**Application Performance**

Modern processors use threads to concurrently run multiple operations. If your application performs all of its logic on a single thread, you do not make the best use of the available processing resources, which can result in a poor experience for your users. In this module, you will learn how to improve the performance of your applications by distributing your operations across multiple threads.

At the same time, applications will sometimes rely on long running processes either locally or on a server over a network connection.  Application performance can also be seen through the eyes of a user in terms of responsiveness.  Through the use of asynchronous operations, you can create a responsive user interface that allows the user to continue interaction with the application while long running processes execute in the background and then return a notification when the operation is complete.  
  
**MultiTasking**

A typical graphical application consists of blocks of code that run when an event occurs; these events fire in response to actions such as the user clicking a button, moving the mouse, or opening a window. By default, this code runs by using the UI thread. However, you should avoid executing long-running operations on this thread because they can cause the UI to become unresponsive. Also, running all of your code on a single thread does not make good use of available processing power in the computer; most modern machines contain multiple processor cores, and running all operations on a single thread will only use a single processor core.

The Microsoft® .NET Framework now includes the Task Parallel Library. This is a set of classes that makes it easy to distribute your code execution across multiple threads. You can run these threads on different processor cores and take advantage of the parallelism that this model provides. You can assign long-running tasks to a separate thread, leaving the UI thread free to respond to user actions.

Prior to Windows Store applications, .NET developers made use of the System.Threading namespace and Thread class to execute multiple threads within an application.  The Thread class is not available in Windows Store apps so you make use of the Task class instead.  Creating a task in a Windows Store application will result in a thread being created for that task.

When you use tasks rather than threads, the CLR can optimize the number of threads required for concurrent tasks and this helps your applications scale better than dealing with threads directly.

**Creating Tasks**

The Task class lies at the heart of the Task Parallel Library in the .NET Framework. As the name suggests, you use the Task class to represent a task, or in other words, a unit of work. The Task class enables you to perform multiple tasks concurrently, each on a different thread. Behind the scenes, the Task Parallel Library manages the thread pool and assigns tasks to threads. You can implement sophisticated multitasking functionality by using the Task Parallel Library to chain tasks, pause tasks, wait for tasks to complete before continuing, and perform many other operations.

You create a new Task object by using the Task class. A Task object runs a block of code, and you specify this code as a parameter to the constructor. You can provide this code in a method and create an Action delegate that wraps this method.

*Note: A delegate provides a mechanism for referencing a block of code or a method. The Action class is a type in the .NET Framework Class Library that enables you to convert a method into a delegate. The method cannot return a value, but it can take parameters. The .NET Framework Class Library also provides the Func class, which enables you to define a delegate that can return a result.*

The following code example shows how to create a task by using an Action delegate:

// Creating a Task by Using an Action Delegate  
Task task1 = new Task(new Action(GetTheTime));  
private static void GetTheTime()  
{  
   Console.WriteLine("The time now is {0}", DateTime.Now);  
}

Using an Action delegate requires that you have defined a method that contains the code that you want to run in a task. However, if the sole purpose of this method is to provide the logic for a task and it is not reused anywhere else, you can find yourself creating (and having to remember the names of) a substantial number of methods. This makes maintenance more difficult. A more common approach is to use an anonymous method. An anonymous method is a method without a name, and you provide the code for an anonymous method inline, at the point you need to use it. You can use the delegate keyword to convert an anonymous method into a delegate.

The following code example shows how to create a task by using an anonymous delegate.

// Creating a Task by Using an Anonymous Delegate  
Task task2 = new Task( delegate { Console.WriteLine("The time now is {0}", DateTime.Now); });

**Using Lambda Expressions to Create Tasks**

A lambda expression is a shorthand syntax that provides a simple and concise way to define anonymous delegates. When you create a Task instance, you can use a lambda expression to define the delegate that you want to associate with your task.

If you want your delegate to invoke a named method or a single line of code, you use can use a lambda expression. A lambda expression provides a shorthand notation for defining a delegate that can take parameters and return a result. It has the following form:

(input parameters) => expression

In this case:

* The lambda operator, =>, is read as “goes to.”
* The left side of the lambda operator includes any variables that you want to pass to the expression. If you do not require any inputs—for example, if you are invoking a method that takes no parameters—you include empty parentheses () on the left side of the lambda operator.

