Concurrent Programming

Agenda

- Concurrent vs. Parallel
- Processes and Threads
- Reasons to Use Threads
- Using a Dedicated Thread
- Thread Scheduling and Priorities
- CLR Thread Pool
- Execution Contexts
- Cooperative Cancellation and Timeout

Concurrent Programming

Agenda

- Tasks and the Task Parallel Library (TPL)
  - Creating and Starting
  - Cancellation and Options
  - Completion and Return Values
  - Exceptions
  - Debugging
  - Task Schedulers
  - For, ForEach, and Invoke

Concurrent Programming

Agenda

- Periodic Operations
- I/O-Bound Asynchronous Operations
- Async and Await
- Thread Synchronization Constructs

Concurrent Programming

Concurrent vs. Parallel

- Concurrent - More than one operation in in progress at a given time
  - Operations may or may not be executing at the same time
- Parallel - More than one operation is executing at the same time
  - Number of operations executing in parallel is limited by the number of processor cores (CPUs) available

Concurrent Programming

Processes and Threads

- In 16-bit Windows, there existed only one thread of execution
  - Executed both operating system and application code
- With the Windows NT kernel, each application in run in what is called a process
  - A process is a collection of resources including a virtual address space and a thread of execution
- The operating system manages the scheduling of threads to run on available CPUs
  - A thread is the smallest sequence of programmed instructions that can be managed by the OS scheduler

Processes and Threads

Multi-CPU Technologies

- Multiple CPUs - Motherboard has multiple sockets on it, with each socket containing a CPU
- Hyper-threaded CPU - Allows a single CPU to look to Windows like two
  - Windows will schedule two threads simultaneously and thinks they are running in parallel
  - Only one thread is actually running at a given time
  - Windows is hyper-threading aware and will schedule execution across multiple hyper-threaded CPUs intelligently
- Multi-core CPU - A single computing component with two or more actual CPUs

Processes and Threads

Thread Overhead

- Threads have space (memory) and time overhead
- Every thread has one of each of the following:
  - Thread kernel object (1 MB) - Contains properties that describe the thread and the thread's context (block of memory that contains a set of CPU registers)
    - When the processor core changes from executing one thread to executing a different thread, this is referred to as a context switch
  - Thread environment block (4 KB) - Contains the thread's thread-local storage and the head of the thread's exception handling chain

Processes and Threads

Thread Overhead

- User-mode stack (1 MB) - Memory reserved for local variables and arguments passed to methods
- Kernel-mode stack (24 KB) - Memory used when application code passes arguments to a kernel-mode function in the OS
  - For security reasons, Windows copies arguments from the user-mode stack to the kernel-mode stack prior to use by the kernel

Processes and Threads

Thread Overhead

- In addition to memory overhead, there is also the overhead of thread-attach and thread-detach notifications
- Whenever a thread is created in a process, every unmanaged DLL loaded in that process has its DllMain method called with either a DLL_THREAD_ATTACH or DLL_THREAD_DETACH flag
- The performance impact of this can vary greatly from one application to another
  - As an example, Visual Studio 2012 can have more that 450 DLLs loaded into its process at a given time!

Thread Overhead

Context Switches

- At any given moment, Windows assigns one thread to a CPU
- A thread is allowed to run for a time-slice (aka quantum)
- When a time-slice expires, Windows context switches to another thread
  - Windows performs context switches approximately every 30 milliseconds
  - A thread can voluntarily end its time-slice early

Thread Overhead

Context Switch Procedure

1. Windows saves the values in the CPU's registers to the currently running thread's context structure inside the thread's kernel object
2. Selects another thread from the set of existing threads to schedule next. If owned by another process, Windows must also switch the virtual address space seen by the CPU
3. Loads the values in the selected thread's context structure into the CPU's registers

Thread Overhead

Garbage Collection

- When performing a garbage collection, the CLR must suspend all the threads of the application and walk their stacks
- In .NET 4 and later, background garbage collection is available
    - Runs on a dedicated thread and selectively suspends other application threads as needed
    - Can result in a more responsive application at the cost of more context switches (memory and CPU usage)

