**Threading**

**Background/Foreground** is a flag (property) on a thread object.

When fore- threads in process stop, CLR forcibly ends any back- threads.

Back- threads stop, no exception.

Unhandled exceptions occurring on any thread cause the application to be terminated by CLR.

It is possible to change a thread from foreground to background and vice versa any time during its lifetime

**Overhead of creating a thread**.

Thread gets 1 MB of reserved for its user-mode stack, and another 12 KB (or so) for its kernel-mode stack. Windows notifies every DLL in process after creating or before destroying thread.

Windows has to keep track of thread objects, every so often additional code has to execute once in 20 ms. Thread is associated with *execution context*. It includes Security, Localization and Transaction settings

Use fore- threads to execute mission-critical tasks, use back- threads for not mission-critical tasks

- If you want it to run at a special priority

- If you want to make foreground thread, preventing the application from dying until thread is completed.

- If the task is extremely long running

- if you want to start a thread and possibly abort it

When **Yield()** occurs, another thread ready to execute will be picked by OS scheduler to run

**Thread Pool**

Concept of a resource pool involves pre-allocating a set of resources that will be reused

Thread pool maintains a queue of operation requests. Call method that appends into the thread pool's queue.

New thread will go through initialization. When the task is complete, the thread will not destroy itself, it will return to the thread pool in a suspended state. On another request the suspended thread will wake up.

CLR's thread pool tries to avoid creating additional threads - more than one thread per 500 milliseconds.

*SetMinThreads* tells thread pool quickly create this many threads. After that it creates threads with delay

There is one thread pool per CLR; this thread pool is shared by all AppDomains controlled by that CLR.

In 40 seconds, if suspended thread is given nothing to do, the thread wakes up and destroys itself

Typical use of thread pool - to do relatively short-lived work items that need to execute in the background

There are a number of ways to enter the thread pool:

• Via the Task Parallel Library or PLINQ (from Framework 4.0)

• By calling static ThreadPool. QueueUserWorkItem (WaitCallback callBack, Object state)

• Via asynchronous delegates

• Via BackgroundWorker

**Synchronization**

*Synchronization* is the act of coordinating concurrent actions for a predictable outcome. Synchronization is particularly important when multiple threads access the same data

Need to lock around accessing *any writable shared field*.

*Blocking methods :* **Sleep, Join, Task.Wait**

**Wait-based synchronization:**

*Exclusive locking :* **lock (Monitor), Mutex, SpinLock**

Exclusive locking constructs allow just one thread to perform some activity or execute a section of code at a time. Their primary purpose is to let threads access shared writing state without interfering with one other.

*Nonexclusive locking* **Semaphore(Slim), ReaderWriterLock(Slim)**

Nonexclusive locking lets you *limit* concurrency.

*Signaling* **ManualResetE** **vent, AutoResetEvent, CountdownEvent, Monitor’s Wait/Pulse, Barrier**

These allow a thread to block until receiving one or more notifications from other thread(s).

**Sync Blocks And The lock Keyword**

The second header field embedded in each reference type sync block index. Used for a variety of purposes, including synchronization, GC book-keeping, finalization, and hash code storage. Several bits of this field determine exactly which information is stored in it at any instant in time.

When required, the CLR allocates a *sync block* from a global array called the *sync block table*.

The sync block contains a week reference to its owning object

**Lock and Monitor**

*Monitor* class can only be used inside of the *AppDomain*. Implemented internally using a Win32 event

|  |
| --- |
| static void Enter(Object obj);  static Boolean TryEnter(Object obj);  static void Exit(Object obj);  try  {  Monitor.Enter(\_syncObject, ref acquired);  }  finally { if (acquired) Monitor.Exit(\_syncObject); |

If object's sync block index is negative, finds a free sync block and records the index of it in the sync block index.

|  |
| --- |
| Problem with **lock(this)** – locking public objects. If CLR's finalizer thread can't acquire the object's lock because the application's primary thread owns the object's lock finalizer thread to stop—no more objects in the process (which includes all AppDomains in the process) can be finalized and reclaimed from the heap! |

A specific **lock(x)** applies to ALL threads in a process.

Synchronizing object can double as the object it’s protecting

Locking doesn’t restrict access to the synchronizing object itself, x.ToString() will not block because another thread has called lock(x);

Value type instances do not have sync block index member, and cannot be used for synchronization.

C# compiler would not allow using value types with the lock keyword.

When pass value type instance to **Monitor.Enter,** it is boxed. In **Monitor.Exit,** it gets boxed again.

The result is that your code ends up locking one object and unlocking a completely different object

Recovering from potential deadlock: Specify time for TryEnter to wait for the lock. If a lock is acquired before the time expires, returns *true*; otherwise *false*.

The Monitor.Wait method blocks the calling thread and **releases** any locks it has acquired.

The Monitor.Pulse method notifies a waiting thread. Thread **reacquires** the lock and resumes execution.

