Threading

Multiple concurrently executing threads

THIS DOCUMENT COVERS

- Threading Basics

Threading Basics

A thread is a single sequential flow of control within a single address space. Each thread has its own call stack and its own local variables. All threads in the same app domain read and write to the same heap. It is the programmer's responsibility to ensure that shared memory is accessed and written correctly.

Thread Definition

- Single sequential flow of control within a single address space
- Each thread has a separate call stack and its own local variables
- Multiple threads read/write to the same heap
- Threads share heap memory with other threads (running in same app domain)
- Programmers must ensure that shared memory is accessed correctly

We utilise threads to improve performance when carrying out I/O, to take advantage of multiple cores and to maintain a responsive user interface when performing long running operations.

Advantages of threads

- Better performance than multiple processes communicating across address spaces
- Improve performance when driving slow devices
- Maintain responsive user interface
- Take advantage of multiple cores

Overuse of multi-threaded programming can, however, reduce performance. Context switching between threads is a relatively expensive operation and large amounts of threads are known to reduce throughout. Multi-threaded programming increases code complexity and can introduce non-deterministic behaviour. Bugs are harder to find as often there are few clues at the point of failure as to the root cause of the problem. In addition to complexity,

Disadvantages of multi-threaded programming

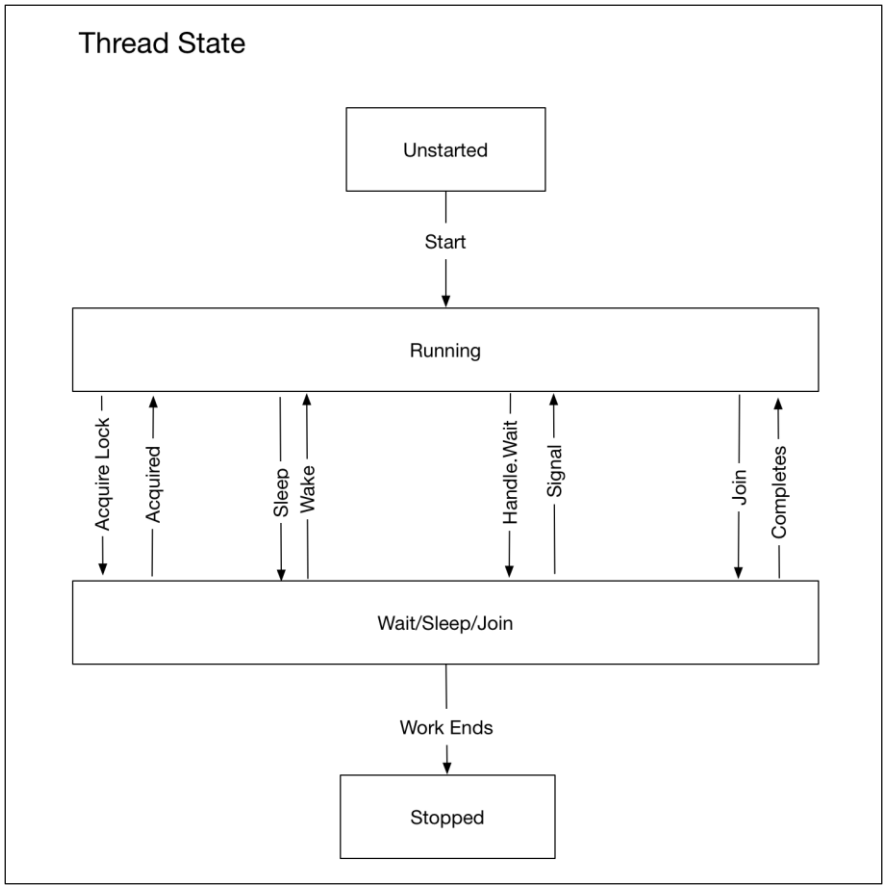
- Context switching between threads is expensive
- Too many threads can reduce throughput and hence performance.
- Non-deterministic behaviour
- Rarely enough information at the point of failure to debug the application
- Only guarantee is that bugs will never show until in production

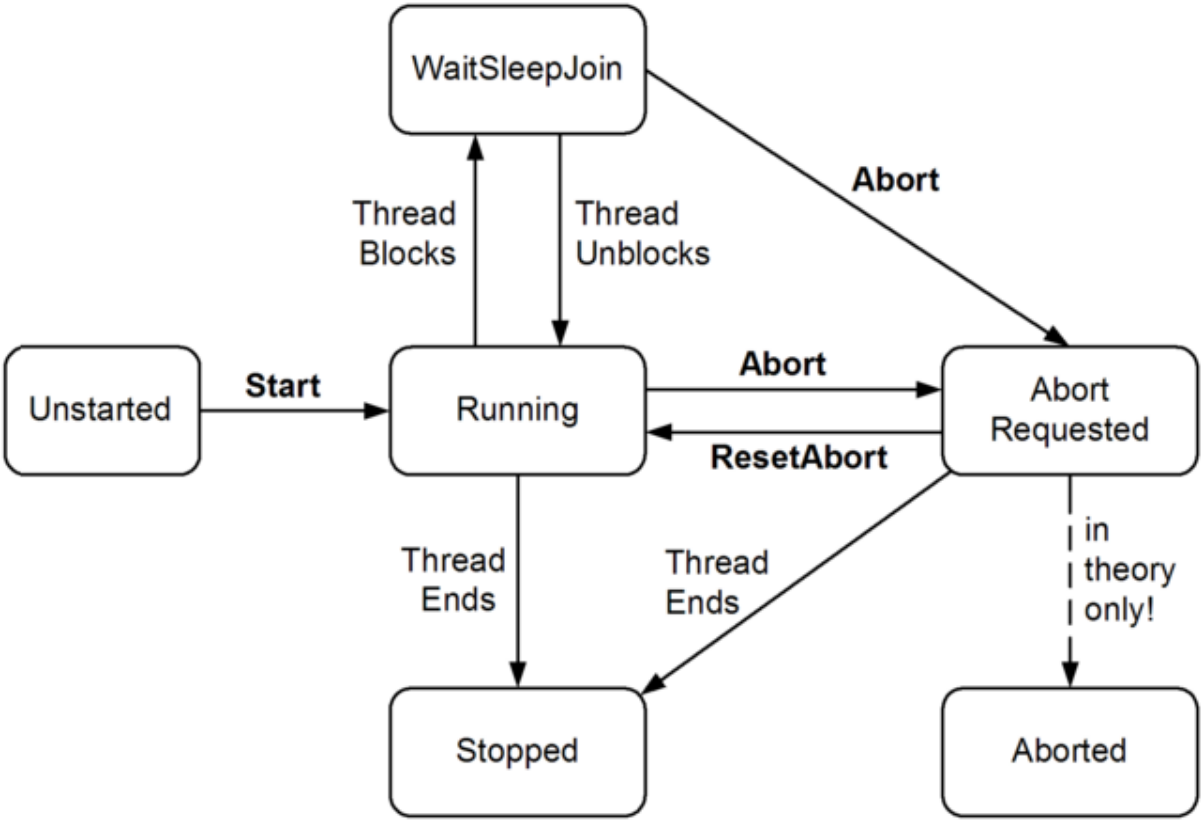
Thread State

A thread state can essentially be in one of four states

- Unstarted
- Running
- Wait/Sleep/Join - blocked
- Stopped

Blocked threads consume almost no processor time




WaitSleepJoin
Abort
Thread
Blocks
Thread
Unblocks
Unstarted
Start
Running
Abort
ResetAbort
Abort
Requested
Thread
Ends
Thread
Ends
in
theory
only!
Stopped
Aborted

Creating Threads and Scheduling Work

New Threads

If we want to start a new thread to execute a piece of code, we create an instance of type Thread and pass in a delegate containing the instructions to be executed by the thread.