Thread Overhead

Debugging

- When using the Visual Studio debugger, Windows suspends all threads in the application being debugged every time a breakpoint is hit
- Resumes all threads when you single-step of run the application
- The more threads you have, the slower your debugging experience will be

Processes and Threads

CLR Threads and Windows Threads

- Originally, the CLR team felt they would someday have the CLR offer logical threads which did not necessarily map directly to Windows threads
  - Idea was later abandoned - a CLR thread will always be identical to a Windows thread
  - Some API methods still exist to deal with the notion that a CLR thread might not map to a Windows thread (i.e. System.Diagnostics.ProcessThread)
- For Windows Store apps, Microsoft has removed some APIs related to threading (i.e. entire System.Thread class is unavailable)

Thread Overhead

Summary

- Threads consume memory and require time to create, destroy, and manage
- Time is wasted when Windows context switches between threads and when garbage collection occurs

Concurrent Programming

Reasons to Use Threads

- Responsiveness
  - For a GUI application, any long-running operation that occurs on the main thread can cause the application to stop responding to user events
- Scalability
  - For a web application, worker processes have a limited number of threads available and using an outside thread can free the original thread to handle another incoming request
- Performance
  - If more than one CPU is available, tasks can be performed in parallel

Concurrent Programming

Using a Dedicated Thread

- You can create a dedicated thread in .NET to perform an asynchronous compute-based operation
- Provides the maximum amount of flexibility and should be considered when...
  - You need a thread to run with a non-normal priority
  - You need a thread to behave as a foreground thread
  - The operation is extremely long-running and you don't want to monopolize a thread pool thread

Concurrent Programming

Using a Dedicated Thread

- To create a dedicated thread, construct an instance of the System.Threading.Thread class and pass the method to run in the thread into its constructor

```
Thread t = new Thread(ComputeBoundOp);
```

- Method supplied must accept a single parameter of type Object and not return a value

```
private void ComputeBasedOp(Object state)
```

Concurrent Programming

Using a Dedicated Thread

- Creating the Thread object does not create the actual underlying Windows thread
- Occurs when the Thread object's Start method is called

```
t.Start(obj);
```

- A call to a thread's Join method can be used to stop executing any code in the calling thread until the target thread has destroyed itself or been terminated

```
t.Join();
```

Lab 1

Using a Dedicated Thread

- Build an application that creates and uses a dedicated thread to perform an asynchronous compute-based operation

Concurrent Programming

Thread Scheduling and Priorities

- A preemptive OS must use some kind of algorithm to determine which threads should be scheduled to run and for how long
- After a time-slice, Windows looks at all of the thread kernel objects in existence
    - Threads that are not waiting for something are considered eligible to be scheduled
- Windows keeps a record of how many times a thread has been context switched to
    - Viewable using Spy++

Thread Scheduling and Priorities

Thread Priorities

- Every thread is assigned a priority
  - Ranges from 0 (lowest) to 31 (highest)
- Windows examines all of the highest priority threads first and schedules them in a round-robin fashion
- If higher priority schedulable threads always exist, lower priority threads will not be given an opportunity to run
  - Condition know as starvation
  - Less likely to occur on multiprocessor machines
- Higher priority threads can preempt lower priority threads even when the lower priority thread is in the middle of a time-slice

Thread Scheduling and Priorities

Process Priority

- Managing thread priority values directly would be very difficult
- To help simply things for application developers, Windows supports six process priority classes to define how applications relate to each other
  - Idle
  - Below Normal
  - Normal - Default (recommend for all .NET applications)
  - Above Normal
  - High - Should only be used when absolutely necessary
  - Realtime - Reserved for things like short latency hardware

Thread Scheduling and Priorities

Relative Thread Priorities

- For threads within a process, a relative thread priority can be used which is relative to the process priority
  - Idle - Not available for .NET applications
  - Lowest
  - Below Normal
  - Normal - Default
  - Above Normal
  - Highest
  - Time-Critical - Not available for .NET applications (used by the CLR's finalizer thread)

Thread Scheduling and Priorities

Relative Thread Priorities

	Idle	BN	Normal	AN	High	Real-Time
Time-Critical	15	15	15	15	15	31
Highest	6	8	10	12	15	26
AN	5	7	9	11	14	25
Normal	4	6	8	10	13	24
BN	3	5	7	9	12	23
Lowest	2	4	6	8	11	22
Idle	1	1	1	1	1	16

