Thread Signalling (Thread Interaction)

Thread interaction (or thread signalling) means that a thread must wait for notification, or a signal, from one or more threads in order to proceed. For example, if thread A calls the Thread.Join method of thread B, thread A is blocked until thread B completes. The synchronization primitives described in the preceding section provide a different mechanism for signalling: by releasing a lock, a thread notifies another thread that it can proceed by acquiring the lock. You use **AutoResetEvent**, **ManualResetEvent**, and **EventWaitHandle** for thread interaction

AutoResetEvent

It represents a thread synchronization event that, when signalled, resets automatically after releasing a single waiting thread. This class cannot be inherited.

```
public sealed class AutoResetEvent : System.Threading.EventWaitHandle
```

A thread waits for a signal by calling AutoResetEvent.WaitOne. If the AutoResetEvent is in the non-signaled state, the thread blocks until AutoResetEvent.Set is called.

Calling Set signals AutoResetEvent to release a waiting thread. AutoResetEvent remains signaled until a single waiting thread is released, and then automatically returns to the non-signaled state. If no threads are waiting, the state remains signaled indefinitely.

If a thread calls WaitOne while the AutoResetEvent is in the signaled state, the thread does not block. The AutoResetEvent releases the thread immediately and returns to the non-signaled state.

You can control the initial state of an AutoResetEvent by passing a Boolean value to the constructor: true if the initial state is signaled and false otherwise.

Example:

The following example shows how to use AutoResetEvent to release one thread at a time, by calling the Set method (on the base class) each time the user presses the Enter key. The example starts three threads, which wait on an AutoResetEvent that was created in the signaled state. The first thread is released immediately, because the AutoResetEvent is already in the signaled state. This resets the AutoResetEvent to the non-signaled state, so that subsequent threads block. The blocked threads are not released until the user releases them one at a time by pressing the Enter key.

After the threads are released from the first AutoResetEvent, they wait on another AutoResetEvent that was created in the non-signaled state. All three threads block, so the Set method must be called three times to release them all.

```
using System;
using System.Threading;
class Example
{
    private static AutoResetEvent event_1 = new AutoResetEvent(true);
    private static AutoResetEvent event_2 = new AutoResetEvent(false);
    static void Main()
```

```
    {
        Console.WriteLine("Press Enter to create three threads and start
them.\r\n" + "The threads wait on AutoResetEvent #1, which was created\r\n" + "in
the signaled state, so the first thread is released.\r\n" + "This puts
AutoResetEvent #1 into the unsignaled state.");
        Console.ReadLine();
        for (int i = 1; i < 4; i++)
        {
            Thread t = new Thread(ThreadProc);
            t.Name = "Thread_" + i;
            t.Start();
        }
        Thread.Sleep(250);
        for (int i = 0; i < 2; i++)
        {
            Console.WriteLine("Press Enter to release another thread.");
            Console.ReadLine();
            event_1.Set();
            Thread.Sleep(250);
        }
        Console.WriteLine("\r\nAll threads are now waiting on AutoResetEvent
#2.");
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine("Press Enter to release a thread.");
            Console.ReadLine();
            event_2.Set();
            Thread.Sleep(250);
        }
    }
    static void ThreadProc()
    {
        string name = Thread.CurrentThread.Name;
        Console.WriteLine("{0} waits on AutoResetEvent #1.", name);
        event_1.WaitOne();
        Console.WriteLine("{0} is released from AutoResetEvent #1.", name);
        Console.WriteLine("{0} waits on AutoResetEvent #2.", name);
        event_2.WaitOne();
        Console.WriteLine("{0} is released from AutoResetEvent #2.", name);
        Console.WriteLine("{0} ends.", name);
    }
}
/* This example produces output similar to the following:

Press Enter to create three threads and start them.
The threads wait on AutoResetEvent #1, which was created
in the signaled state, so the first thread is released.
This puts AutoResetEvent #1 into the unsignaled state.

Thread_1 waits on AutoResetEvent #1.
Thread_1 is released from AutoResetEvent #1.
Thread_1 waits on AutoResetEvent #2.
```

```
Thread_3 waits on AutoResetEvent #1.
Thread_2 waits on AutoResetEvent #1.
Press Enter to release another thread.

Thread_3 is released from AutoResetEvent #1.
Thread_3 waits on AutoResetEvent #2.
Press Enter to release another thread.

Thread_2 is released from AutoResetEvent #1.
Thread_2 waits on AutoResetEvent #2.

All threads are now waiting on AutoResetEvent #2.
Press Enter to release a thread.

Thread_2 is released from AutoResetEvent #2.
Thread_2 ends.
Press Enter to release a thread.

Thread_1 is released from AutoResetEvent #2.
Thread_1 ends.
Press Enter to release a thread.

Thread_3 is released from AutoResetEvent #2.
Thread_3 ends.
 */
```