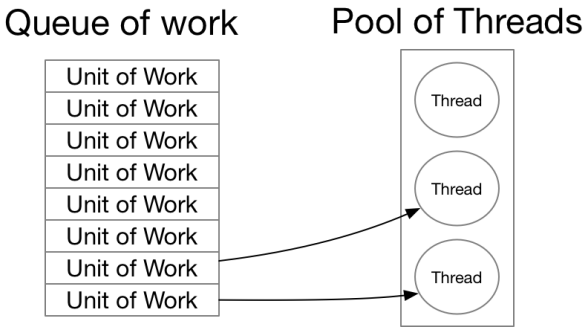
```
void Main()
{
    Thread t1 = new Thread(this.ThreadBodyA);
    Thread t2 = new Thread(this.ThreadBodyB);

    t1.Start();
    t2.Start("Test Message");
}

public void ThreadBodyA() { }
public void ThreadBodyB(object message) { WriteLine(message); }
```

Thread Pooling

One problem with creating a new Thread every time we want to execute code is that thread creation and destruction are expensive operations that introduce a lot of overhead. A thread pool combines a queue with a number of worker threads. The worker threads take work items off the queue and process them. Each .NET application has only one thread pool.



This pooling and recycling of threads is more efficient than continually creating and destroying threads explicitly. An intelligent thread pool will typically reduce the number of threads when the CPU is 100% and increase the number of threads when all threads on the pool are busy and the CPU is less than 50%. We can schedule work to be executed on a thread pool as follows.

Listing 1Scheduling work on the ThreadPool

```
ThreadPool.QueueUserWorkItem(Console.WriteLine, "Hello World");
```

As of .NET 4.0, a more common and powerful way to schedule work on the thread pool is via Tasks. We will cover tasks in more detail later but for now the following code executes a simple piece of work on a background thread.

Listing 2Using tasks with the thread pool

```
Task.Run(() => Console.WriteLine("Hello World"));
```

Uses of the thread pool

- User code
- Asynchronous delegates
- BackgroundWorker class
- System.Timer and System.Threading.Timer
- WCF, Remoting, ASP.NET and Web Services
- Tasks on default task scheduler

Timers

Thread Safety

A thread safe piece of code is one which behaves deterministically in the presence of multiple threads. Typically, non-deterministic behaviour is caused by a race condition. A race condition occurs when four conditions are met.

Four conditions for possible race condition

- Memory locations accessible from more than one thread
- Invariant is associated with shared memory
- Invariant does not hold for some part of update
- Another thread accesses the memory while invariant is broken

The following lists some ways in which we can implement thread safety

- Minimize or remove all shared state e.g. stateless application server
- Immutable objects. Read only objects fully initialized during construction
- Only use one thread to access state
- Synchronisation constructs to prevent concurrent access to blocks of code

The following example introduced a race condition and fixes it with a simple lock. The next section goes over different synchronisation constructs.

NON-THREAD SAFE EXAMPLE

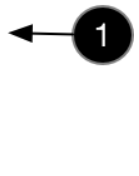
Consider the following piece of code which is not thread safe. The problem occurs when multiple threads call MakeDeposit. If one thread gets pre-empted after executing line one but before it can execute line two and a second thread executes MakeDeposit before the first thread gets re-scheduled, then effectively one of deposits gets lost.

Figure 1 Non thread-safe code

```
public class Account
 {
     public void MakeDeposit(double amount)
     {
         double newBalance = _balance + amount;
         _balance = newBalance;
     }

     public double CurrentBalance => _balance;

     private double _balance;
 }
```



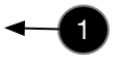
In order to prevent this kind of indeterminacy we need to implement a synchronization strategy. Synchronization is the topic of the next chapter but for now we show how to use a Monitor via the lock keyword to remove the race condition from this simple code.

Listing 3Locking Fixes Race condition

```
public class Account2
{
    public void MakeDeposit(double amount)
    {
        lock (this)
        {
            double newBalance = _balance +amount;
            _balance = newBalance;
        }
    }

    public double CurrentBalance => _balance;

    private double _balance;
}
```



Why General Purpose types are not usually type safe

- Performance overhead introduced even when type used in single threaded environment

- Thread safe type does not make program using it thread safe.
- Consider a type safe list used as follows in non type safe manner

```
if (!typeSafeList.Contains(newItem))
typeSafeList.Add(newItem);
```

SIMPLE STRATEGIES TO INCREASE THREAD SAFETY

- Only use one thread
- Sacrifice granularity by wrapping large sections on code inside lock/monitor
- E.g. single lock protects access to all fields of a type
- Minimize or remove all shared state e.g. stateless application server
- Immutable objects. Read only objects fully initialized during construction

CONTEXT BOUND OBJECT

- Automatic locking regime
- Whenever any method or property is called object wide lock is obtained
- Can cause unintended deadlocks and needlessly lessen concurrency

```
[Synchronization]
class MyContextBoundObject : ContextBoundObject
{
}
```

Locking Constructs

EXCUSIVE LOCKING CONSTRUCTS

Lock/Monitor

Monitor is the standard .net locking construct and the compiler supports its use via the lock keyword. Any reference type object can be used as a lock. There is a cost associated with acquiring a lock, however it is faster if the lock is uncontended than if contended as acquiring a contended lock requires blocking and context switching.

- Faster and more convenient than Mutex
- Only reference objects can be used as locks
- Acquiring an uncontended lock is faster than blocking and context switching with contention
- Most important methods are Monitor.Enter and Monitor.Exit
- TryEnter take a timespan and returns true if lock is obtained, false if times out
- Pulse/ Wait ??

Listing 4 Monitor Shortcut

```
public void MonitorShortcut()
{
  object lockA = new object();

  // This code
  lock (lockA)
  {
    // Do something
  }

  // Generates this code
  Monitor.Enter(lockA);
  try
  {
    // Do Something
  }
  finally { Monitor.Exit(lockA); }
}
```

Mutex

- Can span applications in different processes on same machine
- Can be used to ensure only one instance of a process can run at a tie
- Slower than lock/monitor

Spinlock

Specialized advanced exclusive locking construct that can improve performance in highlight contended environments.

NON-EXCLUSIVE LOCKING CONSTRUCTS

Semaphore

- Allows a specified number of threads into a critical section
- Once specified number of threads are inside other threads block
- Most important methods are WaitOne and Release
- Release is thread agnostic i.e. can be called by thread not holding the semaphore
- Could possibly be used to limit concurrent access to some resource e.g. disk

ReaderWriterLockSlim

- Allows concurrent reads but exclusive writes
  - Main methods are
  - EnterReadLock
  - ExitReadLock
  - EnterWriteLock
  - ExitwriteLock
- Thread holding write lock blocks all other threads trying to read or write
- If no thread contains a write lock any number of threads can obtain read lock
- A write lock is universally exclusive
- A read lock is compatible with other read lock
- Replaces older ReaderWriterLock class
- Older class is slower and more buggy so never use it
- Most important methods are WaitOne and Release
- Release is thread agnostic i.e. can be called by thread not holding the semaphore

LOCKING ISSUES

Deadlock

Acquiring locks incurs a performance cost. Perhaps more importantly though injudicious use of locking can lead to deadlock. The following code shows how

