Threads

- Threads are lightweight processes responsible for multitasking within a single application.

- The class Thread represents an object-oriented wrapper around a given path of execution.

- The class Thread defines a number of methods that allow one to create new threads from a current thread, as well as stop, suspend, and destroy a given thread.
## System.Threading

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interlocked</td>
<td>Provides synchronized access to shared data.</td>
</tr>
<tr>
<td>Monitor</td>
<td>Provides the synchronization of threading objects using locks and wait/signals.</td>
</tr>
<tr>
<td>Mutex</td>
<td>Synchronization primitive that can be used for inter process synchronization.</td>
</tr>
<tr>
<td>Thread</td>
<td>Represents a thread that executes within the CLR.</td>
</tr>
<tr>
<td>ThreadPool</td>
<td>Manages related threads in a given process.</td>
</tr>
<tr>
<td>Timer</td>
<td>Specifies a delegate that can be called at a specified time.</td>
</tr>
<tr>
<td>WaitHandle</td>
<td>Represents all synchronization objects at runtime.</td>
</tr>
</tbody>
</table>

Com S 430
## System.Threading Delegates

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ThreadStart</td>
<td>The ThreadStart class is a delegate that points to the method that should be executed first when the thread is started.</td>
</tr>
<tr>
<td>TimerCallback</td>
<td>Delegate for Timers.</td>
</tr>
<tr>
<td>WaitCallback</td>
<td>This class is a delegate that defines the callback method for ThreadPool user work items.</td>
</tr>
</tbody>
</table>
Static Thread Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentThread</td>
<td>This (read-only) property returns a reference to the currently running thread.</td>
</tr>
<tr>
<td>GetData(),</td>
<td>Retrieves the value from the specified slot on the current thread, for that thread’s current domain.</td>
</tr>
<tr>
<td>SetData()</td>
<td></td>
</tr>
<tr>
<td>GetDomain(),</td>
<td>Returns a reference to the current AppDomain (or the ID of this domain) in which the current thread is running.</td>
</tr>
<tr>
<td>GetDomainID()</td>
<td></td>
</tr>
<tr>
<td>Sleep()</td>
<td>Suspends the current thread for a specified time.</td>
</tr>
</tbody>
</table>
# Thread Members

<table>
<thead>
<tr>
<th>Member</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsAlive</td>
<td>This property returns a boolean that indicates if this thread has been started.</td>
</tr>
<tr>
<td>IsBackground</td>
<td>Gets or sets a value indicating whether or not this thread is a background thread.</td>
</tr>
<tr>
<td>Priority</td>
<td>Gets or sets the priority of a thread, which may be assigned a value from the ThreadPriority enumeration.</td>
</tr>
<tr>
<td>ThreadState</td>
<td>Gets the state of this thread.</td>
</tr>
<tr>
<td>Join()</td>
<td>Instructs the thread to wait for a given thread.</td>
</tr>
<tr>
<td>Interrupt(),</td>
<td>Interrupts, resumes, starts, or suspends a thread.</td>
</tr>
<tr>
<td>Resume(), Start(),</td>
<td></td>
</tr>
<tr>
<td>Suspend()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Thread Example

- Thread 1
- Thread 2
- Shared Resource
Critical Section

Thread 1

Enter
Resume

Shared Resource

Leave

Enter
Resume

Thread 2

Leave

Thread 2
GUI

CriticalSectionRotator

Start Thread

Start Thread

Stop Thread

Start

Stop

Start

Stop
public class DoubleRotator : System.Windows.Forms.Form
{
    // Thread objects
    private Thread fThread1;
    private Thread fThread2;
    // Thread visualization
    private RotatorPanel.CriticalSectionRotator criticalSectionRotator1;
    private RotatorPanel.CriticalSectionRotator criticalSectionRotator2;
    // Mutex: Abstraction to model a critical section
    private Mutex fCriticalSection;
    ...
}
public DoubleRotator()
{
    // Required for Windows Form Designer support
    InitializeComponent();

    // Register event handlers
    criticalSectionRotator1.EnterSection += new EventHandler(EnterSec);
criticalSectionRotator1.LeaveSection += new EventHandler(LeaveSec);
criticalSectionRotator2.EnterSection += new EventHandler(EnterSec);
criticalSectionRotator2.LeaveSection += new EventHandler(LeaveSec);

    // Create Mutex object
    fCriticalSection = new Mutex();
}

Register Handlers
Event Handlers

```csharp
private void EnterSec( object sender, EventArgs e )
{
    // Blocks the current thread until the current WaitHandle
    // receives a signal. If WaitHandle is immediately available,
    // the current thread can continue.
    fCriticalSection.WaitOne();
}

private void LeaveSec( object sender, EventArgs e )
{
    // Releases the Mutex once.
    fCriticalSection.ReleaseMutex();
}
```
Start Threads

```csharp
private void button1_Click(object sender, System.EventArgs e)
{
    fThread1 = new Thread( new ThreadStart( Run1 ) );
    fThread1.Start();
    button1.Enabled = false; // disable START
    button2.Enabled = true;  // enable STOP
}

private void Run1()
{
    for (; ; )
    try
    {
        criticalSectionRotator1.Rotate(); Thread.Sleep( 100 );
    }
    catch (Exception e)
    {
        MessageBox.Show( e.Message ); return;
    }
}
```
Stop Threads

```csharp
private void button2_Click(object sender, System.EventArgs e)
{
    try
    {
        // Stop the thread immediately.
        // Raises a ThreadAbortException in the thread on which it
        // is invoked, to begin the process of terminating the thread.
        fThread1.Abort();
    }
    catch
    {}
    button1.Enabled = true;   // enable START
    button2.Enabled = false;  // disable STOP
}
```
Asynchronous Counter

