else
```

```
                                node = node.next;
                            }
                            else
                                node = node.next;
                        }
                        if (node == null)
                        {
                            current = Append(LockNode.NODE_SHARED);
                        }
                        else
                        {
                            if (node.countReaders < LockNode.ReadThreshold)
                            {
                                current = AddToReadChain(node, reader);
                                if (node.status == LockNode.STATE_RUNNING)
                                {
                                    reader.status = LockNode.STATE_RUNNING;
                                    entryCount += 1;
                                    return;
                                }
                            }
                            else
                            {
                                LockNode t = listTail;
                                if (!t.isNodeShared() || (t.isNodeShared() &&
t.countReaders >= LockNode.ReadThreshold))
                                    current = Append(LockNode.NODE_SHARED);
                                else
                                {
                                    current = AddToReadChain(t, reader);
                                    if (t.status == LockNode.STATE_RUNNING)
                                    {
                                        reader.status = LockNode.STATE_RUNNING;
                                        entryCount += 1;
                                        return;
                                    }
                                }
                            }
                        }
                    }
                    else
                    {
                        if (listTail == null)
                        {
                            AppendWhenTailEmpty(LockNode.NODE_SHARED);
                            return;
                        }
                        if (listTail.countReaders < LockNode.ReadThreshold)
                        {
                            reader = new LockNode(currentThread,
LockNode.NODE_SHARED);
                            current = AddToReadChain(listTail, reader);
                            if (listTail.status == LockNode.STATE_RUNNING)
                            {
                                reader.status = LockNode.STATE_RUNNING;
                                entryCount += 1;
                                return;
                            }
```

```
                        }
                        else
                        {
                            current = Append(LockNode.NODE_SHARED);
                        }
                    }
                }
            }
            while (current.status != LockNode.STATE_SIGNAL) { }
            current.status = LockNode.STATE_RUNNING;
            entryCount = current.headReader.countReaders;
            currentOwner = current.headReader == null ? null :
current.headReader.currentThread;
        }
```

这段代码是一种在多线程环境中用于管理读写锁的逻辑。它的主要目的是允许多个线程同时读取资源，但在写入资源时只允许一个线程。这是通过使用锁节点(`LockNode`)来实现的，其中每个节点代表一个正在尝试读取或写入资源的线程。

1. **初始化**: 首先，代码检查是否有头节点(`listHead`)，如果没有，则创建一个新的`LockNode`作为头节点。
2. **当前线程获取**: 获取当前线程的引用，并存储在`currentThread`变量中。
3. **重入检查**: 如果当前线程已经获得了共享锁，它可以再次进入而不会被阻塞。
4. **查找位置和处理逻辑**:
    - 如果链表中没有写入者(`WriterExists()`返回`false`)，并且尾节点(`listTail`)不存在或者尾节点的读者计数少于阈值(`ReadThreshold`)，则当前线程会尝试将自己加入到读链中(`AddToReadChain`)或者作为一个新的节点追加到链表中(`Append`)。
    - 如果链表中存在写入者，当前线程会在链表中寻找一个合适的位置插入一个共享节点，以便可以共享读取。它会跳过状态为取消(`STATE_CANCELLED`)的节点，并根据读链的完成情况(`IsReadChainDone`)和读者计数来确定插入位置。
5. **等待信号**: 一旦当前线程被加入到链表中，它会等待直到其状态变为`STATE_SIGNAL`，这意味着它现在可以安全地读取资源。
6. **更新状态和所有者**: 当线程开始运行时，它更新自己的状态为`STATE_RUNNING`，并更新`entryCount`为当前读链头节点的读者计数。同时，它还更新资源的当前所有者为读链头节点的线程（如果存在）。

在此逻辑中，`LockNode`是一个关键的数据结构，它包含了线程的信息、节点状态、以及是否是共享节点等信息。状态管理（如`STATE_RUNNING`, `STATE_SIGNAL`等）对于线程之间的同步至关重要。

这种机制允许多个读操作并发执行，只要没有线程正在写入。当有线程需要写入时，它会等待直到所有的读操作完成，这样就保证了数据的一致性和线程安全。

释放锁：

Code：

```
public void UnlockShared()
        {
            lock (this)
            {
                entryCount -= 1;
                LockNode node = listHead.next;
```