Figure 2Deadlock

Deadlock

```
public void DeadlockCreator()
{
    lock (_lockA)                    ← 1
    {
        new Thread(() =>
        {
            lock (_lockB)            ← 2
            {
                lock (_lockA)        ← 3
                {

                }
            }
        }).Start();

        Thread.Sleep(100);

        lock (_lockB)                ← 4
        {
        }

        Console.WriteLine("Finished");
    }
}
```

The following points give recommendation on reducing the likelihood of deadlock

REDUCING THE LIKELIHOOD OF DEADLOCK

- Deadlock can occur if a thread pool thread blocks waiting for async call to complete
- Do not ever block a thread executed on the thread pool waiting for another function on the pool
- Do not create any class whose synchronous methods wait for asynchronous functions, since this class could be called from a thread in a pool
- Do not ever block a thread executed on the pool waiting for another function on the pool
- Do not use any class inside an asynchronous function if the class block waiting for asynchronous functions

Limitations

- Protecting an object with a lock only works if all concurrent threads obtain the lock
- Particularly applicable to widely scoped usage e.g. static members
- In general .NET makes all static members thread safe whereas instance methods are not

Rich Clients

- Only the thread that instantiates a UI object can call any of its members
- In windows forms call Invoke or BeginInvoke

Non-locking synchronization constructs

OVERVIEWS

- Statement is intrinsically atomic if it executes as single indivisible instruction on CPU
- Strict atomicity precludes possibility of pre-emption
- Single read or assignment of field of 32bits or less is atomic on 32bit processor
- Operations on fields larger than the width processor are non atomic
- An alternative to using locks is to wrap non-atomic using Interlocked methods

```
namespace Threading
{
  class MyInterlocking
  {
    public MyInterlocking()
    {
      long sum = 4;
      Interlocked.Increment(ref sum);
      Interlocked.Decrement(ref sum);
      Interlocked.Add(ref sum, 4);
      Interlocked.Read(ref sum);
    }
  }
}
```

VOLATILE READS/WRITES

- Cache coherency means two threads think a field has different values at the same time
- Thread class offers static methods to read/write volatile memory with acquire semantics
- VolatileRead reads a value and invalidates the cache
- VolatileWrite write and flush the cache to main memory

WPF Threading Model

Consider the following piece of code which instantiates a minimal WPF user interface consisting of a single Window. This simple fragment highlights some important aspects of threading in WPF

Listing 5 Minimal WPF Interface

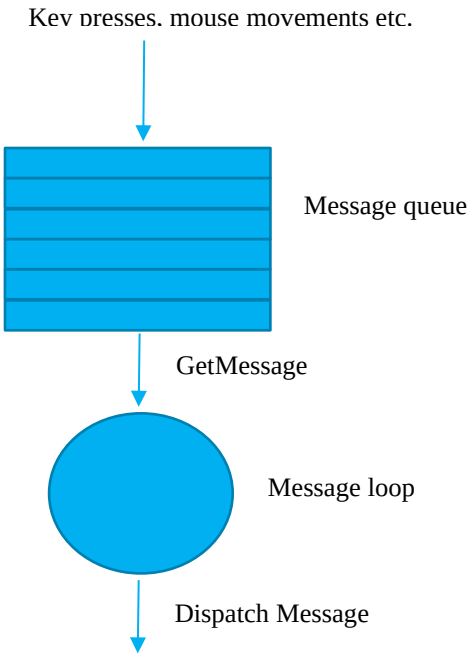
```
[STAThread] ❶
public static void Main()
{
  Window w = new Window();
  w.Show();

  Dispatcher.Run(); ❷
}
```

The first thing of note is that we require a Window. Every WPF user interface must consists of at least one window. This is in fact just a Win32 window. Like a Win32 window a WPF window must run inside a thread whose COM apartment state is `ApartmentState.STA` We achieve by marking with the correct attribute ❶.

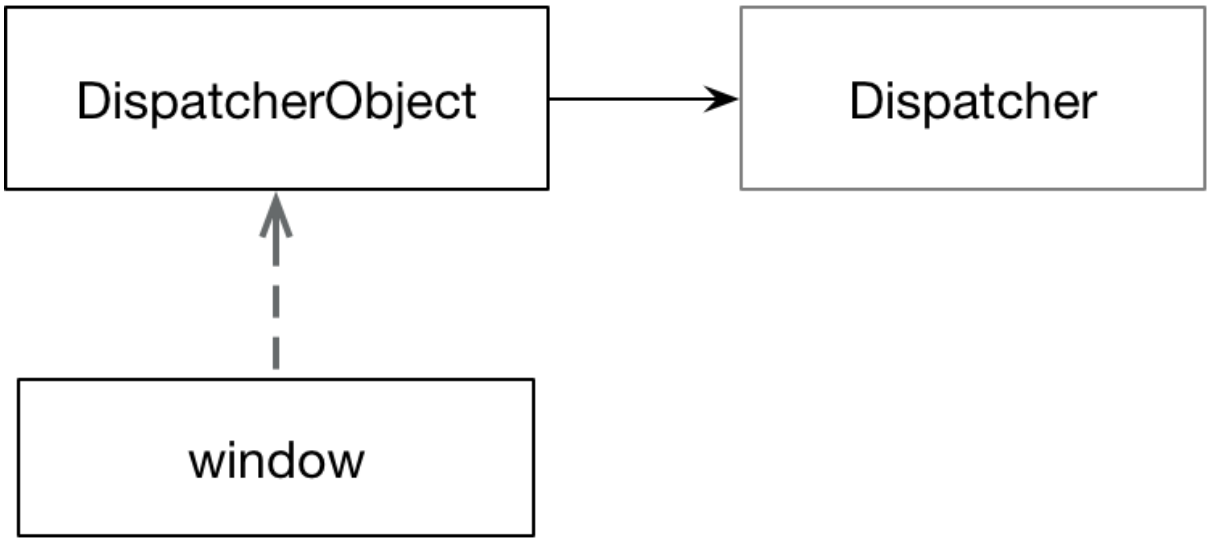
The second important point is that for the thread to be a UI thread it needs to have an associated `Dispatcher` which encapsulates a message loop. We set up the dispatcher by calling `Dispatcher.Run`❷ Inside this method WPF will setup the message loop. It is the message loop that makes the main thread a user interface thread. The message loop sits inside a loop, pulling messages from a message queue and dispatching them.

Figure 3Windows Message Loop



All WPF objects that inherit from `DispatcherObject` are associated with the same Dispatcher as their parent window. These objects are said to have thread affinity. Furthermore, these objects can only be safely updated on the thread with which they have affinity - the thread associated with their dispatcher. Any input messages from a window or its child `DispatcherObjects` such as keystrokes and mouse presses are pushed onto the message queue by the operating system. It is then the message loop's job to pull these from the queue, process them and dispatch them such that handlers can respond to them.

Figure 4DispatcherObject



If any thread other than the user interface thread wants to interact with DispatcherObjects' much do so by first obtaining the DependencyObject's dispatcher and then calling the Dispatcher's ❸ `BeginInvoke` method. The following piece of code highlights the approach

Listing 6 Dispatcher

```
[STAThread]
public static void Main()
{
    Window w = new Window();
    w.Loaded += WOnLoaded;
    w.Show();
    Dispatcher.Run();
}
private static void WOnLoaded(object sender, RoutedEventArgs
routedEventArgs)
{
    Window w = sender as Window;
    w.Background = new SolidColorBrush(Colors.Green);

    Task.Run(() =>
    {
          Thread.Sleep(2000);

          // Update the window on the thread it has affinity with
          Action d = () => w.Background = Brushes.NavajoWhite;
          w.Dispatcher.BeginInvoke(d, null); ❸
    });
}
```

> If we don't use the Dispatcher and instead try and directly set the background from the background thread then we get a runtime exception saying **System.InvalidOperationException: The calling thread cannot access this object because a different thread owns it**

SynchronizationContext

SynchronizationContext is abstract class defined in System.ComponentModel. WPF and WinForms define subclasses which you can get a handle to be calling SynchronizationContext.Current from a UI thread.

