Quick answers to common problems

.NET 4.5 Parallel Extensions Cookbook

80 recipes to create scalable, task-based parallel programs using .NET 4.5

Bryan Freeman

[PACKT] PUBLISHING

.NET 4.5 Parallel Extensions Cookbook

80 recipes to create scalable, task-based parallel programs using .NET 4.5

Bryan Freeman

PACKT PUBLISHING

BIRMINGHAM - MUMBAI

.NET 4.5 Parallel Extensions Cookbook

First published: July 2013

Production Reference: 1190713

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK

ISBN 978-1-84969-022-5

www.packtpub.com

Cover Image by Karl Moore (karl@karlmoore.co.uk)

Credits

Author
Bryan Freeman

Reviewers
Carlos Hulot
Nauzad Kapadia
Jason DE Oliveira
Steven M. Swafford
Ken Tucker
Ariel Woscoboinik

Acquisition Editors
Aarthi Kumaraswamy
Kunal Parikh

Lead Technical Editor
Arun Nadar

Technical Editors
Athira Laji
Mrunmayee Patil
Priya Singh

Project Coordinator
Apeksha Chitnis

Proofreader
Elinor Perry-Smith

Indexer
Rekha Nair

Production Coordinator
Manu Joseph

Cover Work
Manu Joseph

About the Author

Bryan Freeman is a developer, architect, and writer on Parallel and Distributed Technologies. With over 15 years of experience delivering solutions across industry sectors such as healthcare, finance, and transportation, he specializes in technologies such as Microsoft .NET, Windows Communication Foundation, Windows Workflow, and Windows Azure to help clients deliver business value and drive revenue while reducing operational costs.

As an active speaker, writer, and blogger, Bryan is a passionate community advocate for those who develop solutions based on .NET technologies. He is also an active member of the Scottsdale Institute and the Veterans of Foreign Wars.

I would like to thank my beautiful wife Laura, and my daughter Isabelle for being there for me while I spent my evenings writing. Laura and Isabelle have endured the demands of my career with unwavering support.

About the Reviewers

Carlos Hulot has been working in the IT area for more than 20 years in different capabilities, from Software Development, Project Management, to IT Marketing Product Development and Management. Carlos has worked for multinational companies such as Royal Philips Electronics, PriceWaterhouseCoopers, and Microsoft. Currently he is working as an independent IT Consultant. Carlos is a Computer Science lecturer at two Brazilian universities. He holds a Ph.D in Computer Science and Electronics from the University of Southampton, UK and a B.Sc. in Physics from University of São Paulo, Brazil.

Nauzad Kapadia is an independent professional and founder of Quartz Systems, and provides training and consulting services for the entire Microsoft .NET and SQL Server stack. Nauzad has over 17 years of industry experience and has been a regular speaker at events like TechED, DevCon, DevDays, and user group events. He has been a Microsoft MVP (Most Valuable Professional) for 6 years on technologies ranging from C#, ASP.NET, to SQL Server. Whenever he is not working on his computer, he enjoys rock music, photography, and reading.

Jason De Oliveira works as CTO for Cellenza (`http://www.cellenza.com`), an IT Consulting Company specialized in Microsoft Technologies and Agile Methodology in Paris (France). He is an experienced Manager and Senior Solutions Architect, with high-level skills in Software Architecture and Enterprise Architecture.

Jason works for big companies and helps them to realize complex and challenging software projects. He frequently collaborates with Microsoft and you can find him quite often at the Microsoft Technology Center (MTC) in Paris.

He loves sharing his knowledge and experience via his blog, by speaking at conferences, writing technical books, writing articles in the technical press, giving software courses as MCT, and by coaching co-workers in his company.

In 2011, Microsoft gave him the Microsoft® Most Valuable Professional (MVP C#) Award for his numerous contributions to the Microsoft community. Microsoft seeks to recognize the best and brightest from technology communities around the world with the MVP Award. These exceptional and highly respected individuals come from more than 90 countries, serving their local online and offline communities, and having an impact worldwide. Jason is very proud to be one of them.

Please feel free to contact him via his blog if you need any technical assistance or want to exchange on technical subjects (`http://www.jasondeoliveira.com`).

Jason has worked on the following books:

- .Net Framework 4.5 Expert Programming Cookbook (English)
- WCF 4.5 Multi-Layer Services Development with Entity Framework (English)
- Visual Studio 2012 - Développez pour le web (French)

I would like to thank my lovely wife Orianne and my beautiful daughter Julia for supporting me in my work and for accepting long days and short nights during the week and sometimes even during the weekend. My life would not be the same without them!

Steven M. Swafford is a highly motivated Information Technology Professional with 18 years of experience who is result-oriented, and is a strong team player, high-energy, hands-on professional, self-motivated, and detail-oriented. Able to meet challenging deadlines both individually and as part of or leader of a team. Quickly adapts at overcoming obstacles and implementing organizational change.

Currently working as a Software Engineer, Steven develops and maintains web-based software solutions. As a skilled professional he is focused on the design and creation of software. Because communication skills are extremely important, Steven continues to expand his knowledge in order to communicate clearly with all facets of business. Recently Steven has been leading efforts to standardize software development tools and technology, plans and coordinates web accessibility as applied to Information Technology (IT) Solutions, and he is tackling application security in terms of best practice and implementation of the Security Development Lifecycle (SDL). Steven holds a Bachelor's Degree in Computer Science and a Master's Degree in Cyber security. Steven also holds both the CompTIA Network+ and Security+ certifications.

As of January 2013, Steven was promoted to the position of IT Principal Accessibility Manager. This job requires Steven to draw upon his experience to ensure Information, Communications, and Technology (ICT) is accessible under Section 508 and the Web Content Accessibility Guidelines (WCAG).

Ken Tucker has been given the Microsoft Most Valuable Professional (MVP) from 2003 till present and is working at SeaWorld Parks & Entertainment.

I would like to thank my wife Alice-Marie for her support.