```
                while (node != null)
                {
                    if (IsReadChainDone(node))
                    {
                        node = node.next;
                        listHead.next = node;
                        if (node != null)
                            node.previous = listHead;
                    }
                    else
                        break;
                }
                if (node != null)
                {
                    node = GetReaderNode(node);
                    node.status = LockNode.STATE_CANCELLED;
                    if (entryCount == 0)
                        SignalNext();
                }
            }
        }
```

这段代码是`UnlockShared`方法的实现，它被用于释放一个共享锁。在多线程环境中，这允许一个线程表示它已经完成了对共享资源的读取操作。

1. **锁定**: 该方法首先通过`lock (this)`语句对当前对象进行锁定，以确保在修改共享变量（如`entryCount`、链表等）时线程安全。
2. **减少入口计数**: 通过`entryCount -= 1;`将入口计数减少1，表示有一个线程释放了它的读锁。
3. **遍历链表**: 方法接下来会遍历锁链表，从头节点的下一个节点开始，寻找不再参与读操作的节点。
4. **清理读链**: 如果发现一个读链已经完成（即没有线程正在读取或等待读取），通过`IsReadChainDone(node)`检查，那么它会从链表中移除这个读链。这通过移动`listHead.next`到下一个节点并更新节点的`previous`指针来实现。
5. **取消读节点**: 当找到一个读节点时，使用`GetReaderNode(node)`获取这个节点，并将其状态设置为`STATE_CANCELLED`，表示这个节点不再参与读操作。
6. **信号下一个节点**: 如果`entryCount`变为0，说明没有线程持有读锁，调用`SignalNext()`方法来唤醒等待链表中的下一个节点（可能是一个写入者）。

这段代码的目的是确保当一个线程完成读取操作时，它能够正确地通知其他可能正在等待写入的线程。这样，写入者可以在资源上获取独占访问权，从而维护数据的一致性和完整性。通过维护一个链表来跟踪哪些节点（线程）正在读取或等待读取，该方法可以确保在多线程环境下对共享资源的安全访问。

3.4 写操作

获得锁：

Code:

```
public void LockExclusive()
        {
            LockNode current = null;
            lock (this)
            {
                if (listHead == null)
                {
                    listHead = new LockNode();
                }
```

```
            }
            lock (this)
            {
                if (listTail == null)
                {
                    AppendWhenTailEmpty(LockNode.NODE_EXCLUSIVE);
                    return;
                }
            }
            lock (this)
            {
                if (currentOwner == Thread.CurrentThread)
                {
                    entryCount += 1;
                    return;
                }
                else
                {
                    LockNode t = listTail;
                    if (t == null)
                    {
                        AppendWhenTailEmpty(LockNode.NODE_EXCLUSIVE);
                        return;
                    }
                    else
                        current = Append(LockNode.NODE_EXCLUSIVE);
                }
            }
            while (current.status != LockNode.STATE_SIGNAL) { }
            entryCount = 1;
            current.status = LockNode.STATE_RUNNING;
            currentOwner = Thread.CurrentThread;
        }
```

这段代码定义了一个名为 `LockExclusive` 的方法，它的目的是让当前执行代码的线程获取一个独占锁。在读写锁实现中，独占锁允许持有它的线程独占对共享资源的访问，而不允许其他线程同时读取或写入。

1. **初始化节点**: 首先，定义一个 `LockNode` 类型的局部变量 `current`。
2. **检查头节点**: 使用 `lock (this)` 语句创建一个同步块，确保线程安全地访问 `listHead`。如果 `listHead` 为空，说明链表为空，它会创建一个新的 `LockNode` 对象并将其赋值给 `listHead`。
3. **检查尾节点**: 再次使用 `lock (this)` 创建另一个同步块，检查 `listTail` 是否为空。如果为空，则调用 `AppendWhenTailEmpty` 方法添加一个独占节点并直接返回，因为没有其他节点等待。
4. **检查当前所有者**:
   - 再次进入一个 `lock (this)` 块来保证线程安全。
   - 如果当前线程（`Thread.CurrentThread`）已经是锁的所有者（`currentOwner`），则只需将 `entryCount`（表示当前线程重入锁的次数）增加1，然后返回。
   - 如果当前线程不是锁的所有者，检查 `listTail` 是否为空。如果为空，调用 `AppendWhenTailEmpty` 方法添加一个独占节点并返回。
   - 如果 `listTail` 不为空，调用 `Append` 方法添加一个独占锁节点，并将返回的节点赋值给 `current`。

5. **等待信号**: 接下来，使用 `while` 循环等待 `current` 节点的状态变为 `LockNode.STATE_SIGNAL`。这个循环会一直执行，直到当前节点的状态被其他线程（可能是释放锁的线程）更改为 `STATE_SIGNAL`，表示当前线程可以获取锁。
6. **获取锁**: 一旦跳出循环，设置 `entryCount` 为1，表明当前线程现在持有锁，并将 `current` 节点的状态设置为 `LockNode.STATE_RUNNING`，表示当前线程正在运行。
7. **设置当前所有者**: 最后，将 `currentOwner` 设置为当前线程，表明当前线程现在拥有独占锁。

释放锁：

Code：