Common pattern in .NET Framework: *static members are thread-safe; instance members are not.*

**static** members on the DateTime struct have been carefully programmed to be thread-safe.

**Volatile**  keyword indicates that a field might be modified by multiple threads. Volatile fields are not subject to compiler optimizations that assume access by a single thread. This ensures the most up-to-date value in the field at all times.

You can use the *Semaphore* class to define how many threads can have access before other calling threads are blocked. After a thread releases the semaphore, another can take its place.

ReaderWriterLock class lets you obtain multiple-reader/single-writer semantics

**Atomic operation** is uninterruptable, going from start to end always in one pass.

fastest way to manipulate data in a thread-safe way is to use the interlocked family of methods

|  |
| --- |
| public static Int32 Decrement(ref Int32 location);  public static Int32 Add(ref Int32 location1, Int32 value);  public static Int32 Exchange(ref Int32 location1, Int32 value);  public static Int32 CompareExchange(ref Int32 location1, Int32 value, Int32 comparand);  static **volatile** int s\_curr = 0;  static int GetNext() { return Interlocked.Increment(ref s\_curr); } |

***Signaling***is a way of allowing a thread to wait for a condition. Condition is defined using the *WaitHandle* class.

A thread that is waiting for event calls **WaitOne( )** on the event object. Itreturns immediately if the event is in signaled state. Otherwise, it suspends execution of the calling thread until the event is signaled.

Another thread sets event signaled by calling **Set( )**. After that **WaitOne( )** returns and first thread resumes.

The event returns to non-signaled state by calling **Reset( )**.

**ManualResetEvent**, the event remains signaled until a call to **Reset( )** is made.

**AutoResetEvent**, the event automatically changes to a non-signaled as soon as waiting thread resumes

**CountdownEvent** lets you wait on more than one thread.

Nonexclusive Locking Semaphores can be useful in limiting concurrency—preventing too many threads from executing a particular piece of code at once.

**Deadlock**

A deadlock occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked.

|  |
| --- |
| // Thread 1 lock (a) lock (b) // Thread 2 : lock (b) lock (a)  Guaranteed deadlock:  Object x = new obj(), y = new obj();  ThreadPool.QueueWorkItem(() => {lock(x) Thread.Sleep(1000); lock(y)} )  ThreadPool.QueueWorkItem(() => { Thread.Sleep(100); lock(y) lock(x) ) |

thread is in locked state waiting for another to release lock, which another never does

To avoid deadlock, code must ensure the order in which locks are acquired:

Techniques to avoid and detect deadlocks

TryEnter with timeout – can detect when long timeouts occure

**Race condition**

A race condition occurs when two or more threads access a shared resource at the same time.  ???

Race Condition is a scenario where the outcome of the program is affected because of timing of threads.

**Async Patterns**

|  |
| --- |
| **Obsolete**  The **IAsyncResult** and **BeginXxx** **EndXxx** pattern.  The APM supports three rendezvous techniques: *wait-until-done, polling,* and *method callback.*  public interface IAsyncResult  {  WaitHandle AsyncWaitHandle { get; } // For Wait-Until-Done technique  Boolean IsCompleted { get; } // For Polling technique  Object AsyncState { get; } // For Callback technique – **return** value from async operation  Boolean CompletedSynchronously { get; } // Almost never used  }  **Callback** technique is by far the best one - never causes a thread to enter a wait state.  **End*Xxx*** methods accept an **IAsyncResult** object as an argument.  **End*Xxx*** method suspends calling thread until the operation has completed, and then it will return the result  You must call **End*Xxx*** to avoid leak. BeginXxx constructs an instance of an IAsyncResult interface  You queue up an asynchronous request, and then your thread continues doing whatever.  Windows queues work item into thread pool.  Inside callback method, you first call the **End*Xxx***  to obtain the result, free to continue processing the result.  **Asynchronous invocation in delegates**  APM via **delegate**  ThreadPool.QueueUserWorkItem doesn’t provide an easy mechanism for getting return values from thread after it finished executing. Asynchronous delegate invocations solve this, allowing any number of typed arguments to be passed in both directions  static int Work (string s) { return s.Length; }  Func<string, int> funkDelegate = Work;  funkDelegate.BeginInvoke ("test", Done, method);  static void Done (IAsyncResult cookie)  {  funkDelegate targetDelegate = (Func<string, int>) cookie.AsyncState;  int result = targetDelegate.EndInvoke (cookie);  }  Delegate in C# produce a class that has **BeginInvoke** and **EndInvoke** methods.  public sealed class Func<T, TResult> : MulticastDelegate  {  public Func(Object object, IntPtr method);  public TResult Invoke (T arg);  public IAsyncResult BeginInvoke (T arg, AsyncCallback callback, Object object);  public TResult EndInvoke (IAsyncResult result);  }  **BeginInvoke** method has same parameters as delegate definition, plus **AsyncCallback** and **Object.**  **BeginInvoke** queues compute-bound operations to the CLR's thread pool by calling **QueueUserWorkItem**. |

**Rich Client thread affinity**

Rich client objects based on DependencyObject (WPF), or Control (WinForms.) have *thread affinity* - only the thread that instantiates them can subsequently access their members

**Forms.Control** class implements **ISynchronizeInvoke** methods **Invoke, BeginInvoke,** and **EndInvoke** that you can call from any thread to marshal operation to the thread that created the window.