If one holds a reference to a SynchronizationContext one can post to it with similar effect to calling BeginInvoke on a Dispatcher. The advantage of the Synchronisation context is that it is not dependent on any framework. So, code that works with SynchronizationContext can be utilised in WPF and WinForms applications. Calling Post is equivalent to directly calling BeginInvoke on a Dispatcher and calling Send is equivalent to calling Invoke directly on a dispatcher

One key use of SynchronizationContext is when executing code in the background and then using the results to update a user control using Tasks

The following code shows how to update the UI thread safely using a synchronisation context rather than directly accessing the dispatcher.

Listing 7 SynchronizationContext

```
private static void WOnLoaded2(object s, RoutedEventArgs e)
{
    Window w = s as Window;
    w.Background = new SolidColorBrush(Colors.Green);

    SynchronizationContext synchronizationContext =
SynchronizationContext.Current;

    Task.Run(() =>
    {
          Thread.Sleep(2000);
          synchronizationContext.Post(state => w.Background =
Brushes.NavajoWhite, null);
    });
}
```

Listing 8SynchronizationContext Implementation

```
public class SingleThreadedSynchronizationContext : SynchronizationContext
{
    public SingleThreadedSynchronizationContext()
    {
        new Thread(() =>
                {
                    while (true)
                    {
                        try
                        {
                            _callbacks.Take()();
                        }
                        catch (Exception e)
                        {
                            Console.WriteLine(e);
                        }
                    }
                })
        { Name = "SingleThreadedSynchronizationContext" }
            .Start();
    }

    public override void Post(SendOrPostCallback d, object state)
    {
        _callbacks.Add(() => d(state));
    }

    public override void Send(SendOrPostCallback d, object state)
    {
        var tcs = new TaskCompletionSource<object>();

        _callbacks.Add(() =>
        {
            d(state);
            tcs.SetResult(null);
        });

        tcs.Task.Wait();
    }

    private readonly BlockingCollection<Action> _callbacks =
        new BlockingCollection<Action>();
}
```

Patterns

Obsolete patterns

Asynchronous Delegates

We can invoke a delegate asynchronously using its begininvoke method. The delegate will execute on the thread pool. Invoking EndInvoke frees any resources and retrives its return value.

```
Func<int, int> squareDelegate = (x) => x * x;
IAsyncResult ias = squareDelegate.BeginInvoke(2, null, null);
int result = squareDelegate.EndInvoke(ias);
Console.WriteLine(result);
```

Asynchronous delegates introduce significant overhead and one should wherever possible use tasks instead.

Asynchronous Programming Model

BackgroundWorker

Asynchronous delegates

Thread communication

Thread can communicate using simple blocking methods or signalling constructs and event wait handles

SIMPLE BLOCKING METHODS

- Waiting for another thread to finish E.g. Join, EndInvoke,
- Waiting for period of time to elapse

SIGNALLING CONSTRUCTS AND EVENT WAIT HANDLES

- Allow a thread to pause until receiving notification from another
- Avoids the need for inefficient polling
- Event Wait Handles

Tasks

Tasks separate the specification and co-ordination of units of work from the details of how they are scheduled. Unlike threads, it is easy to get a return value from a task. Tasks can be chained together by specifying that one task continues when another completes. Large operations can be formed by combining smaller ones.

Starting Tasks

Listing 9 Startting Tasks

```
❶
var t1 = Task.Factory.StartNew(() =>
Console.WriteLine("TaskFactory.StartNext"));

❷
var t2 = Task.Run(() => Console.WriteLine("Task.Run"));

❷
var t3 = new Task(() => Console.WriteLine("new Task()"));
t3.Start();
```

Unit of Work definition separate from scheduling

The following piece of code creates two separate but identical units of work and then schedules each one differently

Listing 10 Separting Scheduling and UoW

```
Console.WriteLine(Thread.CurrentThread.ManagedThreadId);

void Function()
{
    Console.WriteLine(Thread.CurrentThread.ManagedThreadId);
}

Task t1 = new Task(Function);
Task t2 = new Task(Function);

t1.Start(TaskScheduler.Default);
t2.Start(TaskScheduler.Current);
```

Chaining Tasks

Tasks can be chained together using Continuations. Continuations are set up using the `ContinueWith` method. The following code shows how to chain two tasks together. We have two very basic methods ❶,❷that carry out the work we want to do. We want ❷to execute once ❶is complete. We create a task ❸and pass in an Action delegate that invokes TaskOneFunction. We then add a continuation that will execute our second method. Note however we must add a wrapper method ❺ that takes the antecedent task and calls TaskTwoFunction

Listing 11Chaining One

```
void Main()
{
    MyExtensions.SetupLog4Net();
    logger.Info(nameof(Main));

    Task taskOne = ❸ new Task( new Action(TaskOneFunction));

    ❹
    taskOne.ContinueWith( new Action<Task>(TaskTwoFunctionWrapper));

    taskOne.Start();
}

❶
public void TaskOneFunction()
{
    logger.Info(nameof(TaskOneFunction));
}

❷
public void TaskTwoFunction()
{
    logger.Info(nameof(TaskTwoFunction));
}

❺
public void TaskTwoFunctionWrapper(Task antecedent)
{
    TaskTwoFunction();
}
```

CHAINING VALUE RETURNING TASKS

Now we move on to consider how to chain value returning tasks together. If our first task is of type `Task<int>` then the continuations delegate function must take an argument of type `Task<int>`

Listing 12 Chaining Two

```
void Main()
{
    MyExtensions.SetupLog4Net();

    Task<int> taskOne = new Task<int>( new Func<int>(TaskOneFunction));

    Task<int> continuation =taskOne.ContinueWith( new
Func<Task<int>,int>(TaskTwoFunctionWrapper));

    taskOne.Start();
    logger.Info(continuation.Result);
}


public int TaskOneFunction()
{
    logger.Info(nameof(TaskOneFunction));
    return 5;
}

public int TaskTwoFunction(int x)
{
    logger.Info(nameof(TaskTwoFunction));
    return x * 2;
}

public int TaskTwoFunctionWrapper(Task<int> antecedent)
{
    return TaskTwoFunction(antecedent.Result);
}
```

SUB-TASKS AND UNWRAP

Sometimes one task will want to use another task within its body. Where the tasks return values this can lead to a type of ❶`Task<Task<Tresult>>` which is rather inconvenient. We can get around this using the static method ❷Unwrap which creates a proxy task that only completes when the outer and inner tasks complete. It achieves this internally without blocking using callbacks.

Listing 13Unwrapping

```
Task<double> GetSpot() => Task.Run(() => 100.0);

Task<double> GetForward(double spot) => Task.Run(() =>spot
*Math.Exp(0.1));

❶Task<Task<double>> forward = GetSpot()
.ContinueWith(x => GetForward(x.Result));

forward
❷.Unwrap()
.ContinueWith(f => log.Info(f.Result));
```

The following example show how Unwrap and ContinueWith can be used to chain together value returing tasks where the output from one function forms the input to the next function

Listing 14Unwrap and ContinueWith

```
Task<int> Increment(int x)
{
      return Task.Run(() =>
    {
            log.Info(nameof(Increment));
            return ++x;
    });
}

var result = Increment(0)
.ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap().ContinueWith(f => Increment(f.Result))
.Unwrap();

result.ContinueWith(r => log.Info(r.Result));
```

Scheduling Tasks

The examples of creating and starting tasks in the previous section all utilised default scheduling which schedules tasks on a thread pool. If we want to specify the scheduler, we can use the following call.

