Mutex in C#

Mutex also works likes a lock i.e. acquired an exclusive lock on a shared resource from concurrent access, but it works across multiple processes. As we already discussed exclusive locking is basically used to ensure that at any given point of time, only one thread can enter into the critical section.

The **`Mutex`** class provides the **`WaitOne()`** method which we need to call to lock the resource and similarly it provides **ReleaseMutex()** which is used to unlock the resource. Note that a **Mutex** can only be released from the same thread which obtained it.

Example:

In the following example, each thread calls the **`WaitOne(Int32)`** method to acquire the mutex. If the time-out interval elapses, the method returns false, and the thread neither acquires the mutex nor gains access to the resource the mutex protects. The ReleaseMutex method is called only by the thread that acquires the mutex.

```
using System;
using System.Threading;
class Example
{
    // Create a new Mutex. The creating thread does not own the mutex.
    private static Mutex mut = new Mutex();
    private const int numIterations = 1;
    private const int numThreads = 3;
    static void Main()
    {
        Example ex = new Example();
        ex.StartThreads();
    }
    private void StartThreads()
    {
        // Create the threads that will use the protected resource.
        for(int i = 0; i < numThreads; i++)
        {
            Thread newThread = new Thread(new ThreadStart(ThreadProc));
            newThread.Name = String.Format("Thread{0}", i + 1);
            newThread.Start();
        }
        // main thread returns to Main and exits, but the application continues to
        // run until all foreground threads have exited.
    }
    private static void ThreadProc()
    {
        for(int i = 0; i < numIterations; i++)
        {
```

```
            UseResource();
        }
    }
    // This method represents a resource that must be synchronized
    // so that only one thread at a time can enter.
    private static void UseResource()
    {
        // Wait until it is safe to enter, and do not enter if the request times out.
        Console.WriteLine("{0} is requesting the mutex", Thread.CurrentThread.Name);
        if (mut.WaitOne(1000)) {
           Console.WriteLine("{0} has entered the protected area",
               Thread.CurrentThread.Name);
           // Place code to access non-reentrant resources here.
           // Simulate some work.
           Thread.Sleep(5000);
           Console.WriteLine("{0} is leaving the protected area",
               Thread.CurrentThread.Name);
           // Release the Mutex.
           mut.ReleaseMutex();
           Console.WriteLine("{0} has released the mutex",
                             Thread.CurrentThread.Name);
        }
        else {
           Console.WriteLine("{0} will not acquire the mutex",
                             Thread.CurrentThread.Name);
        }
    }
    ~Example()
    {
       mut.Dispose();
    }
}
// The example displays output like the following:
//       Thread1 is requesting the mutex
//       Thread1 has entered the protected area
//       Thread2 is requesting the mutex
//       Thread3 is requesting the mutex
//       Thread2 will not acquire the mutex
//       Thread3 will not acquire the mutex
//       Thread1 is leaving the protected area
//       Thread1 has released the mutex
```

This type implements the `IDisposable` interface. When you have finished using the type, you should dispose of it either directly or indirectly. To dispose of the type directly, call its Dispose method in a try/catch block. To dispose of it indirectly, use a language construct such as `using` (in C#).

Semaphore in C#

The Semaphore in C# is used to limit the number of threads that can have access to a shared resource concurrently. In other words, we can say that Semaphore **allows one or more threads** to enter into the critical section and execute the task concurrently with thread safety. So, in real-time, we need to use Semaphore when we have a limited number of resources and we want to limit the number of threads that can use it.

The Semaphores are `Int32` variables that are stored in operating system resources. When we initialize the semaphore object we initialize it with a number. This number basically used to limits the threads that can enter into the critical section. So, when a thread enters into the critical section, it decreases the value of the `Int32` variable with 1 and when a thread exits from the critical section, it then increases the value of the `Int32` variable with 1. The most important point that you need to remember is when the value of the `Int32` variable is 0, then no thread can enter into the critical section. You can use the following constructor statement to create the Semaphore instance in C#.

```
public Semaphore (int initialCount, int maximumCount);
```

As you can see in the above statement, we are passing two values to the constructor of the Semaphore class while initializing. These two values represent `InitialCount` and `MaximumCount`.

The `InitialCount` parameter sets the value for the `Int32` variable. that is it defines the initial number of requests for the semaphore that can be granted concurrently. `MaximumCount` parameter defines the maximum number of requests for the semaphore that can be granted concurrently.

For example, if we set the maximum count value as 3 and initial count value 0, it means 3 threads are already in the critical section. If we set the maximum count value as 3 and the initial count value 2. It means a maximum of 3 threads can enter into the critical section and there is one thread that is currently in the critical section.

Note: The second parameter always must be equal or greater than the first parameter otherwise we will get an exception.

Methods for Semaphore

`WaitOne()` Method: Threads can enter into the critical section by using the WaitOne method. We need to call the WaitOne method on the semaphore object. If the `Int32` variable which is maintained by semaphore is greater than 0 then it allows the thread to enter into the critical section.

`Release()` Method: We need to call the Release method when the thread wants to exits from the critical section. When this method is called, it increments the `Int32` variable which is maintained by the semaphore object.

Example:

```
using System;
using System.Threading;
namespace SemaphoreDemo
{
```

```
    public static Semaphore semaphore;
    class Program
    {
        semaphore = new Semaphore(2, 3);
        static void Main(string[] args)
        {
            for (int i = 1; i <= 5; i++)
            {
                Thread threadObject = new Thread(DoSomeTask)
                {
                    Name = "Thread " + i
                };
                threadObject.Start(i);
            }
            Console.ReadKey();
        }
        static void DoSomeTask(object id)
        {
            Console.WriteLine(Thread.CurrentThread.Name  +  "  Wants  to  Enter  into
Critical Section for processing");
            try
            {
            //Blocks the current thread until the current WaitHandle receives a signal.
                semaphore.WaitOne();
                Console.WriteLine("Success: " + Thread.CurrentThread.Name + " is Doing
its work");
                Thread.Sleep(5000);
                Console.WriteLine(Thread.CurrentThread.Name + "Exit.");
            }
            finally
            {
            //Release() method to release semaphore
                semaphore.Release();
            }
        }
    }
}
// The example displays output like the following:
//       Thread 2 Wants to Enter into Critical Section for processing
//       Thread 1 Wants to Enter into Critical Section for processing
//       Thread 3 Wants to Enter into Critical Section for processing
//       Thread 4 Wants to Enter into Critical Section for processing
//       Thread 5 Wants to Enter into Critical Section for processing
//       Success: Thread 2 is doing its work
//       Success: Thread 1 is doing its work
//       Thread 2 Exit.
//       Thread 1 Exit.
//       Success: Thread 3 is doing its work
//       Success: Thread 4 is doing its work
//       Thread 3 Exit
//       Success: Thread 5 is doing its work
//       Thread 4 Exit.
//       Thread 5 Exit.
```