Internally, **Control'**s **Invoke** method calls the Win32 **SendMessage** method. **Control'**s **BeginInvoke** method internally calls the Win32 **PostMessage** method to have the window's thread execute a task asynchronously

.NET 2.0 introduced the concept of a **synchronization context**.

|  |
| --- |
| public delegate void SendOrPostCallback(object state);  public class SynchronizationContext  {  public virtual void Post(SendOrPostCallback callback,object state);  public virtual void Send(SendOrPostCallback callback,object state);  public static SynchronizationContext Current {get;}  } |

**Tasks**

Compared to a thread, a Task is higher-level abstraction—it represents a **concurrent operation that may or may not be backed by a thread**.

Tasks are *compositional* (you can chain them together through the use of *continuations*). They can use the *thread pool* to lessen startup latency, and with a TaskCompletionSource.

|  |
| --- |
| Task.Run (() => Console.WriteLine ("Foo"));  Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); **return 3**; });  var awaiter = primeNumberTask.GetAwaiter();  awaiter.OnCompleted (() =>  {  int result = awaiter.GetResult();  });  primeNumberTask.ContinueWith (antecedent =>  {  int result = antecedent.Result;  }); |

Calling Wait on a task blocks until it completes.

There are two ways to attach a continuation to a task. ContinueWith(Close(), ...). use await instead of ContinueWith.

CLR wraps the exception in an AggregateException in order to play well with parallel programming scenarios

**Schedulers**

|  |
| --- |
| uiScheduler = **TaskScheduler.FromCurrentSynchronizationContext()**;  Task.Run (() => Foo()).ContinueWith (ant => lblResult.Content = ant.Result, **\_uiScheduler**); |

There are *default scheduler* (thread pool), and the *synchronization context scheduler* tell s task or a continuation to execute on this context.

**Async-Await**

The Task-based Asynchronous Pattern (TAP)

Framework 4.5 and later exposes hundreds of task-returning asynchronous methods that you can await (mainly related to I/O). These methods follow a pattern called the *Task-based Asynchronous Pattern* (TAP)

• Returns a “hot” (running) Task or Task<TResult>

• Has a “Async” suffix (except for special cases such as task combinators)

• Is overloaded to accept a cancellation token and/or IProgress<T>

• Returns quickly to the caller (has only a small initial *synchronous phase*)

• Does not tie up a thread

In each async method you write, some code will be before the first occurrence of the await keyword. Equally, some code is in the expression that gets awaited. This code always runs in the calling thread.

The await keyword, by contrast, is non-blocking, which means the current thread is free to do other things during the wait. But what else would the current thread be doing?

**SynchronizationContext**

SynchronizationContext is a class in .NET Framework, which has the ability to run code in a particular type of thread.

The current SynchronizationContext is a **property of the current thread**.

when method resumes, the await keyword’s infrastructure uses Post to resume the method on the captured SynchronizationContext.

C# catches any exceptions that happen in your async method. When an exception happens, it is placed into the Task that was returned to your caller. Task becomes *Faulted*. If a method is awaiting the Task when it faults, instead of resuming normally, the method is resumed by an exception thrown from the await.

support for multiple exceptions was built directly into Task. Instead of being able to contain an Exception directly, Task contains an AggregateException when it faults. An AggregateException contains a collection of other exceptions.

**Timers**

The .NET Framework provides four timers. Two of these are general-purpose multithreaded timers:

• System.Threading.Timer

• System.Timers.Timer

The other two are special-purpose single-threaded timers:

• System.Windows.Forms.Timer (Windows Forms timer)

• System.Windows.Threading.DispatcherTimer (WPF timer)

If you want a timer to fire just once, specify Timeout.Infinite in the constructor’s last argument.

The .NET Framework provides another timer class of the same name in the System.Timers namespace. This simply wraps the System.Threading.Timer, providing additional convenience while using the identical underlying engine

Multithreaded timers use the thread pool to allow a few threads to serve many timers. This means that the callback method or Elapsed event may fire on a different thread each time it is called. Furthermore, the Elapsed event always fires (approximately) on time—regardless of whether the previous Elapsed event finished executing. Hence, callbacks or event handlers must be thread-safe.

Singe-threaded timers: Instead of firing timer events on pooled threads, they post the events to the WPF or Windows Forms message loop.

A fresh Tick will never fire until the previous Tick has finished processing.

You can update user interface elements and controls directly from Tick event handling code,