Thread Scheduling and Priorities

Relative Thread Priorities

- Your application can change the relative thread priority of a thread by settings the thread's Priority property

```
t.Priority = ThreadPriority.AboveNormal;
```

- The System.Diagnostics namespace contains a Process and ProcessThread class (not available in ASP.NET applications)

```
int i = Process.GetCurrentProcess.BasePriority;
```

Thread Scheduling and Priorities

Foreground vs. Background Threads

- The CLR considers every thread to be either a foreground thread or a background thread
- When all foreground threads in a process stop running, the CLR forcibly ends any background threads that are still running
  - No exceptions are thrown
- Threads created by the application's primary thread are created as foreground threads
- Threads from the thread pool are background threads

Concurrent Programming

CLR Thread Pool

- To provide a way to avoid the performance overhead of creating threads, the CLR maintains a pool of threads that your application can use
- When asking the thread pool to perform an asynchronous operation, an existing thread from the pool will be used
  - If all thread pool threads are in use, a new thread will be created and added to the pool
- Information about the thread pool can be obtained via properties of the ThreadPool class

CLR Thread Pool

Performing a Compute-Based Operation

- To queue an asynchronous operation to the thread pool, you can call the QueueUserWorkItem method of the ThreadPool class

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp);
```

Lab 2

CLR Thread Pool

- Modify the previous lab application to use a thread pool thread instead of a dedicated thread

Concurrent Programming

Execution Contexts

- Every thread has an execution context data structure associated with it
- Includes such things as security settings, host settings, and logical call context data
- Ideally, whenever a thread uses another thread to perform tasks, the issuing thread's execution context should flow (be copied) to the helper thread
  - By default, the CLR provides for this but there is a performance impact

Execution Contexts

Suppressing Execution Context Flow

- By using a method of the ExecutionContext class, it is possible to prevent the flow of an execution context to another thread
- Can increase performance in server applications if execution context information is not needed in the other thread

```
ExecutionContext.SuppressFlow();
ThreadPool.QueueUserWorkItem(Foo);
ExecutionContext.RestoreFlow();
```

Concurrent Programming

Cooperative Cancellation and Timeout

- The .NET Framework provides a standard pattern for canceling operations
- Pattern is cooperative meaning that the operation has to explicitly support being cancelled
- Object of type CancellationSourceToken used to provide CancellationToken instances
- An operation queries the IsCancellationRequested property of a CancellationToken

Cooperative Cancellation and Timeout

Cooperative Cancellation

- You can use the CancellationToken's Register method to register a method that will be called when the cancel is called on the CancellationTokenSource

```
var token = cts.Token;
token.Register(WasCancelled);
```

Cooperative Cancellation and Timeout

Timeout

- CancellationTokenSource has constructors that accept a delay parameter and provides a CancelAfter method
- Both will cause the CancellationTokenSource to cancel itself after the specified delay

```
var cts = new CancellationTokenSource(5000);
```

```
cts.CancelAfter(5000);
```

Lab 3

Cooperative Cancellation

- Implement an asynchronous operation that supports cancellation

Concurrent Programming

Tasks

- Using QueueUserWorkItem has its limitations
  - No built-in way to know when an operation has completed
  - No way to get the return value of the method
- To address these limitations, Microsoft introduced the concept of tasks