The right side of the lambda operator includes the expression you want to evaluate. This could be a comparison of the input parameters—for example, the expression (x, y) => x == y will return true if x is equal to y; otherwise, it will return false. Alternatively, you can call a method on the right side of the lambda operator.

The following code example shows how to use lambda expressions to represent a delegate that invokes a named method.

// Using a Lambda Expression to Invoke a Named Method  
Task task1 = new Task ( () => MyMethod() );  
// This is equivalent to: Task task1 = new Task( delegate(MyMethod) );

A lambda expression can be a simple expression or function call, as the previous example shows, or it can reference a more substantial block of code. To do this, specify the code in curly braces (like the body of a method) on the right side of the lambda operator:

(input parameters) => { Visual C# statements; }

The following code example shows how to use lambda expressions to represent a delegate that invokes an anonymous method.

// Using a Lambda Expression to Invoke an Anonymous Method  
Task task2 = new Task( () => { Console.WriteLine("Test") } );  
// This is equivalent to: Task task2 = new Task( delegate { Console.WriteLine("Test") } );

As your delegates become more complex, lambda expressions offer a far more concise and easily understood way to express anonymous delegates and anonymous methods. As such, lambda expressions are the recommended approach when you work with tasks.

**Controlling Task Execution**  
  
The Task Parallel Library offers several different approaches that you can use to start tasks. There are also various different ways in which you can pause the execution of your code until one or more tasks have completed.

**Starting Tasks**

When your code starts a task, the Task Parallel Library assigns a thread to your task and starts running that task. The task runs on a separate thread, so your code does not need to wait for the task to complete. Instead, the task and the code that invoked the task continue to run in parallel.

If you want to queue the task immediately, you use the Start method.

// Using the Task.Start Method to Queue a Task  
var task1 = new Task( () => Console.WriteLine("Task 1 has completed.") );  
task1.Start();

Alternatively, you can use the static TaskFactory class to create and queue a task with a single line of code. The TaskFactory class is exposed through the static Factory property of the Task class.

// Using the TaskFactory.StartNew Method to Queue a Task  
var task3 = Task.Factory.StartNew( () => Console.WriteLine("Task 3 has completed.") );

The Task.Factory.StartNew method is highly configurable and accepts a wide range of parameters. If you simply want to queue some code with the default scheduling options, you can use the static Task.Run method as a shortcut for the Task.Factory.StartNew method.

// Using the Task.Run Method to Queue a Task  
var task4 = Task.Run( () => Console.WriteLine("Task 4 has completed. ") );

**Waiting for Tasks**

In some cases, you may need to pause the execution of your code until a particular task has completed. Typically you do this if your code depends on the result from one or more tasks, or if you need to handle exceptions that a task may throw. The Task class offers various mechanisms to do this:

* If you want to wait for a single task to finish executing, use the Task.Wait method.
* If you want to wait for multiple tasks to finish executing, use the static Task.WaitAll method.
* If you want to wait for any one of a collection of tasks to finish executing, use the static Task.WaitAny method.

The following code example shows how to wait for a single task to complete.

// Waiting for a Single Task to Complete  
var task1 = Task.Run( () => LongRunningMethod() );  
// Do some other work.  
// Wait for task 1 to complete.  
task1.Wait();  
// Continue with execution.

If you want to wait for multiple tasks to finish executing, or for one of a collection of tasks to finish executing, you must add your tasks to an array. You can then pass the array of tasks to the static Task.WaitAll or Task.WaitAny methods.

The following code example shows how to wait for multiple tasks to complete.

// Waiting for Multiple Tasks to Complete  
Task[] tasks = new Task[3]  
{  
   Task.Run( () => LongRunningMethodA()),  
   Task.Run( () => LongRunningMethodB()),  
   Task.Run( () => LongRunningMethodC())  
};  
// Wait for any of the tasks to complete.  
Task.WaitAny(tasks);  
// Alternatively, wait for all of the tasks to complete.  
Task.WaitAll(tasks);  
// Continue with execution.

