Thread synchronization

Thread synchronization is a technique used in multithreaded programming to ensure that only one thread is executing a certain piece of code at any given point in time. This is particularly important when multiple threads are accessing or modifying shared resources, as it can prevent data inconsistency and race conditions. Here are some common ways to manage thread synchronization in C#:

@lijotech

Lock Keyword

The lock keyword in C# is a synchronization mechanism that allows only one thread to access a specific piece of code or a common field at a time. This is used to avoid inconsistent results when reading and writing public variables.

Use Case: The lock keyword is useful when you need to ensure that only one thread can access a shared resource at a time.

Drawbacks: The lock keyword can lead to deadlocks if not used carefully. For example, if thread A locks object X and then tries to lock object Y, but at the same time thread B has already locked object Y and is waiting for object X to be released, a deadlock occurs

Lock Keyword Example

```
class Account
{
    private Object thisLock = new Object();
    int balance = 100;

    void Withdraw(int amount)
    {
        lock (thisLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
            }
        }
    }
}
```

In the example, the lock keyword is used to ensure that only one thread can access the balance field at a time

@lijotech

Monitor Class

The Monitor class in C# provides a mechanism for synchronizing access to objects. The **Monitor.Enter** method acquires an exclusive lock on the object, and the **Monitor.Exit** method releases the lock.

Use Case: The Monitor class is useful when you need more control over the lock. For example, the Monitor class provides methods to try to acquire a lock without blocking, to release a lock, and to notify threads waiting for a lock that it has been released.

Drawbacks: Like the lock keyword, the Monitor class can also lead to deadlocks if not used carefully. Additionally, forgetting to call **Monitor.Exit** in a finally can lead to serious problems, as the lock would never be released.

Monitor Class Example

```
class Account
{
    private Object thisLock = new Object();
    int balance = 100;
    void Withdraw(int amount)
    {
        Monitor.Enter(thisLock);
        try
        {
            if (balance >= amount)
            {
                balance -= amount;
            }
        }
        finally
        {
            Monitor.Exit(thisLock);
        }
    }
}
```

In this example, the Monitor class is used to ensure that the Withdraw method is thread-safe.

@lijotech

Mutexes

A Mutex is similar to a lock, but it can work across multiple processes, which means it can be used for interprocess synchronization.

Use Case: Mutexes are useful when you need to synchronize threads across multiple processes.

Drawbacks: Like with locks and monitors, the main drawback of using a Mutex is the potential for deadlocks. Also, Mutexes are slower than locks and monitors.

@lijotech

Mutexes Example

```
class Program
{
    private static Mutex mutex = new Mutex();
    static int resource = 0;

    static void Worker()
    {
        mutex.WaitOne();
        resource += 1;
        mutex.ReleaseMutex();
    }
}
```

In this example, a Mutex is used to ensure that only one thread at a time can enter a critical section of code.

ReaderWriterLockSlim

This is used when you have a shared resource where you want to allow multiple threads for reading or exclusive access for writing.

Use Case: ReaderWriterLockSlim is useful for scenarios where your applications have multiple readers and fewer writers.

Drawbacks: The main drawback of **ReaderWriterLockSlim** is that it incurs some overhead. Therefore, you should use it when the benefits of concurrent access are higher than the cost of acquiring and releasing the lock.

@lijotech

ReaderWriterLockSlim Example

```
public class Example
{
    static ReaderWriterLockSlim _rw = new ReaderWriterLockSlim();
    static int resource = 0;

    static void Read()
    {
        _rw.EnterReadLock();
        int temp = resource; // Read from resource
        _rw.ExitReadLock();
    }

    static void Write()
    {
        _rw.EnterWriteLock();
        resource = 123; // Write to resource
        _rw.ExitWriteLock();
    }
}
```

In this example, a ReaderWriterLockSlim is used to allow multiple threads to read from a shared resource simultaneously, but only one thread to write to it.

@lijotech

Semaphores

A Semaphore is a thread synchronization construct that can be used to protect an entire block of code or just part of it. It can also be used to limit concurrent access to a certain number of threads.

Use Case: Semaphores are useful when you need to limit the number of threads that can access a certain resource or group of resources.

Drawbacks: The main drawback of semaphores is complexity. It can be difficult to correctly implement a semaphore with correct initialization, increment, and decrement operations.

@lijotech

Semaphores Example

```
class Program
{
    private static Semaphore _pool = new Semaphore(0, 3);

    static void Worker()
    {
        _pool.WaitOne();
        // Do some work.
        _pool.Release();
    }
}
```

In this example, a Semaphore is used to limit the number of threads that can enter a critical section of code.

@lijotech

Semaphore and SemaphoreSlim

They are both used for limiting the number of threads that can access a resource or pool of resources concurrently in C#.

Semaphore	SemaphoreSlim
The Semaphore class is a thin wrapper around the Win32 semaphore object, which means it can be used for inter-process synchronization. It can also create local semaphores. Semaphore always passes the job to the OS, which makes it relatively expensive if the semaphore is not strongly contested.	The SemaphoreSlim class is a lightweight, fast semaphore that can be used for waiting within a single process when wait times are expected to be very short. SemaphoreSlim relies as much as possible on synchronization primitives provided by the common language runtime (CLR). SemaphoreSlim does not support named semaphores or the use of a wait handle for synchronization. It is based on **SpinWait** and **Monitor**, so the thread that waits to acquire the lock is burning CPU cycles for some time in hope to acquire the lock prior to yielding to another thread.

In summary, SemaphoreSlim is a lighter and faster alternative to Semaphore and should be used when wait times are expected to be very short and only within a single process. Semaphore, on the other hand, should be used when you need to synchronize threads across multiple processes.

@lijotech

Stay Focused & Happy Coding

Follow Lijo Sebastian for more coding tips

@lijotech

@lijotech