Ariel Woscoboinik graduated as a Bachelor of Information Technology from the University of Buenos Aires and as an IT Technician from ORT schools. Since his childhood, he has been programing and getting more and more involved in the world of technology. Later on, he became interested in organizations and their business models and succeeded in converging both interests into his career: looking for the best solutions to involve people, processes, and technology.

Currently, he works as a Software Development Manager for Telefe, the leading TV channel in Argentina.

Ariel has been working with Microsoft Technologies since high school. During his career, he has worked for highly prestigious companies from Myriad industries: Microsoft, MAE, Intermex LLC, Pfizer, Monsanto, Banco Santander, IHSA, Disco S.A., Grupo Ecosistemas, Perception Group, Conuar.

Among his passions are drama, (as he is an amateur actor), film-watching, soccer, and travel around the world.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLiB™

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

The multicore processor boom is fully underway. It is becoming difficult to find a server, desktop, or laptop that doesn't have at least two processor cores. There are even several models of dual core mobile phones on the market.

However, to fully realize the power of modern multicore processors, developers need to be able to effectively recognize opportunities for parallelization and write parallel code.

Largely due to the complexities of creating and managing the lifecycle of classic threads, parallel coding is not routinely practiced by the large majority of developers.

To help ease the complexities of writing parallel code, Microsoft has introduced the Parallel Extensions in .NET framework. The Parallel Extensions or Task Parallel Library helps us to write efficient, scalable, parallel code in a natural manner without having to deal directly with the classic threading model or the thread pool. Using these extensions, we can conveniently write parallel versions of existing sequential code that can automatically utilize all available processors, thus providing significant performance gains.

This book provides coverage of the major classes of the .Net 4.5 Parallel extensions from front to back, and will provide a developer with no multithreaded development experience ability to write highly scalable asynchronous applications that take advantage of today's hardware.

What this book covers

Chapter 1, *Getting Started with Task Parallel Library*, looks at the various ways to create and start a task, cancel a task, handle exceptions in a task, and control the behavior of a task. At the core of the .NET Parallel Extensions is the System.Threading.Task class.

Chapter 2, *Implementing Continuations*, helps us learn how to create task continuations and child tasks. We will also learn how to control the condition under which continuations run, how to handle exceptions using continuations, and how to chain tasks and continuations together to form a pipeline pattern. In asynchronous programming, it is very common for one asynchronous operation to invoke a second operation and pass data to it.

Chapter 3, *Learning Concurrency with Parallel Loops*, helps us learn how to use parallel loops to process large collections of data in parallel. We will also learn how to control the degree of parallelism in loops, break out of loops, and partition source data for loop processing.

Chapter 4, *Parallel LINQ*, helps us learn how to create a parallel LINQ query, control query options such as results merging, and the degree of parallelism for the query. Parallel LINQ is a parallel implementation of LINQ to Objects.

Chapter 5, *Concurrent Collections*, helps us learn how to use concurrent collections as a data buffer in a producer-consumer pattern, how to add blocking and bounding to a custom collection, and how to use multiple concurrent collections for forming a pipeline. Concurrent collections in .NET Framework 4.5 allows developers to create type-safe as well as thread-safe collections.

Chapter 6, *Synchronization Primitives*, helps us look at the synchronization primitives available in the .NET Framework, and how to use the new lightweight synchronization primitives. Parallel applications usually need to manage access to some kind of shared data.

Chapter 7, *Profiling and Debugging*, helps us examine the parallel debugging and profiling tools build into Visual Studio 2012, and how to use them to debug and optimize our parallel applications.

Chapter 8, *Async*, helps us learn about the Async feature of the .NET Framework, and using asynchrony to maintain a responsive UI. We've all seen client applications that don't respond to mouse events or update the display for noticeable periods of time due to synchronous code holding on to the single UI thread for too long.

Chapter 9, *Dataflow Library*, helps us examine the various types of dataflow blocks provided by Microsoft, and how to use them to form a data processing pipeline. The Task Parallel Library's dataflow library is a new set of classes that are designed to increase the robustness of highly concurrent applications by using asynchronous message passing and pipelining to obtain more control and better performance than manual threading.

What you need for this book

This book assumes you have a solid foundation in general C# development. You should have at least a working knowledge of lambda expressions and delegates. You will need a Windows 7, Windows 8, or Windows Server 2008 R2 PC. A multicore or multiprocessor machine is not required, but is preferable. In addition, you will need Visual Studio 2012 and the .NET Framework 4.5.

Who this book is for

This book is intended to help experienced C# developers write applications that leverage the power of modern multicore processors. It provides the necessary knowledge for an experienced C# developer to work with .NET Parallel Extensions. Previous experience of writing multithreaded applications is not necessary.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "In this recipe, we will take a look at the basics of adding items to, and removing items from `BlockingCollection`."

A block of code is set as follows:

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/',
  '\u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
  Windows NT 6.1; Trident/6.0)";
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[var result = Enumerable.Range(0, 10000).AsParallel()
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .WithDegreeOfParallelism(2)
        …
        .Select((x, i) => i)
        .ToArray();
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Task Parallel Library

In this chapter, we will cover the following recipes:

- Creating a task
- Waiting for tasks to finish
- Returning results from a task
- Passing data to a task
- Creating a child task
- Lazy task execution
- Handling task exceptions using try/catch block
- Handling task exceptions with AggregateException.Handle
- Cancelling a task
- Cancelling one of many tasks

Introduction

At the beginning of the personal computer era, there was no concept of multiple threads offered by an operating system. Typically, operating system code and application code ran on a single thread of execution. The problem with this was that if a single application misbehaved, or simply took a long time to execute, the whole machine would stall, and often had to be rebooted.

As the development of the Windows operating systems progressed, Microsoft realized that they needed to improve this situation. In the Windows NT kernel, each application runs in its own process. A process is a collection of resources in which a virtual address space is allocated for each application. The advent of these processes ensured that code and data being used by one application could not be accessed and corrupted by another application, thus improving the reliability of the system.