Listing 15 Explicit Scheduling

```
TaskScheduler scheduler = TaskScheduler.Default;
Task t3 = new Task(() => Console.WriteLine("new Task()"));
t3.Start(scheduler);
```

Again, we use the default scheduler but this time we explicitly specify it. The following built in options are supported

Listing 16 Scheduling Options

```
TaskScheduler s1 = TaskScheduler.Default;

TaskScheduler s2 = TaskScheduler.Current;

TaskScheduler s3 = TaskScheduler.FromCurrentSynchronizationContext()
```

It is often useful to add a level of indirection to the schedulers used. In this way we can switch in different schedulers as suits our needs. I.e. we might want to use a single threaded scheduler rather than thread pool when unit testing. Often a team will create something along the lines of `ISchedulerProvider` to provide this extra level of indirection.

If the out of the box .NET schedulers don't do what you want you can always implement your own by sub classing TaskScheduler.

Listing 17 Custom Task Scheduler

```
class CurrentThreadScheduler : TaskScheduler
{
    protected override void QueueTask(Task t) =>
          TryExecuteTask(t);

    protected override bool TryExecuteTaskInline(Task t, bool b) =>
          TryExecuteTask(t);

    protected override IEnumerable<Task> GetScheduledTasks() =>
          Enumerable.Empty<Task>();
}
```

HOW SCHEDULERS WORK

Consider the following scheduler

CHAINING AND SCHEDULING EXAMPLE

Listing 18 Chaining and Scheduling Example

```
private void ButtonBase_OnClickAsyncAwait(object sender, RoutedEventArgs
e)
{
      ❶
    int Id() => Thread.CurrentThread.ManagedThreadId;

    Console.WriteLine($@"Handler {Id()}");

    var rateTask = new Task<double>(() =>
    {
          Thread.Sleep(1000);
          Console.WriteLine($@"Task1 {Id()}");
          return 0.1;
    });

    var fwdTask = rateTask.ContinueWith(task =>
    {
          Thread.Sleep(1000);
          Console.WriteLine($@"Task2 {Id()}");
          return 100 * Math.Exp(task.Result);
    }, TaskScheduler.Default); ❸

    fwdTask.ContinueWith(task =>
    {
          Console.WriteLine($@"Task3 {Id()}");
          TextBlock.Text = task.Result.ToString();
    }, TaskScheduler.FromCurrentSynchronizationContext()); ❹

    rateTask.Start(TaskScheduler.Default); ❷
}
```

❶ The button click itself is executed by the UI thread and as such has an associated SynchronizationContext associated with it. ❷The first task in the chain is started and explicitly scheduled on the default thread pool scheduler. ❸The first continuation is also explicitly scheduled on the default scheduler. ❹The final continuation uses the special `TaskScheduler.FromCurrentSynchronizationContext()` to execute the task on the SynchronizationContext captured at the point the final task is created.

If we want the second background task to execute on the same thread pool thread as the first background task we can use the `TaskContinuationOptions.ExecuteSynchronously` ❺option as follows

Listing 19 TaskContinuationOptions.ExecuteSynchronously

```
var fwdTask = rateTask.ContinueWith(task =>
{
```

```
        Thread.Sleep(1000);
        Console.WriteLine($@"Task2 {Id()}");
        return 100 * Math.Exp(task.Result);
    }, TaskContinuationOptions.ExecuteSynchronously); ❺
```

Exception handling

Another benefit of tasks is that any uncaught exception raised from task code is re-thrown such that it can be caught by the caller when it invokes Task.Wait or Task.Result

Listing 20 Exception Handling

```
public Task GetExceptionTask()
{
    return Task.Run(() =>
    {
          MyLogger.Log("exception raised");
          throw new ArgumentException();
    });
}

[Test]
public void TaskExceptionTest()
{
    try
    {
          var exceptionTask = GetExceptionTask();
          exceptionTask.Wait();
    }
    catch (Exception e)
    {
          MyLogger.Log($"Exception {e.GetType()}");
          MyLogger.Log($"Inner exception {e.InnerException.GetType()}");
    }
}
```

Because a task can cause multiple exceptions calling Wait or GetResult results in any exceptions being wrapped by an AggregateException.

Cancelling Tasks

The threading API provides two types that standardise cancellation

- CancellationTokenSource
- CancellationToken

By using two types the API seeks to separate the responsibilities of listening to and responding to cancellation events from the raising of cancellations. The following code shows a simple use case .

Listing 21 Cancelling Tasks

```
void Main()
{
    ❶
    var source = new CancellationTokenSource();

    ❷
    Task t = LongRunningAsyncMethod(source.Token);

    ❸
    source.CancelAfter(2);

    try
    {
          ❹
          t.Wait();
    }
    catch (Exception ex)
    {
          Console.WriteLine($"Caught {ex.InnerException}");
    }

    Console.WriteLine($"Final status: {t.Status}");
}

static async Task LongRunningAsyncMethod(CancellationToken token)
{                                                    ❷
    await Task.Delay(TimeSpan.FromSeconds(20), token);
}
```

❶ We start off by creating a CancellationTokenSource which we then pass into a long running operation ❷ that will take about 20 seconds to run. We then instruct the CancellationTokenSource to carry out a cancellation after 2 seconds ❸ and then carry out a wait on the task ❹. The output from executing this fragment is

```
Caught System.Threading.Tasks.TaskCanceledException: A task was canceled.
Final status: Canceled
```

Awaiters

Awaiters provide another way of specifying a continuation. In the following code the task inside the OnCompleted handler is a continuation which is this example is executed on the same thread that the task was executing on

Listing 22 Basic Awaiter Example

```
void Main()
{
    ❶
    MyExtensions.SetupLog4Net();
    logger.Info("Main() started");

    Task<double> task = GetSpotPrice();

    TaskAwaiter<double> awaiter = task.GetAwaiter();

    awaiter.OnCompleted(() =>
    {
          ❸
          logger.Info(awaiter.GetResult());
    });
}


public Task<double> GetSpotPrice()
{
    return Task.Run(() =>
    {
          ❷
          logger.Info("Task running");
          return 110.0;
    });
}
```

The code at points ❷and ❸execute on the same thread which is different from the main thread that ❶executes on.

Awaiters are types that meet the following criteria.

1. Implement `InotifyCompletion`
2. Contain a property `public bool IsCompleted`
3. Contain a method `public TResult GetResult()`

So we could create our own noddy awaiter for our own noddy type as follows.