ManualResetEvent

It represents a thread synchronization event that, when signalled, must be reset manually. This class cannot be inherited.

```
public sealed class ManualResetEvent : System.Threading.EventWaitHandle
```

When a thread begins an activity that must complete before other threads proceed, it calls ManualResetEvent.Reset to put ManualResetEvent in the non-signaled state. This thread can be thought of as controlling the ManualResetEvent. Threads that call ManualResetEvent.WaitOne block, awaiting the signal. When the controlling thread completes the activity, it calls ManualResetEvent.Set to signal that the waiting threads can proceed. All waiting threads are released.

Once it has been signaled, ManualResetEvent remains signalled until it is manually reset by calling the Reset() method. That is, calls to WaitOne return immediately.

You can control the initial state of a ManualResetEvent by passing a Boolean value to the constructor: true if the initial state is signalled, and false otherwise.

Example:

The following example demonstrates how ManualResetEvent works. The example starts with a ManualResetEvent in the unsignaled state (that is, false is passed to the constructor). The example creates three threads, each of which blocks on the ManualResetEvent by calling its WaitOne method. When the user presses the Enter key, the example calls the Set method, which releases all three threads. Contrast

this with the behavior of the AutoResetEvent class, which releases threads one at a time, resetting automatically after each release.

Pressing the Enter key again demonstrates that the ManualResetEvent remains in the signaled state until its Reset method is called: The example starts two more threads. These threads do not block when they call the WaitOne method, but instead run to completion.

Pressing the Enter key again causes the example to call the Reset method and to start one more thread, which blocks when it calls WaitOne. Pressing the Enter key one final time calls Set to release the last thread, and the program ends.

```
using System;
using System.Threading;
public class Example
{
    // mre is used to block and release threads manually. It is
    // created in the unsignaled state.
    private static ManualResetEvent mre = new ManualResetEvent(false);
    static void Main()
    {
        Console.WriteLine("\nStart 3 named threads that block on a
ManualResetEvent:\n");
        for(int i = 0; i <= 2; i++)
        {
            Thread t = new Thread(ThreadProc);
            t.Name = "Thread_" + i;
            t.Start();
        }
        Thread.Sleep(500);
        Console.WriteLine("\nWhen all three threads have started, press Enter to
call Set()" + "\nto release all the threads.\n");
        Console.ReadLine();
        mre.Set();
        Thread.Sleep(500);
        Console.WriteLine("\nWhen a ManualResetEvent is signaled, threads that
call WaitOne()" + "\ndo not block. Press Enter to show this.\n");
        Console.ReadLine();
        for(int i = 3; i <= 4; i++)
        {
            Thread t = new Thread(ThreadProc);
            t.Name = "Thread_" + i;
            t.Start();
        }
        Thread.Sleep(500);
        Console.WriteLine("\nPress Enter to call Reset(), so that threads once
again block" + "\nwhen they call WaitOne().\n");
        Console.ReadLine();
        mre.Reset();
        // Start a thread that waits on the ManualResetEvent.
        Thread t5 = new Thread(ThreadProc);
        t5.Name = "Thread_5";
        t5.Start();
```

```
        Thread.Sleep(500);
        Console.WriteLine("\nPress Enter to call Set() and conclude the demo.");
        Console.ReadLine();
        mre.Set();
    }
    private static void ThreadProc()
    {
        string name = Thread.CurrentThread.Name;
        Console.WriteLine(name + " starts and calls mre.WaitOne()");
        mre.WaitOne();
        Console.WriteLine(name + " ends.");
    }
}
/* This example produces output similar to the following:

Start 3 named threads that block on a ManualResetEvent:

Thread_0 starts and calls mre.WaitOne()
Thread_1 starts and calls mre.WaitOne()
Thread_2 starts and calls mre.WaitOne()

When all three threads have started, press Enter to call Set()
to release all the threads.


Thread_2 ends.
Thread_0 ends.
Thread_1 ends.

When a ManualResetEvent is signaled, threads that call WaitOne()
do not block. Press Enter to show this.


Thread_3 starts and calls mre.WaitOne()
Thread_3 ends.
Thread_4 starts and calls mre.WaitOne()
Thread_4 ends.

Press Enter to call Reset(), so that threads once again block
when they call WaitOne().


Thread_5 starts and calls mre.WaitOne()

Press Enter to call Set() and conclude the demo.

Thread_5 ends.
 */
```