```
public void UnlockExclusive()
        {
            lock (this)
            {
                entryCount -= 1;
                LockNode node = GetNextNode();
                if (node != null)
                {
                    if (entryCount == 0)
                    {
                        node.status = LockNode.STATE_CANCELLED;
                        SignalNext();
                    }
                }
            }
        }
```

这段代码定义了一个名为 `UnlockExclusive` 的方法，它用于释放当前线程持有的独占锁。这个方法通常在读写锁实现中使用，当一个线程完成了它对共享资源的独占访问，并希望允许其他线程访问该资源时调用。

1. **锁定对象**: 使用 `lock (this)` 语句创建一个同步块，这用于确保在多线程环境中线程安全地修改对象的状态。
2. **减少入口计数**: `entryCount` 是一个计数器，它跟踪当前线程重入独占锁的次数。每次调用 `UnlockExclusive` 时，`entryCount` 都会减少 1。
3. **获取下一个节点**: 调用 `GetNextNode` 方法来获取锁链表中的下一个节点。
4. **检查和处理节点**: 如果找到下一个节点（`node != null`）：
    - 检查 `entryCount` 是否为 0，这意味着当前线程已经完全释放了独占锁。
    - 如果 `entryCount` 为 0，将找到的节点的状态设置为 `LockNode.STATE_CANCELLED`，表示该节点不再有效。
    - 然后调用 `SignalNext` 方法，这个方法会唤醒等待链表中下一个节点的线程，允许它们尝试获取锁。
5. **释放锁**: 如果 `entryCount` 减到 0，这意味着独占锁被完全释放了，`SignalNext` 会触发锁的状态变化，允许等待的线程继续执行。

4 测试与验证

Code：

```
namespace Synchronization
{
```

```
    class Program
    {
        static readonly SharedExclusiveLock sharedExclusiveLock = new
SharedExclusiveLock();
        static readonly int readerCount = 5;
        static readonly int writerCount = 2;

        static void Main(string[] args)
        {
            // Start reader threads.
            for (int i = 0; i < readerCount; i++)
            {
                Thread readerThread = new Thread(ReaderThread) { Name =
$"Reader-{i+1}" };
                readerThread.Start();
            }

            // Start writer threads.
            for (int i = 0; i < writerCount; i++)
            {
                Thread writerThread = new Thread(WriterThread) { Name =
$"Writer-{i+1}" };
                writerThread.Start();
            }

            Console.ReadKey();
        }

        static void ReaderThread()
        {
            while (true)
            {
                sharedExclusiveLock.LockShared();
                Console.WriteLine($"{Thread.CurrentThread.Name} has acquired the
shared lock.");
                // Simulate reading activity.
                Thread.Sleep(100);
                Console.WriteLine($"{Thread.CurrentThread.Name} has released the
shared lock.");
                sharedExclusiveLock.UnlockShared();

                // Simulate time between reading.
                Thread.Sleep(1000);
            }
        }

        static void WriterThread()
        {
            while (true)
            {
                sharedExclusiveLock.LockExclusive();
                Console.WriteLine($"{Thread.CurrentThread.Name} has acquired the
exclusive lock.");
                // Simulate writing activity.
                Thread.Sleep(500);
                Console.WriteLine($"{Thread.CurrentThread.Name} has released the
exclusive lock.");
                sharedExclusiveLock.UnlockExclusive();
```

```
                // Simulate time between writing.
                Thread.Sleep(2000);
            }
        }
    }