Each process in Windows was also given its own thread. A thread is an operating system construct that functions like a virtual CPU. At any given moment, one of these threads is allowed to run on the physical CPU for a slice of time. When the time for a thread to run expires, it is swapped off of the CPU for another thread. Therefore, if a single thread enters an infinite loop, it can't monopolize all of the CPU time on the system. At the end of its time slice, it will be switched out for another thread.

Over the years, computers with multiple processors began to appear. These multiple processor machines were able to execute multiple threads at once. It became possible for an application to spawn new threads to run a compute-bound process asynchronously, thus gaining a performance improvement.

Over the past few years, the trend in processor development has shifted from making processors faster and faster, to making processors with multiple CPU cores on a single physical processor chip. Individuals who purchase these new machines expect their investment to pay off in terms of applications which are able to run efficiently across the available processor cores. Maximizing the utilization of the computing resources provided by the next generation of multi-core processors requires a change in the way the code is written.

The .NET framework has supported writing multi-threaded applications from the beginning, but the complexity of doing so has remained just out of reach for many .NET developers. To fully take the advantage of multi-threading, you needed to know quite a bit about how Windows works under the hood. For starters, you had to create and manage your own threads, which can be a demanding task as the number of threads in an application grows, and can often be the source of hard-to-find bugs.

Finally, help has arrived. Starting in .NET 4.0, Microsoft introduced the .NET Parallel Extensions, which gave us a new runtime, new class library types (the **Task Parallel Library** (**TPL**)), and new diagnostic tools to help with the inherent complexities of parallel programming.

The TPL isn't just a collection of new types. It's a completely new way of thinking about parallel programming. No longer do we need to think in terms of threads. With the TPL, we can now think in terms of `task`. With this new task-based model, we just need to figure out the pieces of our application that can execute concurrently, and convert those pieces into tasks. The runtime will take care of managing and creating all of the underlying threads that actually do the work. The `System.Threading.Task` class in itself is just a wrapper for passing a delegate, which is a data structure that refers to a `static` method or to a class instance, and an instance method of that class.

A TPL `task` still uses the classic thread pool internally, but the heavy lifting of spinning up new threads to carry out the `tasks` and determining the optimum number of threads required to take full advantage of the hardware, is all done by the runtime.

In this chapter, we will take a look at the basics of creating a parallel `task`. You will learn how to pass data into a `Task` using the `Task` state object, returning data from a `Task`, cancelling the `Task`, and handling exceptions within a `Task`.

Creating a task

`Tasks` are an abstraction in the .NET framework to represent asynchronous units of work. In some ways, a task resembles the creation of a classic .NET thread, but provides a higher level of abstraction, which makes your code easier to write and read.

We will look at the three basic ways to create and run a new task.

- The `Parallel.Invoke()` method: This method provides an easy way to run any number of concurrent statements
- The `Task.Start()` method: This method starts a task and schedules it for execution with `TaskScheduler`
- The `Task.Factory.StartNew()` method: This method creates and starts a task using `Task.Factory`

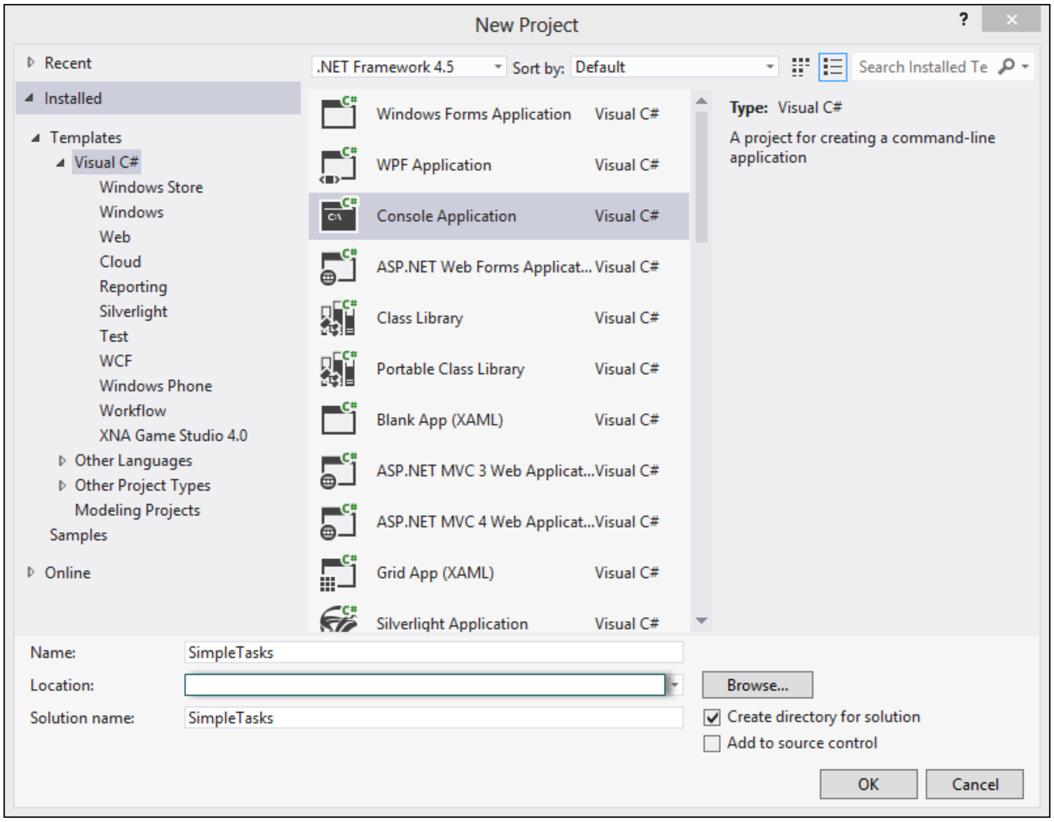
In this recipe, we will create a new task using each of these three methods. To give our tasks something to do, we will be using `WebClient` to read the text of three classic books. We will then split the words of each book into a string array, and display a count of the words in each book.

How to do it...

Ok, let's start building a `Console` application that demonstrates the various ways to create a parallel task.