**Returning a Value from a Task**  
  
For tasks to be effective in real-world scenarios, you need to be able to create tasks that can return values, or results, to the code that initiated the task. The regular Task class does not enable you to do this. However, the Task Parallel Library also includes the generic Task<TResult> class that you can use when you need to return a value.  
  
When you create an instance of Task<TResult>, you use the type parameter to specify the type of the result that the task will return. The Task<TResult> class exposes a read-only property named Result. After the task has finished executing, you can use the Result property to retrieve the return value of the task. The Result property is the same type as the task’s type parameter.  
  
The following example shows how to use the Task<TResult> class.  
  
// Retrieving a Value from a Task  
// Create and queue a task that returns the day of the week as a string.  
Task<string> task1 = Task.Run<string>( () => DateTime.Now.DayOfWeek.ToString() );  
// Retrieve and display the task result.  
Console.WriteLine(task1.Result);  
  
If you access the Result property before the task has finished running, your code will wait until a result is available before proceeding.  
**Cancelling Long-Running Tasks**  
  
Tasks are often used to perform long-running operations without blocking the UI thread, because of their asynchronous nature. In some cases, you will want to give your users the opportunity to cancel a task if they are tired of waiting. However, it would be dangerous to simply abort the task on demand, because this could leave your application data in an unknown state. Instead, the Task Parallel Library uses cancellation tokens to support a cooperative cancellation model. At a high level, the cancellation process works as follows:

1. When you create a task, you also create a cancellation token.
2. You pass the cancellation token as an argument to your delegate method.
3. On the thread that created the task, you request cancellation by calling the Cancel method on the cancellation token source.
4. In your task method, you can check the status of the cancellation token at any point. If the instigator has requested that the task be cancelled, you can terminate your task logic gracefully, possibly rolling back any changes resulting from the work that the task has performed.

Typically, you would check whether the cancellation token has been set to canceled at one or more convenient points in your task logic. For example, if your task logic iterates over a collection, you might check for cancellation after each iteration.

The following code example shows how to cancel a task.

// Cancelling a Task  
// Create a cancellation token source and obtain a cancellation token.  
CancellationTokenSource cts = new CancellationTokenSource();  
CancellationToken ct = cts.Token;  
// Create and start a task.  
Task.Run( () => doWork(ct) );  
// Method run by the task.  
private void doWork(CancellationToken token)  
{  
   …  
   // Check for cancellation.  
   if(token.IsCancellationRequested)  
   {  
      // Tidy up and finish.  
      …  
      return;  
   }  
   // If the task has not been cancelled, continue running as normal.  
   …  
}

This approach works well if you do not need to check whether the task ran to completion. Each task exposes a Status property that enables you to monitor the current status of the task in the task life cycle. If you cancel a task by returning the task method, as shown in the previous example, the task status is set to RanToCompletion. In other words, the task has no way of knowing why the method returned—it may have returned in response to a cancellation request, or it may simply have completed its logic.

If you want to cancel a task and be able to confirm that it was cancelled, you need to pass the cancellation token as an argument to the task constructor in addition to the delegate method. In your task method, you check the status of the cancellation token. If the instigator has requested the cancellation of the task, you throw an OperationCanceledException exception. When an OperationCanceledException exception occurs, the Task Parallel Library checks the cancellation token to verify whether a cancellation was requested. If it was, the Task Parallel Library handles the OperationCanceledException exception, sets the task status to Canceled, and throws a TaskCanceledException exception. In the code that created the cancellation request, you can catch this TaskCanceledException exception and deal with the cancellation accordingly.

To check whether a cancellation was requested and throw an OperationCanceledException exception if it was, you call the ThrowIfCancellationRequested method on the cancellation token.

The following code example shows how to cancel a task by throwing an OperationCanceledException exception.

// Canceling a Task by Throwing an Exception  
// Create a cancellation token source and obtain a cancellation token.  
CancellationTokenSource cts = new CancellationTokenSource();  
CancellationToken ct = cts.Token;  
// Create and start a task.  
Task.Run( () => doWork(ct) );  
// Method run by the task.  
private void doWork(CancellationToken token);  
{  
   …  
   // Throw an OperationCanceledException if cancellation was requested.  
   token.ThrowIfCancellationRequested();     
   // If the task has not been cancelled, continue running as normal.  
   …  
}

**Running Tasks in Parallel**  
  
The Task Parallel Library includes a static class named Parallel. The Parallel class provides a range of methods that you can use to execute tasks simultaneously.