}
```

这段代码在 `Synchronization` 命名空间下定义了一个名为 `Program` 的类，该类在其主函数中创建并启动了一系列的读者和写者线程，以测试 `SharedExclusiveLock` 类的工作方式，模拟了一个典型的读写锁的应用场景。测试这段代码的思路主要包括以下几个方面：

1. 验证锁的基本功能

- **共享锁 (Read Lock) 的测试**：确保多个读者（reader）线程可以同时持有共享锁，进行读取操作。这可以通过观察控制台输出来验证，看是否有多个读者线程同时报告它们已获得共享锁。
- **独占锁 (Write Lock) 的测试**：验证当一个写者（writer）线程请求并获得独占锁时，其他线程（无论是读者还是写者）都不能获得锁。这同样可以通过控制台输出来观察，注意写者线程获取独占锁期间，其他线程是否被正确阻塞。

2. 锁的公平性和性能

- **公平性**：检查读者和写者线程是否都有机会按照它们请求锁的顺序获得锁。虽然从这段代码直接观察不容易得出结论，但可以通过记录和分析每个线程获取和释放锁的时间点来评估。
- **性能**：评估在高并发情况下锁的性能。注意观察系统是否出现性能瓶颈，比如因为锁竞争导致的线程饥饿现象。

3. 死锁的可能性

- **死锁检测**：虽然代码设计上应该避免了死锁（通过确保共享锁和独占锁的逻辑分离），测试时仍需仔细观察是否存在死锁情况，特别是在读写线程频繁交替运行时。

4. 锁的重入性

- 如果 `SharedExclusiveLock` 实现了锁的重入性（代码中未显示实现的部分），可以测试线程是否能重入它已经持有的锁，无论是共享锁还是独占锁。

5. 多线程下的稳定性

- **长时间运行测试**：让程序运行较长时间，观察是否有异常发生，比如线程崩溃或资源泄露。

测试结果（这里只截取部分控制台输出）：

```
Reader-2 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-2 has released the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
```

```
Reader-1 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-4 has released the shared lock.
Reader-1 has released the shared lock.
Reader-3 has released the shared lock.
Reader-1 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-4 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-1 has released the shared lock.
Reader-3 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-4 has released the shared lock.
Reader-3 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-1 has released the shared lock.
Reader-5 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-4 has released the shared lock.
Reader-5 has released the shared lock.
Reader-2 has released the shared lock.
Reader-3 has released the shared lock.
Reader-1 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-5 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has released the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
```

```
Reader-2 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-4 has released the shared lock.
Reader-3 has released the shared lock.
Reader-2 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-5 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has released the shared lock.
Reader-2 has released the shared lock.
Reader-4 has released the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-5 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-3 has released the shared lock.
Reader-1 has released the shared lock.
Reader-4 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-4 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-5 has released the shared lock.
Reader-2 has released the shared lock.
Reader-1 has released the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Reader-4 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-3 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-1 has released the shared lock.
Reader-4 has released the shared lock.
```

```
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-4 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-4 has released the shared lock.
Reader-3 has released the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-2 has released the shared lock.
Reader-2 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-4 has released the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-2 has released the shared lock.
Reader-3 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-4 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-4 has released the shared lock.
Reader-3 has released the shared lock.
Reader-2 has released the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-2 has released the shared lock.
Reader-1 has released the shared lock.
Reader-5 has released the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-4 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-1 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-5 has released the shared lock.
```

```
Reader-1 has released the shared lock.
Reader-2 has released the shared lock.
Reader-4 has released the shared lock.
Reader-3 has released the shared lock.
Reader-1 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-1 has released the shared lock.
Writer-2 has acquired the exclusive lock.
Writer-2 has released the exclusive lock.
Writer-1 has acquired the exclusive lock.
Writer-1 has released the exclusive lock.
Reader-1 has acquired the shared lock.
Reader-4 has acquired the shared lock.
Reader-5 has acquired the shared lock.
Reader-3 has acquired the shared lock.
Reader-2 has acquired the shared lock.
Reader-3 has released the shared lock.
Reader-4 has released the shared lock.
Reader-2 has released the shared lock.
Reader-5 has released the shared lock.
Reader-1 has released the shared lock.
```

可以看到，该读写锁支持多个线程的并发读访问与多个线程的写访问，同时避免了写饥饿以及死锁。