1. Launch Visual Studio 2012.

2. Start a new project using the C# **Console Application** project template, and assign `SimpleTasks` as the **Solution name** as shown in the following screenshot:



3. Add the following `using` statements at the top of your `Program` class:

```
using System;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
```

> **Downloading the example code**
>
> You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

4. First, let's create a task using `Parallel.Invoke`. Add the following code to the `Main` method of the `Program` class:

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";

Parallel.Invoke(() =>
    {
    Console.WriteLine("Starting first task using Parallel.
Invoke");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words =client.DownloadString(@"http://www.gutenberg.org/
files/2009/2009.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Origin of Species word count: {0}",
wordArray.Count());
    client.Dispose();
    }
);
```

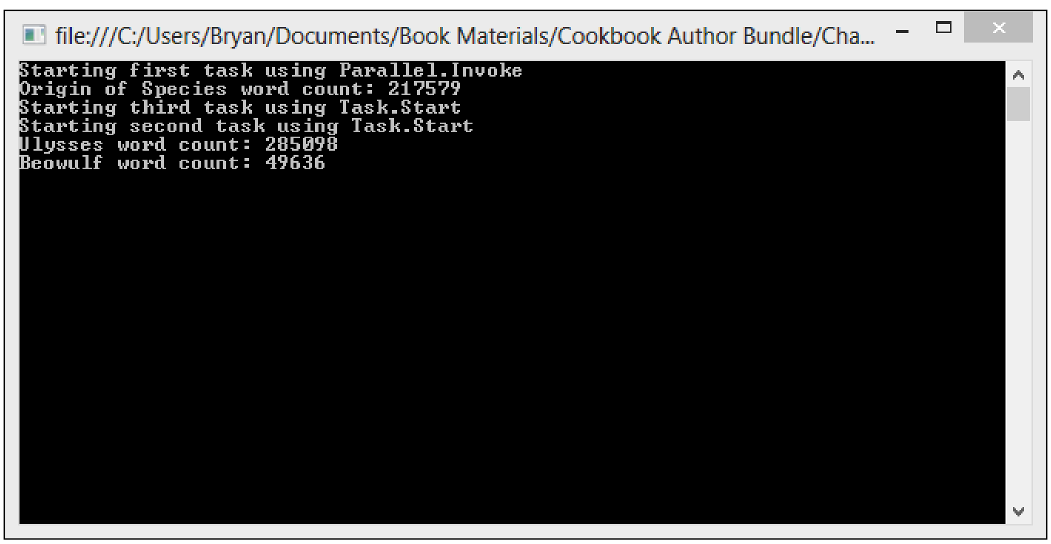
5. Next, let's start `task` using the `Start` method of the `Task` object. Add the following code to the `Main` method of the `Program` class just below the code for the previous step:

```
var secondTask = new Task(() =>
    {
    Console.WriteLine("Starting second task using Task.Start");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/16328/16328-8.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Beowulf word count: {0}", wordArray.
Count());
    client.Dispose();
    }
  );
secondTask.Start();
```

6. Finally, let's create `task` using `Task.Factory.StartNew`. Add the following code to the `Main` method of the `Program` class:

```
Task.Factory.StartNew(() =>
    {
    Console.WriteLine("Starting third task using Task.Factory.
StartNew");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/4300/4300.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Ulysses word count: {0}", wordArray.
Count());
    client.Dispose();
    }
);
//wait for Enter key to exit
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot. Note that the exact order of the text you see may vary as `tasks` run asynchronously:



How it works...

The `Parallel.Invoke` method can implicitly create and run any number of statements concurrently by passing an action delegate for each delegate of work to be done.

```
Parallel.Invoke(( )=>DoSomething( ), ( )=>DoSomethingElse( ));
```

It is worth noting however, that the number of `tasks` actually created by `Parallel.Invoke` may or may not be equal to the number of delegates passed in, especially if there are a large number of delegates.

Using `Task.Start()` or `Task.Factory.StartNew()` creates new tasks explicitly. The new `tasks` will be allocated threads by the `ThreadPool` class, which handles the actual creation of the threads the `tasks` use for carrying out their work. As developers, we are shielded from all of this thread creation work, because it is done for us by the `Task` object.

When you create a task, you are really just creating a wrapper around a delegate of work to be performed. The delegate can be a named delegate and anonymous method, or a lambda expression.

So, which of these methods of creating `task` is the best? `Task.Factory.StartNew` is usually the preferred method, because it is more efficient in terms of the synchronization costs. Some amount of synchronization cost is incurred when using `Thread.Start`, because it is necessary to ensure that another thread is not simultaneously calling start on the same `Task` object. When using `Task.Factory.StartNew`, we know that the task has already been scheduled by the time `task` reference is handed back to our code.

Note also that you can't call `Start()` on a task that has already run and completed. If you need the tasks to do the work again, you need to create new `task` with the same delegate of work.

For the remainder of this book, we will primarily be using `Task.Factory.StartNew`.