```
ThreadPool.QueueUserWorkItem(ComputeBoundOp, "data");
```

```
new Task(ComputeBoundOp, "data").Start();
```

```
Task.Run(() => ComputeBoundOp("data"));
```

Tasks

Cancellation and Options

- When creating a task, you can optionally pass a CancellationToken and TaskCreationOptions flags
    - PreferFairness - You want this task to run as soon as possible
    - LongRunning - Hint to the TaskScheduler for use of thread pool threads
    - AttachedToParent - Associates a task with its parent task
    - DenyChildAttach - Does not allow other tasks to attach
    - HideScheduler - Forces child tasks to use the default scheduler opposed to the parent's scheduler

Tasks

Completion and Return Values

- It is possible to construct a Task(Of TResult) where TResult is the return type of the operation
- The return value of the operation can then be obtained via the Result property of the task
  - If Result is called before the operation is complete, the thread calling Result will block until it is available
  - It is also possible to explicitly wait for an operation to complete by calling task.Wait( )

Tasks

Exceptions

- If a task's operation throws an unhandled exception, the exception will be swallowed, stored in a collection, and the thread is allowed to return to the pool
- When Wait( ) or Result is called, an AggregateException will be thrown in the calling thread at that time
  - InnerExceptions property returns the collection of exceptions

Lab 4

Task Results

- Build an application to perform a compute-bound task that returns a value

Tasks

WaitAny and WaitAll

- Task contains two static methods that allow a thread to wait on on array of Task objects
- WaitAny blocks the calling thread until any of the Task objects in the array have completed
  - Returns an index into the array indicating which Task object completed
- WaitAll blocks the calling thread until all the Task objects in the array have completed
  - Method returns false if a timeout occurred

Tasks

Canceling a Task

- You can use a CancellationToken to cancel a Task
- The Task code must check the token to determine if a cancellation has been requested
- Typical pattern is to throw an OperationCancelledException
  - Makes it easier to differentiate between a completed task and a cancelled one
- If a Task is cancelled before it is scheduled for execution, an exception will not be thrown (Task simply does not run)

Lab 5

Task Cancellation

- Modify the previous lab to support cancellation of the Task

Tasks

Starting a New Task Automatically

- In order to write scalable software, you must not have your threads block
- Calling Wait or querying a task's Result property when the task is not finished is not ideal
- There is a better way to find out when a task has completed
- When a task completes, you can have it automatically start another task

```
task.ContinueWith(task => Console.WriteLine("Done"));
```

Tasks

Starting a New Task Automatically

- If the task completes before ContinueWith is called, the call to ContinueWith will notice that the task is already complete and start the second task immediately
- You can call ContinueWith multiple times for a given task
- You can specify additional options when calling ContinueWith
  - NotOnRanToCompletion, NotOnFaulted, NotOnCancelled, OnlyOnCancelled, OnlyOnRunToCompletion, etc.

Lab 6

Task Cancellation

- Modify the previous lab using ContinueWith

Tasks

Debugging

- Each Task object has a unique ID
- Task IDs start at 1 and are increment by 1 as each ID is assigned
- Visual Studio shows task IDs in its Parallel Tasks and Parallel Stacks windows
- When running code in the debugger, you can query Task's static CurrentId property

Tasks

Debugging

- Each Task object has a Status property
  - Created
  - WaitingForActivation
    - Initial state for Tasks created via ContinueWith
  - WaitingToRun
  - Running
  - WaitingForChildrenToComplete
  - RanToCompletion
  - Canceled
  - Faulted

Tasks

Task Schedulers

- A TaskScheduler object is responsible for executing scheduled tasks and exposes task information to the debugger
- The FCL ships with two TaskScheduler-derived types
  - Thread pool task scheduler
  - Synchronization context task scheduler

Tasks

Thread Pool Task Scheduler

- Used by default
- Schedules tasks to the thread pool's worker threads

Tasks

Synchronization Context Task Scheduler

- Typically used for applications with a graphical user interface
- Schedules tasks onto the application's GUI thread so the task code can successfully update the UI
- Does not use the thread pool
- Obtained using TaskScheduler's static FromCurrentSynchronizationContext method

```
scs = TaskScheduler.FromCurrentSynchronizationContext();
```

Tasks

For, ForEach, and Invoke

- There are some common programming scenarios that can potentially benefit from the improved performance possible with tasks
- The static class System.Threading.Tasks.Parallel encapsulates these common scenarios
    - Uses Task objects internally

Tasks

Parallel.For

- It is possible to have multiple thread pool threads assist in the processing of items in a collection

```
for (int i = 0; i < 1000; i++) DoWork(i);
```

```
Parallel.For(0, 1000, i => DoWork(i));
```

Tasks

Parallel.ForEach

```
foreach (var p in People) CalculatePay(p);
```

```
Parallel.ForEach(People, p => CalculatePay(p));
```

Tasks

Parallel.Invoke

- Parallel.Invoke can be used to execute any collection of methods in parallel