Listing 23 Implementing Awaiters

```
async void Main()
{
    MyExtensions.SetupLog4Net();
    logger.Info("Main() started");

    AnAwaitableType<double> awaitable =
          new AnAwaitableType<double>(100.0);

    AwaiterImplementation<double> awaiter = awaitable.GetAwaiter();
    awaiter.OnCompleted(() =>
    {
          logger.Info($"{awaiter.GetResult()}");
    });
}

public class AnAwaitableType<T>
{
    public AnAwaitableType(T value) => _value = value;

    public AwaiterImplementation<T> GetAwaiter() =>
          new AwaiterImplementation<T>(_value);

    private T _value;
}

public class AwaiterImplementation<TResult> : INotifyCompletion
{
    public AwaiterImplementation(TResult value) => _value = value;

    public bool IsCompleted => true;

    public void OnCompleted(Action continuation) => continuation();

    public TResult GetResult() => _value;

    private TResult _value;
}
```

The default task awaiters have two interesting properties. First if a synchronization context is present at the point the continuation is the registered it will be captured and the continuation will be posted to that context. In the following code the onloaded handler is running on the UIThread

Listing 24Capturing SynchronizationContext

```
private static void WOnLoaded(object s, RoutedEventArgs e)
{
    Task<double> task = GetSpotPrice();

    TaskAwaiter<double> awaiter = task.GetAwaiter();

    awaiter.OnCompleted(() => logger.Info(awaiter.GetResult()));
}

public static Task<double> GetSpotPrice()
{
    return Task.Run(() =>
    {
          logger.Info("Task running");
          return 110.0;
    });
}
```

Because the WonLoaded handler is running on the UIThread the output is as follows.

```
[Main Query Thread] Main() started
[5] Task running
[Main Query Thread] 110
```

The second important point regards exceptions. If a task throws an exception then the awaiter unwraps the exception from the Agreggate Exception. Of course this means if the AggregateException had more than one inner exception only the first is returned and the others are lost.

Listing 25 TaskAwaiter unwraps exceptions

```
    awaiter.OnCompleted(() =>
    {
          try
          {
                logger.Info(awaiter.GetResult());
          }
          catch (Exception ex)
          {
                logger.Info($"{ex.GetType().Name}");
          }

    });

…

public Task<double> GetSpotPrice()
{
    return Task.Run(() =>
    {
```

```
            throw new Exception("Something happened");
            return 0.0;
        });
    }
```

If we want the code to give us the original AggregateException we can write our own awaiter to replace the use of the default Task awaiter. X

Async Await

C #5 added support for asynchronous functions which are methods or anonymous functions whose declaration contains the `async` keyword. The code inside the async method can utilize the `await` keyword (asynchronous wait) to make asynchornous code look like synchronous code. The compiler genenerated state machine allows one to write much cleaner and simpler code that does away with the need to write explicit callbacks and specialized multithreaded exception handling.

The basic idea is as follows. Imagine a piece of code that calls a long running function that returns a task. We can define the method with the async keyword and then internally we can use the `await` keyword as follows.

Listing 26 Simple Async Await

```
private async void ButtonBase_OnClick(object sender, RoutedEventArgs e)
{
Console.WriteLine($@"{nameof(ButtonBase_OnClick)}:
{Thread.CurrentThread.ManagedThreadId}");

Task<int> task = LongRunningTaskAsync();
task.Start(TaskScheduler.Default); ❶

int result = await task;❷

Console.WriteLine($@"{nameof(ButtonBase_OnClick)}:
{Thread.CurrentThread.ManagedThreadId}");❸
}

public Task<int> LongRunningTaskAsync()
{
  return new Task<int>(() =>
  {
    Console.WriteLine($@"LongRunningTaskAsync:
{Thread.CurrentThread.ManagedThreadId}");
    return 100;
  });
}
```

The button handler is already running on a UI thread. When we kick of the long running task on a background thread ❶we immediately call await on it. ❷The code looks like the method will block at this stage but what actually happens in the method returns and a continuation is setup to execute the remainder of the method once the long running task completes. The async/await mechanism has logic to capture the synchronization context if once exists and hence the continuation is posted to the correct UI thread ❸

How Async/Await works

Consider a very basic async method called `GetDoubleAsync` that awaits a task returned as the result of the method `GetLongRunningTask`. LongRunningTask represents an operation whose result will sometimes be available immediately and sometime will require an expensive background calculation. For educational purposes we pass in a flag to show which path is being executed. We want to observe the call stack and thread of execution. In the general case the code after an await is a continuation and hence it does not have the caller of the async method in its stack trace.

Listing 27 LongRunningTaskAsync

```
public Task<double> LongRunningTaskAsync(bool completeImmediately)
{
    if (completeImmediately)
    {
          Log(nameof(LongRunningTaskAsync));
          return Task<double>.FromResult(100.0);
    }

    return Task<double>.Run(() =>
    {
          Log(nameof(LongRunningTaskAsync));
          return 100.0;
    });
}
```

TASK EXECUTES ON POOL AS LONG RUNNING

Consider now the case where the LongRunningTask requires to calculate its value on the threadpool

```
private async Task<double> GetDoubleAsync()
{
    ❶Log(nameof(GetDoubleAsync));
     double result = await LongRunningTaskAsync(false);
    ❷Log(nameof(GetDoubleAsync));
     return result;
}
```

The output for a given run is then as follows Notice the two parts of the method; before and after the await call both execute on the thread associated with the synchronization context and the long running task executes on a background thread

```
GetDoubleAsync 20
LongRunningTaskAsync 21
GetDoubleAsync 20
```

The stack trace at point ❶ looks as follows. Notice how it still has the caller frame.

```
UserQuery.GetDoubleAsync()
UserQuery.<<Main>b__0_0>d.MoveNext()
```

At point❷however the calling method has disapeared from the stack trace. We just have compiler generated code on the stack. This is the code the compiler will have generated to implement async await

```
  UserQuery.<GetDoubleAsync>d__1.MoveNext()
AsyncMethodBuilderCore.MoveNextRunner.InvokeMoveNext(Object stateMachine)
```

TASK EXECUTES IMMEDIATELY

Now consider the case where the long running task result is available immediately.

```
private async Task<double> GetDoubleAsync()
{
    ❶Log(nameof(GetDoubleAsync));
     double result = await LongRunningTaskAsync(true);
    ❷Log(nameof(GetDoubleAsync));
     return result;
}
```

The output becomes as follows showing that everying executes on the same synchronization context thread.

```
GetDoubleAsync 12
LongRunningTaskAsync 12
GetDoubleAsync 12
```

And our stack trace then become as follows. First before the async await ❶ note the caller Main is still on the stack

```
UserQuery.<GetDoubleAsync>d__1.MoveNext()
CompilerServices.AsyncTaskMethodBuilder`1.Start[TStateMachine](TStateMach
ine& stateMachine)
UserQuery.GetDoubleAsync()
UserQuery.<Main>b__0_0()
```

And after the async the caller is still on the stack because no continuation was needed due to the ❷long running task completing immediately

```
UserQuery.<GetDoubleAsync>d__1.MoveNext()
CompilerServices.AsyncTaskMethodBuilder`1.Start[TStateMachine](TStateMach
ine& stateMachine)
UserQuery.GetDoubleAsync()
UserQuery.<Main>b__0_0()
```

COMBINING AWAIT STATEMENTS

Consider the following piece of code which needlessly reduced concurrency

```
public async Task SomeEventHandler()
{
    Log(1, $"{nameof(SomeEventHandler)} Before await");
    double rate = await GetPresentValue( 1.0);
}

public async Task<double> GetPresentValue(double t)
{
    Log(2, $"{nameof(GetPresentValue)} Before await");
    var pv = await GetFutureValue() * Math.Exp(await GetRate() * t);
    Log(5,$"{nameof(GetPresentValue)} After await");
    return pv;
}

public Task<double> GetRate()
{
    return Task<double>.Run(() =>
    {
          Log(4, $"{nameof(GetRate)}");
          return 0.1;
    });
}

public Task<double> GetFutureValue()
{
    return Task<double>.Run(() =>
    {
          Log(3, $"{nameof(GetFutureValue)}");
          return 100.0;
    });
}

public void Log(int num, String s)
{
    var onSc = sc == null ? false : sc.OnSynchThread();
    Thread t = Thread.CurrentThread;
    Console.WriteLine($"{num} ({t.ManagedThreadId},{onSc}) {s}");
}
```