Waiting for tasks to finish

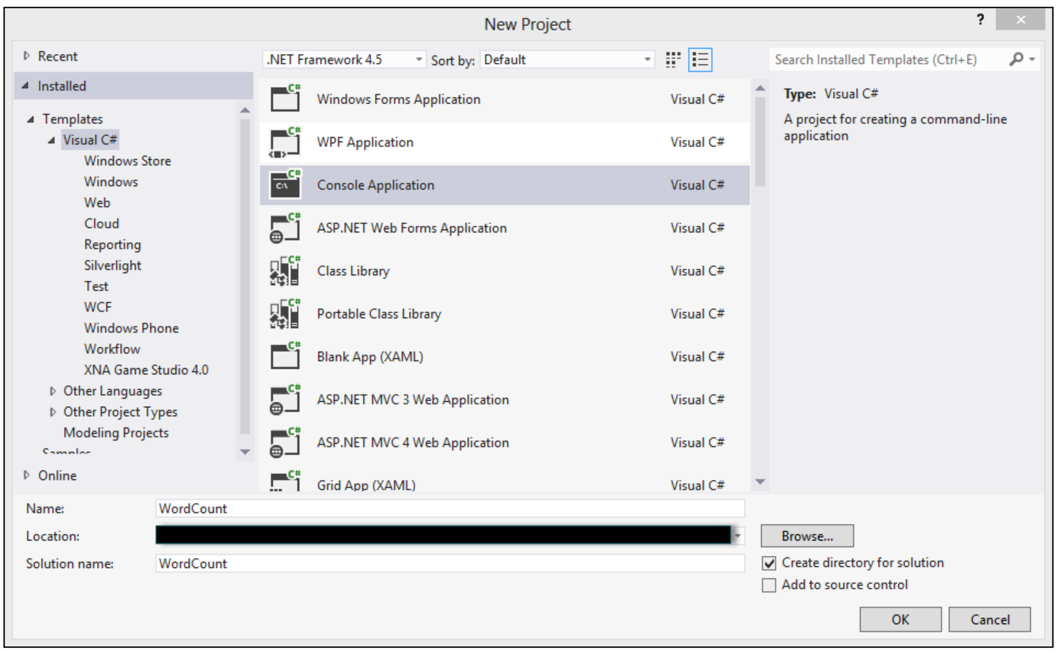
When developing a parallel application, you will often have situations where a task must be completed before the main thread can continue processing. The Task Parallel Library includes several methods that allow you to wait for one or more parallel `tasks` to complete. This recipe will cover two such methods: `Task.Wait()` and `Task.WaitAll()`.

In this recipe we will be creating three tasks, all of which read in the text classic books and produce a `word count`. After we create the first task, we will wait for it to complete using `Task.Wait()`, before starting the second and third task. We will then wait for both the second and third tasks to complete using `Task.WaitAll()` before writing a message to the console.

How to do it...

Let's create a `Console` application that demonstrates how to wait for `task` completion.

1. Launch Visual Studio 2012.
2. Start a new project using the C# **Console Application** project template, and assign `WordCount` as the **Solution name**.



3. Add the following `using` statements at the top of your `Program` class:

```
using System;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
```

4. In the `Main` method of the `Program` class, add a character array containing the basic punctuation marks. We will use this array in `string.Split()` to eliminate punctuation marks. Also add a string `constant` for the `user-agent` header of the `WebClient`.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
```

5. OK, now let's create our first task. This task will use `WebClient` to read the *Origin of Species* by Darwin, and get its word count. Enter the following code in the `Main` method of the `Program` class just below the previous statement:

```
var task1 = Task.Factory.StartNew(() =>
    {
    Console.WriteLine("Starting first task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009/2009.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Origin of Species word count: {0}",
wordArray.Count());
    }
);
```

6. Now, just below the previous task, write the following statements to wait on the task, and write a message to the `Console` application:

```
task1.Wait();
Console.WriteLine("Task 1 complete. Creating Task 2 and Task 3.");
```

7. Below the previous statement, enter the code to create the second and third tasks. These tasks are very similar to the first task.

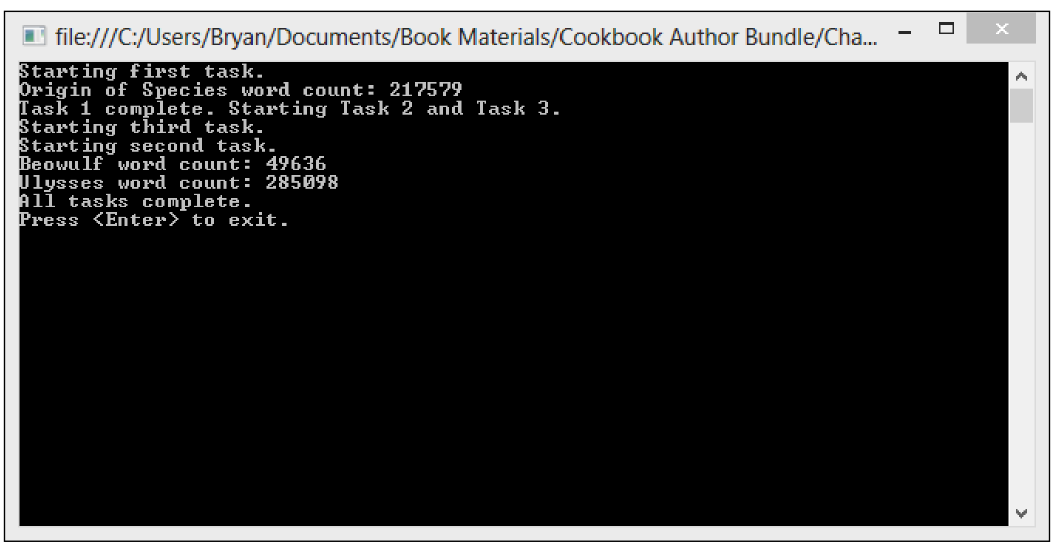
```
var task2 = Task.Factory.StartNew(() =>
{
  Console.WriteLine("Starting second task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/16328/16328-8.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
```

```
        Console.WriteLine("Beowulf word count: {0}", wordArray.
Count());
 });

var task3 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Starting third task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/4300/4300.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Ulysses word count: {0}", wordArray.
Count());
});
```

8. Finally, let's use `Task.WaitAll()` to wait for the second and third task to complete, then prompt the user to exit the program. `Task.WaitAll()` takes an array of `task` as its parameter, and can be used to wait for any number of tasks to complete.

```
Task.WaitAll(task2,task3);
Console.WriteLine("All tasks complete.");
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot. Note that the exact order of the last few lines of text may still vary depending on the execution order of the second and third tasks.



How it works...

Although `Task.Wait()` and `Task.WaitAll()` are fairly self-explanatory, both have several overloads that offer different functionalities.

`Task.Wait()` can take either an `Int32` or `TimeSpan` parameter to specify a specific period of time to wait. It can also accept a `CancellationToken` token parameter for cancellation, which will be covered later in the chapter.