```
Method1();
Method2();
Method3();
```

```
Parallel.Invoke(
    () => Method1(),
    () => Method2(),
    () => Method3());
```

Tasks

For, ForEach, and Invoke

- For all of the Parallel methods, the calling thread participates in the processing of the work
- If the calling thread finishes its work before the other threads have completed, the calling thread will suspend itself until all the work is done
- If any operation throws an unhandled exception, the Parallel method will ultimately throw an AggregateException

Tasks

For, ForEach, and Invoke

- When using the Parallel methods, there is an assumption that it is okay for the work items to be performed concurrently
- Avoid work items that modify shared data
    - Adding synchronization locks would avoid corruption but eliminates the benefit of processing items in parallel

Periodic Operations

System.Threading.Timer

- The System.Threading namespace defines a Timer class
- Can be used to have a thread pool thread call a method periodically
- Internally, the thread pool has one just one thread it uses for all Timer objects
  - Calls QueueUserWorkItem when a method is due to be called

```
Timer timer = new Timer(DoStuff, null, 1000, 5000);
```

I/O-Bound Asynchronous Operations

Introduction

- All examples so far have been compute-bound asynchronous operations
  - Work efficiently
- Performing I/O-bound operations asynchronously allows hardware devices to handle the tasks so that threads and CPU are not used at all
  - Wait efficiently
  - Files, databases, networking

I/O-Bound Asynchronous Operations

Files

- When using a synchronous I/O method such as FileStream.Read, the system will put the calling thread to sleep while the operation is in progress
- Calling thread does not consume CPU but does consume memory
- Does make UI non-responsive if thread is a UI thread

I/O-Bound Asynchronous Operations

Files

- When using an asynchronous I/O method such as FileStream.ReadAsync, a Task object is created and returned
- You can call ContinueWith to register a callback method
- Alternatively, you can use the new async and await keywords

Async and Await

Introduction

- .NET 4.5 adds a feature called asynchronous functions
- Uses Tasks internally
- Simplifies programming of I/O-bound operations
    - Web access (HttpClient, SyndicationClient)
    - Files (StorageFile, StreamReader, XmlReader)
    - Images (MediaCapture, BitmapEncoder)
    - Database access (SqlCommand, SqlDataReader)

Async and Await

Introduction

- The async and await keywords allow you to write asynchronous methods much the same as you would write synchronous ones

```
async Task<int> AccessTheWebAsync()
{
  HttpClient client = new HttpClient();

  string urlContents =
    await client.GetStringAsync("http://msdn.microsoft.com");

  return urlContents.Length;
}
```

Async and Await

Introduction

- If you have independent work to perform while the operation is in progress, you can call Await later in the method

```
async Task<int> AccessTheWebAsync()
{
  HttpClient client = new HttpClient();

  Task<string> getStringTask =
    client.GetStringAsync("http://msdn.microsoft.com");

  DoIndependentWork();

  string urlContents = await getStringTask;