Server started.
Register player #0
Register player #1
Register player #2
Register player #3
Register player #4
End of asynchronous event "StartGame".
Number of listeners: 5
All listeners have been informed.
End of asynchronous event "StartGame".
Number of listeners: 5
All listeners have been informed.
End of asynchronous event "StartGame".
Number of listeners: 5
All listeners have been informed.
StartGame

Com S 430
public void RegisterPlayer(IPlayer aPlayer)
{
    // synchronize concurrent access
    lock(this)
    {
        aPlayer.Number = fPlayers.Count;
        fPlayers.Add(aPlayer);
        Console.WriteLine(…);
    }
}
private void fTimer_Tick(object sender, System.EventArgs e)
{
    fCount += 1;
    if ( fCount <= fMax )
    {
        Graphics g = Graphics.FromHwnd( panel1.Handle );
        g.DrawImage( BackgroundImage, panel1.ClientRectangle );
        g.DrawString( … );
        fTimer.Start();
    }
    else
    {
        button1.Enabled = true;
        fTimer.Stop();
        fCount = -1;
    }
}
Nested Locks

Server

StartGame

StartNewGame

Event Thread

Event Handler

Change Form

Nested Lock

Click

Main Thread

Com S 430
Asynchronous Calls

- In a synchronous call to a method, the caller thread blocks while the call is active. When the call completes, the method can return results.

- When a thread makes an asynchronous call to a method, the call returns immediately. The caller thread is not blocked. On completion the called method can return a result using a callback.
**Used-defined Delegates**

- For every user-defined delegate the C# compiler generates a sealed delegate class.

```csharp
public delegate int StringDelegate(string aString);

public sealed class StringDelegate : System.MulticastDelegate
{
    public StringDelegate(object obj, int method);
    public virtual int Invoke(string aString);
    public virtual IAsyncResult BeginInvoke(string aString, AsyncCallback asc, object stateObject);
    public virtual int EndInvoke(IAsyncResult result);
}
```
Asynchronous Calls on Delegates

**BeginInvoke**
- The BeginInvoke parameters will start with the input parameters of the delegate.
- BeginInvoke starts the asynchronous call to the method, and then the infrastructure queues the method to run on a thread pool thread and creates synchronization objects needed to determine if the method has completed.

**EndInitinvoke**
- TheEndInitinvoke parameters will start with the output parameters (marked as out or ref).
-EndInitinvoke is used to harvest the results and allow the system to perform cleanup.
int lIndex = 0;
IAsyncResult[,] lResults =
   new IAsyncResult[StartGame.GetInvocationList().Length];

foreach( Delegate lDelegate in StartGame.GetInvocationList() )
{
   try {
      lHandler = (StartGameHandler)lDelegate;
      lResults[lIndex++] = lHandler.BeginInvoke( fMessage, null, null );
   } catch { StartGame -= lHandler; } 
}

   // wait for completion
   for ( int i = 0; i < lIndex; i++ ) lResults[i].AsyncWaitHandle.WaitOne();
The class ThreadPool provides a pool of threads that can be used to post work items, process asynchronous I/O, wait on behalf of other threads, and process timers.

You can also queue work items that are not related to a wait operation to the thread pool. To request that a work item be handled by a thread in the thread pool, call the QueueUserWorkItem method. This method takes as a parameter a reference to the method or delegate that will be called by the thread selected from the thread pool. There is no way to cancel a work item after it has been queued.
Inform All Subscribers

class Worker
{
    private string fMessage;

    public Worker( string aMessage ) { … }

    public void InitiateGame( object sender ) { … }
}

public void StartNewGame( string aMessage )
{
    Worker lWorkItem = new Worker( aMessage );
    WaitCallBack lCallBack = new WaitCallBack( lWorkItem. InitiateGame );
    ThreadPool.QueueUserWorkItem( lCallBack ); // asynchronous call
}
Important Form Methods

- **BeginInvoke:**
  - Executes a delegate asynchronously on the thread that the control's underlying handle was created on.

- **EndInvoke:**
  - Retrieves the return value of the asynchronous operation represented by the IAsyncResult object passed.

- **Invoke:**
  - Executes a delegate on the thread that owns the control's underlying window handle.
private void HandleMessage( string aMessage )
{
    MessageBox.Show( aMessage );
    fMax = (new Random()).Next( 2, 10 );
    fTimer_Tick( null, null );
}

Timer is activated by the event handler’s thread. Timer events will never receive Counter.
private void HandleMessage( string aMessage )
{
    MessageBox.Show( aMessage );
    fMax = (new Random()).Next( 2, 10 );
    // invoke this.StartTimer on the Thread that owns this form
    Invoke( new Notify( this.StartTimer ) );
}

Invoke will try to look the form object, but the thread that owns form has already look the form object - application freezes.
Correct Handler

```csharp
private void HandleMessage( string aMessage )
{
    MessageBox.Show( aMessage );
    fMax = (new Random()).Next( 2, 10 );
    // invoke this.StartTimer on the Thread that owns this form
    BeginInvoke( new Notify( this.StartTimer ) );
}
```

BeginInvoke is started, when the thread that owns the form object becomes available. In this case, the look need to perform previous action has been released.
InvokeRequired

The property InvokeRequired gets a value indicating whether the caller must call an invoke method when making method calls to the control because the caller is on a different thread than the one the control was created on.

delegate void Notify();

private update_control()
{
    if ( this.InvokeRequired )
        this.Invoke( new Notify( this.DoUpdate ) );
    else
        this.DoUpdate();
}