Algorithms and Data Structures

Collection Concurrency

Robert Horvick

www.pluralsight.com

Outline

- **Concurrency**
  - Overview
  - Problems
- **Solutions**
  - Caller Synchronization
  - Monitor Locking
  - Read/Write Locking
- **Concurrent Collections in .NET**

Concurrency Overview

- **Multiple instructions execute at the same time (concurrently)**
- **Multiple Threads**
  - Several threads executing within a single process
  - Example: Multi-threaded .NET application accessing a shared collection
- **Multiple Processes**
  - Several processes executing on a single computer system
  - Example: Multiple applications accessing a common file
- **Multiple Systems**
  - Several systems, each accessing a common resource
  - Example: Multiple processes accessing a common database table

Single Threaded Execution

- **Example: Processing one job at a time**
  - Jobs created and queued
  - Jobs processed one at a time

```
Queue<Job> pendingJobIds = new Queue<Job>();
for (int jobId = 0; jobId < 1000; jobId++)
{
    pendingJobIds.Enqueue(new Job(jobId));
}

while (pendingJobIds.Count > 0)
{
    pendingJobIds.Dequeue().Process();
}
```

```
class Job
{
    int _jobId;

    public Job(int jobId)
    {
        _jobId = jobId;
    }

    public void Process()
    {
        Thread.SpinWait(500);
    }
}
```

Multi Threaded Execution

- **Example: Processing multiple jobs at a time**
  - Jobs created and queued
  - Multiple jobs processed in parallel
- **Changes**
  - Multiple threads created to run the job processing
  - Job class modified to detect multiple execution
- **Concurrency Issues**
  - Race conditions arise when multiple callers access the queue at once

Multi Threaded Execution

```
Queue<Job> pendingJobIds = new Queue<Job>();
for (int jobId = 0; jobId < 100000; jobId++)
{
    pendingJobIds.Enqueue(new Job(jobId));
}
```

```
Thread[] processingThreads = new Thread[2];

// Delegate to process the threads
ThreadStart runJobs = delegate()
{
    while (pendingJobIds.Count > 0)
    {
        pendingJobIds.Dequeue().Process();
    }
};
```

```
// Create and start the threads
for (int i = 0; i < processingThreads.Length; i++)
{
    processingThreads[i] = new Thread(runJobs);
    processingThreads[i].Start();
}
```

```
// Wait until all the threads are done
foreach (Thread t in processingThreads)
{
    t.Join();
}
```

```
class Job
{
    int _jobId;
    bool _processed;
    object _syncLock = new object();

    public Job(int jobId)
    {
        _jobId = jobId;
    }

    public void Process()
    {
        lock (_syncLock)
        {
            if (_processed)
            {
                throw new InvalidOperationException(
                    string.Format(
                    "{0} executed multiple times",
                    _jobId));
            }

            Thread.SpinWait(500);
            _processed = true;
        }
    }
}
```

What's the Problem?

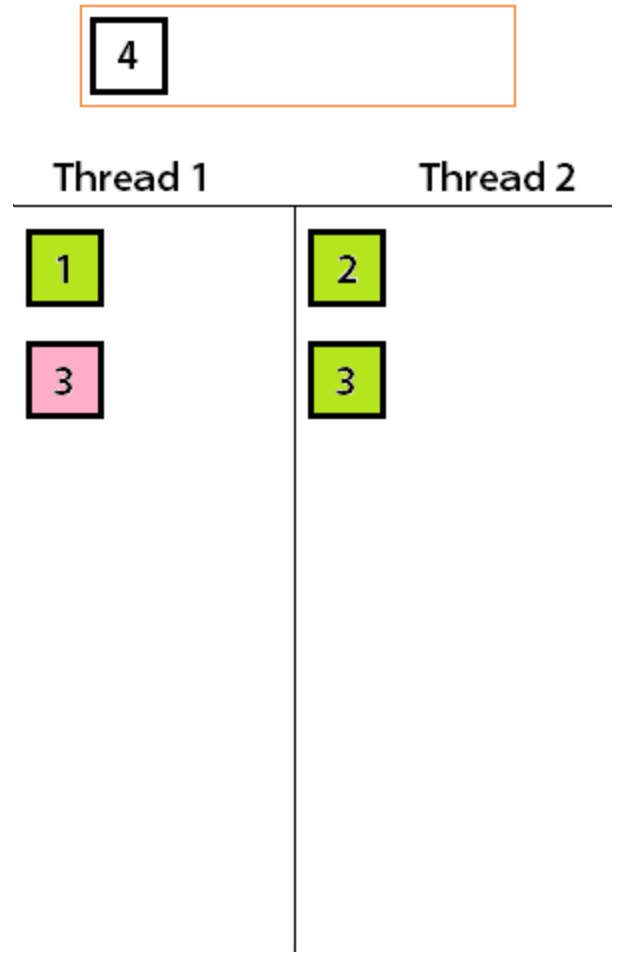
```
// Delegate to process the threads
ThreadStart runJobs = delegate()
{
    while (pendingJobIds.Count > 0)
    {
        pendingJobIds.Dequeue().Process();
    }
};


public T Dequeue()
{
    T value = _items[_head];

    if (_head == _items.Length - 1)
    {
        // if the head is at the last
        // index in the array - wrap around.
        _head = 0;
    }
    else
    {
        // move to the next value
        _head++;
    }

    _size--;

    return value;
}
```



Caller Synchronization

- **The collection defers the responsibility for thread safety to the caller**
- **Simplified collection design**
- **No overhead when being used in a non-concurrent manner**
- **Synchronization can be performed by any means**
  - Monitor (lock in C#)
  - ReaderWriterLockSlim (not ReaderWriterLock!)
  - Mutex
  - Semaphore
  - Other