`Task.WaitAll()` always takes an array of `Task` as its first parameter, and has a second parameter which can be an `Int32` or `TimeSpan` as in `Task.Wait`.

Another useful method not shown in the recipe is `Task.WaitAny()`. `WaitAny` is very similar to `WaitAll`, except that it waits for only one `Task` in the array of `Task` to complete. The first `Task` of `Task` array to finish, completes the wait condition, and execution of the main thread is allowed to move forward.

It is important to note that when you call one of the `Wait` methods, the runtime will check to see if the task you are waiting on has started executing. If `task` has started executing, then the thread that called `Wait` will block until `task` has finished executing. However, if `task` has not started running, then the runtime may execute the task using the thread that calls `Wait`.

The various overloads and behaviors of `Task.Wait`, `Task.WaitAll`, and `Task.WaitAny` are shown in the following table:

`Wait()`	Waits for the task to complete execution.
`Wait(CancellationToken)`	Waits for the task to complete execution or `CancellationToken` to be set.
`Wait(Int32)`	Waits for task to complete or number of milliseconds to pass. A value of `-1` waits indefinitely.
`Wait(TimeSpan)`	Waits for the task to complete execution or specified timespan to pass.
`Wait(Int32, CancellationToken)`	Waits for task to complete, number of milliseconds to pass, or `CancellationToken` to be set.
`WaitAll(Task[])`	Waits for all of the tasks in array to complete execution.
`WaitAll(Task[], Int32)`	Waits for all of the tasks in the array to complete execution or number of milliseconds to pass. A value of `-1` waits indefinitely.

`WaitAll(Task[], CancellationToken)`	Waits for all of the tasks in array to complete execution or for a `CancellationToken` to be set.
`WaitAll(Task[], TimeSpan)`	Waits for all of the tasks in array to complete execution or specified timespan to pass.
`WaitAll(Task[], Int32, CancellationToken)`	Waits for all of the tasks in array to complete execution, number of milliseconds to pass, or `CancellationToken` to be set.
`WaitAny(Task[])`	Waits for any of the tasks in the array to complete execution.
`WaitAny(Task[], Int32)`	Waits for any of the tasks in array to complete execution or number of milliseconds to pass. A value of `-1` waits indefinitely.
`WaitAny(Task[], CancellationToken)`	Waits for any of the tasks in array to complete execution or for a `CancellationToken` to be set.
`WaitAny(Task[], TimeSpan)`	Waits for any of the tasks in array to complete execution or specified timespan to pass.
`WaitAny(Task[], Int32, CancellationToken)`	Waits for any of the tasks in array to complete execution, number of milliseconds to pass, or `CancellationToken` to be set.

Returning results from a task

So far, our tasks have not returned any values. However, it is often necessary to return a result from a task so it can be used in another part of our application. This functionality is provided by the `Result` property of `Task<TResult>`.

In this recipe, we will be creating a solution similar with tasks similar to the previous solution, but each of our three tasks return a result which can then be used to display the word count to the user.

How to do it...

Let's go to Visual Studio and see how we can return result values from our tasks.

1. Start a new project using the C# **Console Application** project template, and assign `WordCount2` as the **Solution name**.
2. Add the following `using` statements are at the top of your `Program` class:

```
using System;
using System.Linq;
```

```
using System.Net;
using System.Threading.Tasks;
```

3. In the `Main` method of the `Program` class, add a character array containing the basic punctuation marks. We will use this array in `string.Split()` to eliminate punctuation marks. Also add a string constant for the `WebClient` `user-agent` header.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
```

4. Start by creating three tasks of type `Task<int>` named `task1`, `task2`, and `task3`. Your tasks should look as shown in the following code snippet:

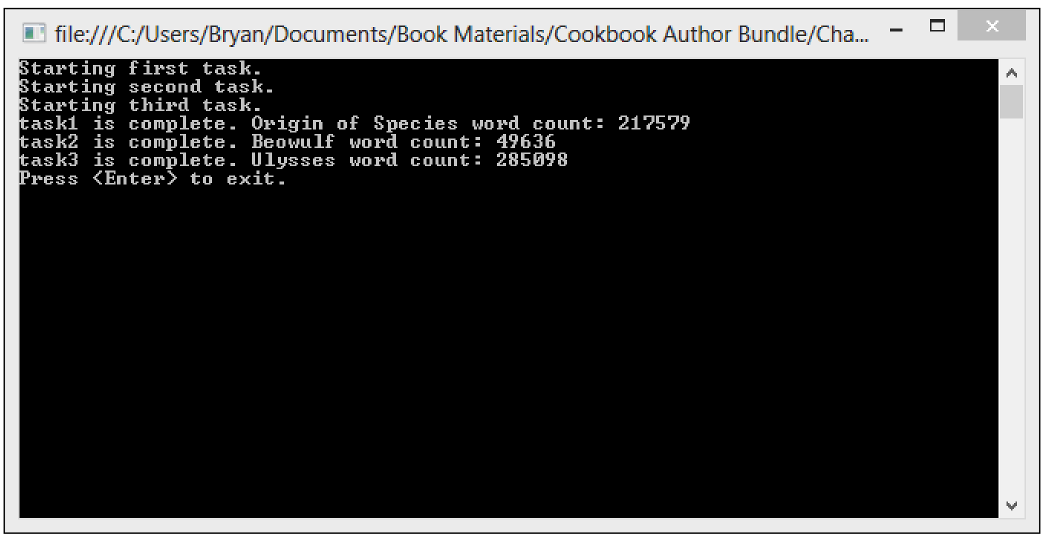
```
Task<int> task1 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Starting first task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009/2009.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
});

Task<int> task2 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Starting second task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/16328/16328-8.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
});

 Task<int> task3 = Task.Factory.StartNew(()
 {
    Console.WriteLine("Starting third task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
```

```
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/4300/4300.txt");
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
});
```

5. Immediately below your tasks, add Console.Writeline() statements that use Task.Result to display the results to the user. The remainder of the Main method should now look as shown in the following code snippet:

```
Console.WriteLine("task1 is complete. Origin of Species word
count: {0}",task1.Result);
Console.WriteLine("task2 is complete. Beowulf word count: {0}",
task2.Result);
Console.WriteLine("task3 is complete. Ulysses word count: {0}",
task3.Result);
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following:



How it works...