  return urlContents.Length;
}
```

Async and Await

API Async Methods

- You can recognize members in the .NET Framework that support async programming by looking for the Async suffix attached to the member name and a Task return type
- System.IO.Stream
  - CopyTo, CopyToAsync
  - Read, ReadAsync
  - Write, WriteAsync

Async and Await

Threads

- The async and await keywords don't cause additional threads to be created
- You can use Task.Run to move CPU-bound work to a background thread but a background doesn't help with a process that's just waiting for results to become available

Async and Await

Capabilities

- By marking a method with Async, you enable two capabilities:
  - The method can use Await to designate suspension points
    - Tells the compiler that the method can't continue past that point until the awaited asynchronous process is complete
  - The method can itself be awaited by methods that call it
- An async method typically contains one or more occurrencs of an Await operator
  - Not using Await will cause the method to run synchronously and generate a compiler warning

Async and Await

Waiting in Parallel

- When performing multiple high-latency operations, you can significantly improve overall performance by using Task.WhenAll
- The operations appear to run in parallel but no additional threads are created
- For situations where multiple sources are available for the same data, Task.WhenAny can be used
  - 3rd party web services

Thread Synchronization Constructs

Introduction

- Thread synchronization is used to prevent corruption when multiple threads access shared data at the same time
- For async functions thread synchronization is not required when code accesses data contained within the async function
- A thread synchronization lock is used prevent a section of code from being executed by multiple threads at the same time
- Locks hurt performance
  - Takes time to acquire and release a lock
  - CPUs must coordinate with each other
  - Failure to acquire a lock can cause the thread pool to create additional threads

Thread Synchronization Constructs

Avoid

- Thread synchronization is costly and complex
- Design your applications to avoid it as much as possible
- Avoid shared data such as static fields
- When a thread uses the new operator, an object reference is returned
  - Only the constructing thread has a reference to the object
  - If you avoid passing this reference to another thread, there is no need to synchronize access to the object
- Value types are always copied, so each thread operates on its own copy
- Synchronization is not needed if all access is read-only

Thread Synchronization Constructs

Thread Safety

- A method is considered to be thread safe if it can safely be called by multiple threads simultaneously
  - Does not necessarily mean the method uses locks
- All static methods in the .NET Framework Class Library are guaranteed to be thread safe
  - Non-static methods should be assumed to be not thread safe
  - It is recommended that you follow the same practice in your own class libraries

Thread Synchronization Constructs

Atomic Reads and Writes

- The CLR guarantees that reads and writes to variables of type Boolean, Char, Byte, Int16, Int32, IntPtr, and Single are atomic
- Other types (e.g. Int64, Double, etc.) could result in a torn read
  - Possible for the value to be read in an intermediate state

Thread Synchronization Constructs

Mutex

- A mutex represents a mutual-exclusive lock
- Works like a semaphore with a count of one
- Records the ID of the thread that obtained the mutex
- When ReleaseMutex is called, ensures it is the same thread that obtained the mutex

Thread Synchronization Constructs

Monitor

- The Monitor class provides a more full-featured and efficient mutex
- Every object of a reference type has a data structure called a sync block associated with it
- Monitor is a static class whose methods accept a reference type object and manipulate that object's sync block

Thread Synchronization Constructs

Monitor

```
public class MyObject
  private readonly obj = new Object();

  public void Foo()
  {
    Monitor.Enter(obj);
    // Do something
    Monitor.Exit(obj);
  }
}
```

Thread Synchronization Constructs

Monitor

- When using Monitor, you must ensure the lock is released
  - Throwing an exception prior to Monitor.Exit is problematic

```
public void Foo()
{
  Monitor.Enter(obj);
  try
  {
    // Do something
  }
  finally
  {
    Monitor.Exit(obj);
  }
}
```

Thread Synchronization Constructs

SyncLock

- The lock statement in C# can be used to make the use of Monitor easier

```
public void Foo()
  lock (obj)
  {
    // Do something
  }
}
```

Thread Synchronization Constructs

ReaderWriterLockSlim

- Sometimes, it is okay for many threads to perform read operations with an object but write operations require exclusive access
- ReaderWriterLock was introduced in .NET 1.0
- ReaderWriterLockSlim was introduced in .NET 3.5 and should now always be used instead of ReaderWriterLock

Thread Synchronization Constructs

ReaderWriterLockSlim

```
public class MyObject
  private readonly rwls = new ReaderWriterLockSlim();

  public void ReadSomething()
  {
    rwls.EnterReadLock();
    // Do read
    rwls.ExitReadLock();
  }

  public void WriteSomething()
  {
    rwls.EnterWriteLock();
    // Do write
    rwls.ExitWriteLock();
  }
}
```

Thread Synchronization Constructs

Concurrent Collection Classes

- Very often, shared state gets introduced in the form of a collection that multiple threads would like to access to add or remove items
- .NET includes four thread-safe collection classes in the System.Collections.Concurrent namespace
  - ConcurrentQueue
  - ConcurrentStack
  - ConcurrentDictionary
  - ConcurrentBag

Thread Synchronization Constructs

Concurrent Collection Classes

- Collections are as efficient as they can be
- ConcurrentQueue and ConcurrentStack are lock-free
  - Use Interlocked methods to manipulate the collection
- ConcurrentDictionary uses a Monitor internally
- ConcurrentBag consists of a collection in each thread
  - A Monitor is only used if an object is requested that is not present in that thread's collection