Caller Synchronization with Monitor

- **Monitor (lock) is used to protect the collection from race conditions**
- **Pros**
  - Non-threadsafe collections can be used safely in a multi-threaded environment
- **Cons**
  - Caller is responsible for all thread safety
  - Readers will block other readers

Caller Synchronization Code

- A lock object is allocated

```
static object _pendingJobsLock = new object();
```

- The lock protects the call to Dequeue
- This has a bug and a performance problem!
  - Count needs to be included in the Dequeue lock scope
  - The lock is held while Process is called

```
// Delegate to process the threads
ThreadStart runJobs = delegate()
{
    while (pendingJobIds.Count > 0)
    {
        Job jobToProcess = null;
        lock (_pendingJobsLock)
        {
            if (pendingJobIds.Count > 0)
            {
                jobToProcess = pendingJobIds.Dequeue();
            }
        }

        if (jobToProcess != null)
        {
            jobToProcess.Process();
        }
    }
};
```

Collection Synchronization (Monitor)

- **The collection manages thread-safety by using Monitors**
- **Locking is done at the method or property level**
  - Peek
  - Enqueue
  - Dequeue
  - Etc
- **Pros**
  - The caller does not need to implement method-level locking
- **Cons**
  - It is deceptively safe
  - Readers block other readers

Monitor Lock Code

- **Lock object added to class**

```
readonly object syncLock = new object();
```

- **Lock entered when a non-thread-safe method is called (e.g., Dequeue)**
- **Deceptively safe**
  - Just because a collection is thread-safe does not mean that all usage patterns will be thread safe. For example the following uses multiple locks so the decision cannot be trusted.

```
while (pendingJobIds.Count > 0)
{
    pendingJobIds.Dequeue().Process();
}
```

```
public T Dequeue()
{
    lock (syncLock)
    {
        if (_size == 0)
        {
            throw new QueueEmptyException();
        }

        T value = _items[_head];

        if (_head == _items.Length - 1)
        {
            // if the head is at the last
            // index in the array - wrap around.
            _head = 0;
        }
        else
        {
            // move to the next value
            _head++;
        }

        _size--;

        return value;
    }
}
```

Collection Synchronization (Reader Writer Lock)

- **The collection manages thread-safety by using a reader writer lock**
  - In .NET this should be the ReaderWriterLockSlim type
- **Locking is done at the method or property level**
  - Same as Monitor
- **Pros**
  - The caller does not need to implement method-level locking
  - Readers do not block other Readers
- **Cons**
  - It is deceptively safe
  - More overhead than Monitor
    - Performance Note: This is especially noticeable in the fast read operations, such as Peek, because the lock cost becomes relatively significant

Reader Writer Lock Code

- **Lock object added to class**

```
ReaderWriterLockSlim rwLock = new ReaderWriterLockSlim();
```

- **Lock entered when a non-thread-safe method is called (e.g., Dequeue)**
- **Deceptively safe**
  - Just because a collection is thread-safe does not mean that all usage patterns will be thread safe. For example the following uses multiple locks so the decision cannot be trusted.

```
while (pendingJobIds.Count > 0)
{
    pendingJobIds.Dequeue().Process();
}
```

```
public T Dequeue()
{
    rwLock.EnterWriteLock();
    try
    {
        T value = default(T);

        if (_size == 0)
        {
            throw new QueueEmptyException();
        }

        value = _items[_head];

        if (_head == _items.Length - 1)
        {
            // if the head is at the last
            // index in the array - wrap around.
            _head = 0;
        }
        else
        {
            // move to the next value
            _head++;
        }

        _size--;

        return value;
    }
    finally
    {
        rwLock.ExitWriteLock();
    }
}
```

.NET Framework

- **.NET 4 added several concurrent collections**
  - ConcurrentDictionary<TKey, TValue>
  - ConcurrentQueue<T>
  - ConcurrentStack<T>
  - ConcurrentBag<T>
- **These are not drop-in replacements for existing types.**
  - E.g., ConcurrentQueue has TryDequeue rather than Dequeue
- **These types should be preferred when writing code requiring thread-safe access to a collection**
- **ConcurrentQueue and ConcurrentStack are lock free collections**

ConcurrentQueue Example

```
ConcurrentQueue<Job> pendingJobs = new ConcurrentQueue<Job>();
for (int jobId = 0; jobId < 100000; jobId++)
{
    pendingJobs.Enqueue(new Job(jobId));
}

Thread[] processingThreads = new Thread[2];

// Delegate to process the threads
ThreadStart runJobs = delegate()
{
    Job job = null;
    while (pendingJobs.TryDequeue(out job))
    {
        job.Process();
    }
};

// Create and start the threads
for (int i = 0; i < processingThreads.Length; i++)
{
    processingThreads[i] = new Thread(runJobs);
    processingThreads[i].Start();
}

// Wait until all the threads are done
foreach (Thread t in processingThreads)
{
    t.Join();
}
```

Summary

- **Concurrency**
  - Single Vs. Multi-Threaded Processing
- **Solutions**
  - Caller Synchronization
  - Monitor Locking
  - ReaderWriterLockSlim Locking
- **Concurrent Collections in .NET**

References

- Monitor
  - http://msdn.microsoft.com/en-us/library/System.Threading.Monitor.aspx
- ReaderWriterLockSlim
  - http://msdn.microsoft.com/en-us/library/System.Threading.ReaderWriterLockSlim.aspx
- Concurrent Collections
  - http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx
- Algorithms and Data Structures (Queue implementation)
  - http://www.pluralsight-training.net/microsoft/Courses/TableOfContents?courseName=ads-part1
- "Concurrent Programming on Windows" (Joe Duffy)
  - http://www.bluebytesoftware.com/books/winconc/winconc_book_resources.html