Task<TResult> subclasses the standard Task class and provides the additional feature of the ability to return a value. This is done by switching from providing an Action delegate to providing a Func<TResult> delegate.

It is worth noting that calling the `Task.Result` accessor will ensure that the asynchronous operation is complete before returning, so this is another method of waiting for a task to complete. Once the result of `Task` is available, it will be stored and returned immediately on later calls to the `Result` accessor.

Passing data to a task

You can supply the data used by `task` by passing an instance of `System.Action<object>` and an object representing the data to be used by the action.

In this recipe, we will be revisiting our WordCount example, but this time we will be parameterizing the data the tasks will act upon.

How to do it...

The ability to pass data into a task allows us to create a single task that can operate on multiple pieces of input data. Let's create a `Console` application so we can see how this works:

1. Start a new project using the C# **Console Application** project template and assign `WordCount3` as the **Solution name**.
2. Add the following `using` statements at the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   using System.Collections.Generic;
   ```

3. In the `Main` method of the `Program` class, add a character array containing the basic punctuation marks. We will use this array in `string.Split()` to eliminate punctuation marks. Also add a constant string for the `WebClients` user-agent task.

   ```
   char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
   u000A' };
   const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
   Windows NT 6.1; Trident/6.0)";
   ```

4. For this recipe, let's create a `new Dictionary` instance that can hold our book titles and URLs. Immediately after the previous statement, add the following code to create and initialize the `dictionary`:

   ```
   var dictionary = new Dictionary<string, string>
   {
       {"Origin of Species", "http://www.gutenberg.org/
   files/2009/2009.txt"},
   ```

```
    {"Beowulf", "http://www.gutenberg.org/files/16328/16328-8.txt"},
    {"Ulysses", "http://www.gutenberg.org/files/4300/4300.txt"}
};
```

5. This time we will be creating anonymous tasks in a loop. We still would like to wait for the tasks to complete before prompting the user to exit the program. We need a collection to hold our tasks, so we can pass them to `Task.WaitAll()` and wait for completion. Below the previous statement, create a `List<Task>` to hold our tasks.

```
var tasks = new List<Task>();
```

6. Next, we want to create a `for` loop to loop through `KeyValuePairs` in the dictionary. Let's put the `for` loop below the previous statement.

```
foreach (var  pair in dictionary)
{
}
```

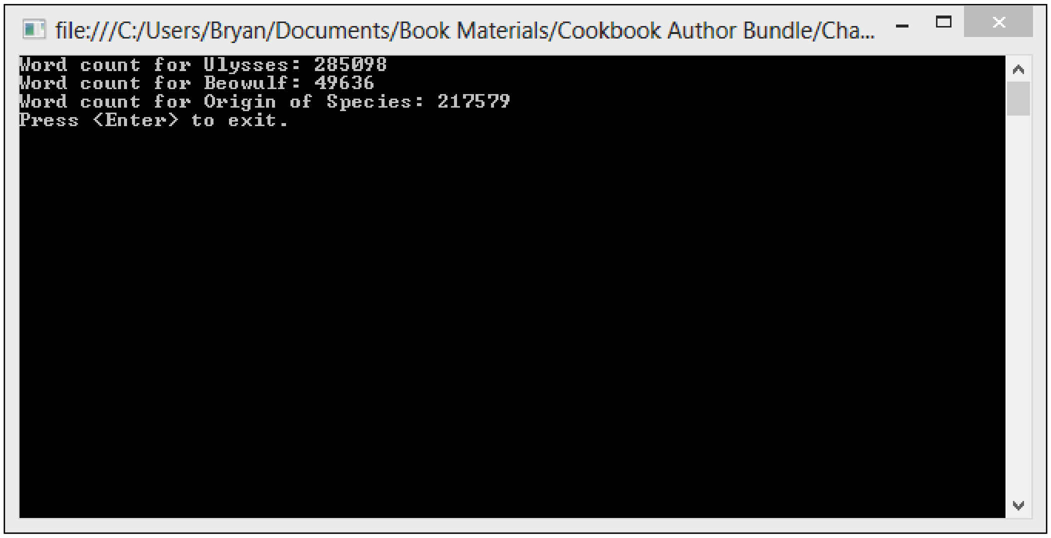
7. Inside the body of your `for` loop, put the definition of `task`, and add it to your task list as follows. Note the `KeyValuePair` being passed into `task` is in the form of an object. In the delegate body, we cast this object back to a `KeyValuePair`. Other than that, task is pretty much the same.

```
tasks.Add( Task.Factory.StartNew((stateObj) =>
{
    var taskData = (KeyValuePair<string, string>)stateObj;
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(taskData.Value);
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    Console.WriteLine("Word count for {0}: {1}", taskData.Key,
wordArray.Count());
},pair));
```

8. After the `for` loop, let's finish things up by waiting on the tasks to complete using `Task.WaitAll()` and prompting the user to exit. The last few lines should be as follows:

```
Task.WaitAll(tasks.ToArray());
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

By passing data to `Task` using the state feature, we now have a very powerful model for task creation, because we can create many tasks at once, each having the same code statements in the body and passing in the data that `Task` operates on. It also makes our code much more concise and readable.