The logging statements show the thread they are executing on. We see the output as follows.

```
1 (23,True) SomeEventHandler Before await
2 (23,True) GetPresentValue Before await
3 (5,False) GetFutureValue
4 (5,False) GetRate
5 (23,True) GetPresentValue After await
```

IMPLEMENTING ASYNC/AWAIT WITH AWAITERS

We can show how we might go about writing our own async await implementation using awaiters. Note this is easy for very simple methods but quickly becomes very hard in complex methods hence the beauty of having the compiler generate state machines for us

```
private Task<double> GetDoubleAwaiter()
{
    Log(nameof(GetDoubleAwaiter));

    // Get the awaiter
    TaskAwaiter<double> awaiter =
LongRunningTaskAsync(true).GetAwaiter();

    // Special case when the result is already available
    if (awaiter.IsCompleted)
    {
          Log(nameof(GetDoubleAwaiter));
          return Task<double>.FromResult(awaiter.GetResult());
    }

    // General case requires us to register a continuation
    TaskCompletionSource<double> tcs = new
TaskCompletionSource<double>();
    awaiter.OnCompleted(() =>
    {
          Log(nameof(GetDoubleAwaiter));
          tcs.SetResult(awaiter.GetResult());
    });

    return tcs.Task;
}
```

WRITING OUR OWN AWAITERS

Tasks come with an implementation of awaiters but we can create our own awaiters for types. For a type to be used on the rhs of an await expression it must have a method called GetAwaiter (or have a suitable extension method) that returns a type that implements a specific pattern

Questions

THREADING BASICS

What is a thread?

Single sequential flow of control within a single address space

Multiple threads read/write same memory locations

Each thread has separate call stack with own local variables

Programmer ensures shared memory accessed in correct way

Why would we use multi-threaded programming?

Multiple separate processes in separate address spaces expensive to set up and cost of communicating high

Driving slow devices such as disks, terminals, printers

Performing RPC to remote server

Maintain responsive user interface

Scalability – taking advantage of multiple CPUs and multiple remote servers

What threading facilities does .NET provide?

Thread creation

Join

Mutual exclusion

What are the main problems with multi-threaded programming?

Only guarantee is that bugs will never show until in production

Non-deterministic behaviour

Rarely enough information at the point of failure to debug the application

Increased complexity

Non-reproducible bugs

Cost in allocating and switching threads

Compare and contrast a thread and a process?

Processes run in parallel on a computer and threads run in parallel in a process

Processes are fully isolated from each other and threads only have limited isolation

Threads share heap memory with other threads running in the same app domain

What is meant by the term preemption?

A threads execution is interspersed with execution of code on another thread

What happens when a C# client application starts?

A single main thread in automatically created by the CLR and OS

What is the primary cause of complexity and obscure errors in multithreading?

Shared data

When can multiple threads improve performance?

Only if the application is I/O bound OR

Computer has multiple CPU's

Why limit the number of threads in a process?

Creating a thread is not cheap

Destroying a thread is not cheap

Context switching is expensive

Why is creating a thread expensive?

1MB of address space for the user-mode stack

12K of address space for the kernel-mode stack

Ever .DLL needs notification of thread creation

Why is destroying a thread expensive?

Every .DLL needs notification

Kernel object and stack need to be freed

Why is context switching expensive?

Enter kernel mode

Save CPU registers into current threads kernel object

Determine next thread to schedule

Load CPU registers from about to run threads kernel object

Level kernel mode

EXECUTING CODE ASYNCHRONOUSLY

What are the disadvantages of creating and starting threads for every asynchronous task?

Creating and destroying threads is expensive

Write code to create and start a new thread

```
void Main()
{
    Thread t1 = new Thread(this.ThreadBodyA);
    Thread t2 = new Thread(this.ThreadBodyB);

    t1.Start();
    t2.Start("Test Message");
}

public void ThreadBodyA() { }
public void ThreadBodyB(object message) { WriteLine(message); }
```

What is the difference between foreground and background threads?

Foreground threads keep the application alive so long as one of them is running

Background threads abruptly terminate as soon as all foreground threads end

What is the only situation where multiple threads share local variables?

Local variables captured in an anonymous method passed as the delegate to a thread constructor.

What is returned by a threads isAlive property?

Once started returns true until the threads ends

When does a thread end?

When the delegate passed to the threads constructor finishes executing

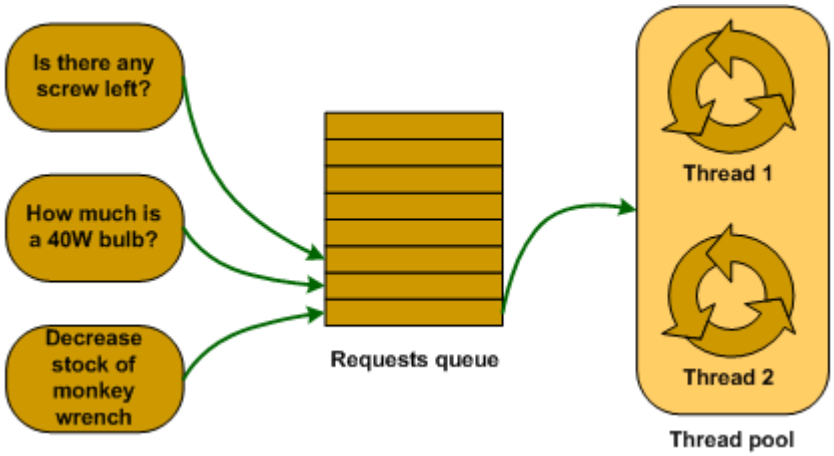
Can you restart a thread once it ends?

- No

What is a thread pool?

Combines queue with number of worker threads

Worker threads process requests off the queue



What are the benefits of a thread pool?

Reduces overhead of creating and destroying threads

What are the features of an intelligent thread pool?

Worker threads process requests off the queue

Remove threads when CPU is 100% to prevent thrashing

Add threads where all are busy and CPU < 50% as most likely existing work items are I/O bound

What are the features of the .NET thread pool?

One per process

All methods are static

Methods queued as follows

Write code to execute work on the thread pool?

```
  ThreadPool.QueueUserWorkItem(new WaitCallback(this.DoWork));
}

public void DoWork(object state) {}
```

What is the signature of WaitCallback?

```
public delegate WaitCallback( object state ) ;
```

TIMERS

What are tbe vailable types of Timer

System.Threading.Timer

System.Timers.Timer

System.Windows.Forms.Timer (Windows Form)

System.Windows.Threading.DispatcherTimer (WPF)

What are the main differences between the first two and last two?

The first two are general purpose multi-threaded timers

The second two are special purpose single threaded timers used when updating Windows forms controls or WPF elements

Characteristics of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?

Used when updating Windows form controls or WPF Elements

Doesn't use thread pool

Always fires on same thread that creates it

Advantages of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?

Thread-safe

New tick will never fire until previous one runs

Can update UI controls directly from Tick event handler

Disadvantages of System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer?

Thread-safe

Can impact UI responsiveness if tick handler isn't lightweight

Less accurate because can be delayed while other UI requests are processed

When to use System.Windows.Forms.Timer and System.Windows.Threading.DispatcherTimer

Small job that involves updating some aspect of user interface

E.g. clock or countdown display

Characteristics of System.Threading.Timer

Simplest multi-threaded timer

Uses the thread pool

Callback may fire on any different threads

Callbacks or event handlers must be thread safe

What are the characteristics of System.Timers.Timer?

Wrapper around System.Threading.Timer

Adds convenience methods and uses the underlying implementation

Who use .NET thread pool?