**Executing a Set of Tasks Simultaneously**

If you want to run a fixed set of tasks in parallel, you can use the Parallel.Invoke method. When you call this method, you use lambda expressions to specify the tasks that you want to run simultaneously. You do not need to explicitly create each task—the tasks are created implicitly from the delegates that you supply to the Parallel.Invoke method.

The following code example shows how to use the Parallel.Invoke method to run several tasks in parallel.

// Using the Parallel.Invoke Method  
Parallel.Invoke( () => MethodForFirstTask(),   
                           () => MethodForSecondTask(),  
                           () => MethodForThirdTask() );

**Running Loop Iterations in Parallel**

The Parallel class also provides methods that you can use to run for and foreach loop iterations in parallel. Clearly it will not always be appropriate to run loop iterations in parallel. For example, if you want to compare sequential values, you must run your loop iterations sequentially. However, if each loop iteration represents an independent operation, running loop iterations in parallel enables you to maximize your use of the available processing power.

To run for loop iterations in parallel, you can use the Parallel.For method. This method has several overloads to cater to many different scenarios. In its simplest form, the Parallel.For method takes three parameters:

* An Int32 parameter that represents the start index for the operation, inclusive.
* An Int32 parameter that represents the end index for the operation, exclusive.
* An Action<Int32> delegate that is executed once per iteration.

The following code example shows how to use a Parallel.For loop. In this example, each element of an array is set to the square root of the index value. This is a simple example of a loop in which the order of the iterations does not matter.

// Using a Parallel.For Loop  
int from = 0;  
int to = 500000;  
double[] array = new double[capacity];  
// This is a sequential implementation:  
for(int index = 0; index < 500000; index++)  
{  
   array[index] = Math.Sqrt(index);  
}  
// This is the equivalent parallel implementation:  
Parallel.For(from, to, index =>  
{  
   array[index] = Math.Sqrt(index);  
});

To run foreach loop iterations in parallel, you can use the Parallel.ForEach method. Like the Parallel.For method, the Parallel.ForEach method includes many different overloads. In its simplest form, the Parallel.ForEach method takes two parameters:

* An IEnumerable<TSource> collection that you want to iterate over.
* An Action<TSource> delegate that is executed once per iteration.

The following code example shows how to use a Parallel.ForEach loop. In this example, you iterate over a generic list of Coffee objects. For each item, you call a method named CheckAvailability that accepts a Coffee object as an argument.

// Using a Parallel.ForEach Loop  
var coffeeList = new List<Coffee>();  
// Populate the coffee list…  
// This is a sequential implementation:  
foreach(Coffee coffee in coffeeList)  
{  
   CheckAvailability(coffee);  
}  
// This is the equivalent parallel implementation:  
Parallel.ForEach(coffeeList, coffee => CheckAvailability(coffee));

**Using Parallel LINQ**

Parallel LINQ (PLINQ) is an implementation of Language-Integrated Query (LINQ) that supports parallel operations. In most cases, PLINQ syntax is identical to regular LINQ syntax. When you write a LINQ expression, you can “opt in” to PLINQ by calling the AsParallel extension method on your IEnumerable data source.

The following code example shows how to write a PLINQ query.

// Using PLINQ  
var coffeeList = new List<Coffee>();  
// Populate the coffee list…  
var strongCoffees =   
   from coffee in coffeeList.AsParallel()  
   where coffee.Strength > 3  
   select coffee;

If you want the results ordered, you can use AsOrdered() in the query as shown in this code example.

// Using PLINQ  
var coffeeList = new List<Coffee>();  
// Populate the coffee list…  
var strongCoffees =   
   from coffee in coffeeList.AsParallel().AsOrdered()  
   where coffee.Strength > 3  
   select coffee;

This is only an introduction to PLINQ.  For more information visit the [Parallel LINQ Reference](http://go.microsoft.com/fwlink/?LinkId=532536) page on MSDN.