User code

Asynchronous delegates

BackgroundWorker helper class

System.Timer and System.Threading.Timer

WCF, Remoting, ASP.NET and Web Services

Tasks using default task scheduler

Does each app domain have its own thread pool?

No there is only one thread pool per process

What two types of thread does the .NET thread pool use?

Worker threads – execute user functions and timers

Completion Port threads – Used for IO operations where possible

What are worker threads?

Manager by .NET framework

Execute user functions and timers

Write code to queue a method for asynchronous execution on a thread pool?

What advantage do asynchronous delegates have over thread pools queue work items?

Typed method arguments

THREAD SAFETY

Describe thread safety?

A thread safe piece of code is one which behaves deterministically in the presence of multiple threads of execution

What are the general approaches to thread safety?

Locking constructs to prevent concurrent access to a block of code

Remove or minimise access to shared heap memory in multi-threaded scenarios

What are the two exclusive locking constructs in .NET?

Monitor and Mutex

What is the difference between the two?

Mutex can be used to lock across multiple processes on the same computer

Which of the two back the lock construct?

Monitor

Write code to show what the compiler generates when you use the lock keyword

```
private object _myLock = new Object();

void Main()
{
    Monitor.Enter(_myLock);
    try
    {
          // Do some work
    }
    finally
    {
          Monitor.Exit(_myLock);
    }
}
```

What four conditions must be met needed for race to be possible?

Memory locations accessible from more than one thread

Invariant is associated with shared memory needed

Invariant does not hold for some part of update

Another thread accesses the memory while invariant is broken

Is the code totalRequests++ thread safe?

The compiler will generate three assembler instructions to load the value into a register, increment is and write it back

Synchronization Constructs

What is needed to ensure program correctness?

A correct program should ensure all memory falls into three buckets

- Thread exclusive
- Read only
- Lock Protected

What is the main criticism of locks?

They provide mutual exclusion for regions of code, but generally programmers want to protect regions of memory

What is a lock?

- Mutual exclusion
- Simple resource scheduling mechanism
- Resource is shared memory inside lock keyword
- Policy is one thread at a time

What is a lock also known as?

- Critical section
- Binary semaphore

What is the eternal trade of when determining granularity of locking schemes?

- Course provides correctness and sequential performance
- Fine provides parallel scalability

How is a lock implemented in C#?

- In C# implemented by Monitor class
- Monitor.Enter causes all other threads to wait until Monitor.Exit called

What is a Mutex and how does it differ from a monitor?

- A Mutex is given a name and applies across processes
- Might be used to ensure only one instance of a process runs at a time
- Acquiring/Releasing Mutex 50 times slower than a Monitor

What is a ReaderWriterLock/ReaderWriterLockSlim?

- Allows multiple readers concurrent access
- Writers get exclusive access
- Thread holding a write lock blocks all other threads
- If no thread holds write lock multiple threads can concurrently access a read lock

Why is locking expensive?

- Processors need to co-operate
- Taking lock can mean another processor needs to flush pending writes so current thread sees all updates
- If an IO bound operation holds a lock it can lead to inefficient use of CPU, especially in multi processor environments

What is meant by the term re-entrant?

A thread that has entered a lock can enter a lock again without blocking

What code with the compiler generate when it sees the lock keyword?

```
public void MonitorShortcut()
{
  object lockA = new object();

  // This code
  lock (lockA)
  {
    // Do something
  }

  // Generates this code
  Monitor.Enter(lockA);
  try
  {
    // Do Something
  }
  finally { Monitor.Exit(lockA); }
}
```

DEADLOCKS

What four conditions must hold for deadlock to occur?

Mutual Exclusion – a thread owns resource, other can't acquire

Holder of resource allowed to perform unbounded wait

Resources cannot be forcibly removed from current owners

Circular wait condition exists

How can deadlock be prevented?

Have few enough locks that its never necessary to take more than one at a time

Have a convention on order in which locks are take (levels)

WPF THREADING MODEL

What does a thread need to be a WPF UI thread?

for the thread to be a UI thread it needs to have an associated `Dispatcher` *which encapsulates a message loop. We set up the dispatcher by calling* `Dispatcher.Run`

What is thread infinity?

WPF objects which extend DispatcherObject can only be safely updated on the thread with which they have affinity - the thread associated with their dispatcher

How does one safely update a DispatcherObject from a thread other than the one on which it has infinity?

By obtaining its Dispatcher and calling Dispatcher.Invoke on it w.Dispatcher.BeginInvoke(d, null); Even better use SynchronizationContext.Post or SynchronizationContext.Send

What is the difference between BeginInvoke and Invoke?

Invoke blocks the calling thread until the passed delegate completes it work

why shouldn't one use Dispatcher.Invoke?

Dispatcher.Invoke blocks the calling thread until the Dispatchers associated thread executes the given delegate. The following examples show when using invoke causes a deadlock that is fixed by using BeginInvoke

What is a SynchronizationContext?

An abstraction for thread safe update of UI components. In WPF delegates to an underlying Dispatcher

What method on Synchronization is equivalent to Dispatcher.Invoke and which to Dispatcher.BeginInvoke?

Post is equivalent to BeginInvoke and Send is equivalent to Invoke.

Thread Communication

What is the most common method of communication between thread?

Shared memory

What is the effect of the join method?

Causes the calling thread to block until the given thread ends

Why is it rarely used in practice?

Because most threads are daemons, have no result or communicate using some synchronizatin construct

What are EventWaitHandled used for?

Signaling when one thread waits until it receives notification from another

When a thread calls WaitOne() it will bolock until another thread calls set()

Where might you use a wait handle?

An initialization mwethod

what is the difference between a ManualResetEvent and an AutoResetEvent?

Calling set on a ManualResetEvent causes all blocking threads through

Calling set on a AutoResetEvent causes only one blocking thread through

Tasks

What is a task?

Tasks separate the specification and co-ordination of units of work from the details of how they are scheduled. Unlike threads, it is easy to get a return value from a task. Tasks can be chained together by specifying that one task continues when another completes. Large operations can be formed by combining smaller ones.

Why use Tasks?

Allow us to define units of works separate from their execution scheduling

Tasks are compositional meaning we can chain them together.

Exceptions thrown by the unit of work are re-thrown to callers of Wait/Result

Why does one not, in general need to dispose of Tasks?

ASYNC AWAIT

Can you see any problem with the following code?

```
public async Task<double> GetPresentValueInefficient(double t)
{
    var pv = await GetFutureValue() * Math.Exp(await GetRate() * t);
    return pv;
}
```

The await calls are needlessly synchronized. One will wait for the other to complete

Improve the code from the following question?

```
public async Task<double> GetPresentValueEfficient(double t)
{
    var fvTask = GetFutureValue();
    var rateTask = GetRate();

    var pv = await fvTask * Math.Exp(await rateTask* t);
    return pv;
}
```

Write code using an awaiter to achieve the same result as this code?

```
public async Task<double> GetPresentValue(double t)
{
    double rate = await GetRate();
    return rate;
}

public Task<double> GetRateAwaiter()
{
    Task<double>  rateTask = GetRate();
    TaskAwaiter<double> rateAwaiter = rateTask.GetAwaiter();
    TaskCompletionSource<double> tcs = new
TaskCompletionSource<double>();

    if ( rateAwaiter.IsCompleted)
    {
          return Task<double>.FromResult(rateAwaiter.GetResult());
    }
    else
    {
          rateAwaiter.OnCompleted(() =>
        {
                tcs.SetResult(rateAwaiter.GetResult());
        });
    }

    return tcs.Task;
}
```