In our application we need to pass two items of data into the `task`: a book title and the URL of the book, so we created `dictionary`.

```
var dictionary = new Dictionary<string, string>
{
    {"Origin of Species", "http://www.gutenberg.org/files/2009/2009.
txt"},
    {"Beowulf", "http://www.gutenberg.org/files/16328/16328-8.txt"},
    {"Ulysses", "http://www.gutenberg.org/files/4300/4300.txt"}
};
```

We would also want to wait on all of these tasks to complete before we prompt the user to exit, so we need to create a collection that can be converted to an array of tasks to hold our `Task` objects. In this case, we made a list of tasks. In the body of our look that creates the tasks, we will add the tasks to the list.

```
var tasks = new List<Task>();

foreach (var pair in dictionary)
```

```
{
    tasks.Add( //TASK DECLARATION HERE   ));
}
```

In our loop, we will pass in each of `KeyValuePairs` in `dictionary` as an object, using the `Task(Action<Object>, Object)` constructor. This syntax is just a bit odd because you actually refer to the `state` object twice.

```
Task.Factory.StartNew((stateObj) =>
{
    // TASK Body
},pair ));}
```

The key takeaway here is that the only way to pass data to a `Task` constructor is using `Action<Object>`. To use the members of a specific type, you must convert or explicitly cast the data back to the desired type in the body of the `Task`.

```
var taskData = (KeyValuePair<string, string>)stateObj;
```

Creating a child task

Code that is running a task can create another task with the `TaskCreationOptions.AttachedToParent set`. In this case, the new task becomes a child of the original or parent task.

In this recipe, we will be using a simplified version of the WordCount solution that uses a parent task to get the text of one book into a string array, and then spins up a child task to print the results.

How to do it...

Let's return to our WordCount solution, so we can see how to create a child task and attach it to a parent.

1. Start a new project using the C# **Console Application** project template, and assign `WordCount4` as the **Solution name**.
2. Add the following `using` statements at the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   ```

3. In the `Main` method of the `Program` class, add a character array containing the basic punctuation marks. We will use this array in `string.Split()` to eliminate punctuation marks. Also, add a constant string for the `WebClient` `user-agent` header.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
```

4. First, let's create the basic structure of our parent task. This is very similar to the other tasks we have created so far, and takes no parameters, and returns no values.

```
Task parent = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Parent task starting");
    const string uri = "http://www.gutenberg.org/files/2009/2009.
txt";
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var book = client.DownloadString(uri);
    var wordArray = book.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    // Child Task will go here
});
```

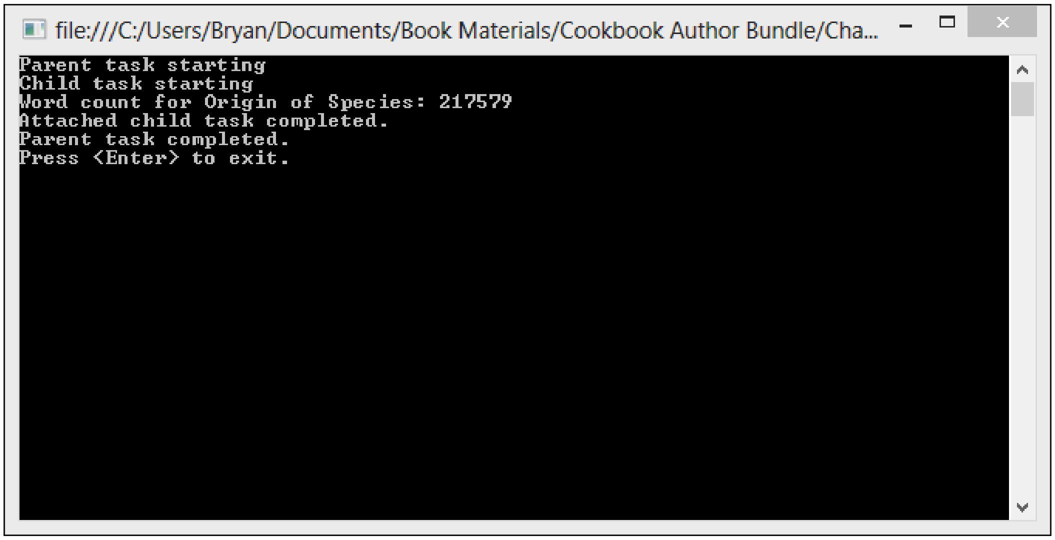
5. Next, right after the comment in the parent task, let's create a child task to print the results and set the `AttachedToParent` option.

```
Task.Factory.StartNew(()=>
{
    Console.WriteLine("Child task starting");
    Console.WriteLine("Word count for Origin of Species:
{0}",wordArray.Count());
    Console.WriteLine("Attached child task completed.");
},TaskCreationOptions.AttachedToParent);
```

6. Finally, just below the close of the parent task, let's wait for the parent task to complete, and prompt the user to exit the application with the following code:

```
parent.Wait();
Console.WriteLine("Parent task completed.");
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

7. That's pretty much it. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

Using `TaskCreationOptions.AttachedToParent` expresses structured parallelism. The parent task will wait for the child task to finish, so at the end of our program, all we have to do is wait for the parent task.

The nested child task, itself, is just an ordinary `task` created in the delegate of another `task`. A parent task may create any number of child tasks, limited only by system resources.

You can also create a nested task without using `TaskCreationOptions.AttachedToParent`. The only real difference is that the nested tasks created without this option are essentially independent from the outer task. A task created with the `TaskCreationOptions.AttachedToParent` option set is very closely synchronized with the parent.

The outer task could also use the `DenyChildAttach` option to prevent other tasks from attaching as child tasks. However, the same outer task could still create an independent nested task.

Lazy task execution

Lazy initialization of an object means that object creation is deferred until the object is actually used by your program. If you have a parallel task that you want to execute only when the value returned from the task is actually needed, you can combine lazy task execution with the `Task.Factory.StartNew` method.

In this recipe, we will return to our, by now familiar WordCount solution, to show you how to execute a parallel task and compute a word count for our book, only when we display the result to the console.

How to do it...

Let's create a console application that demonstrates how we can defer task creation until the result of the task is needed.

1. Start a new project using the C# **Console Application** project template, and assign `WordCount5` as the **Solution name**.
2. Add the following `using` statements at the top of your `Program` class.

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   ```

3. The first step is to declare `System.Threading.Task<int>` for lazy initialization. In the `Main` method of your `Program` class, put a `Lazy` declaration as follows:

   ```
   var lazyCount = new Lazy<Task<int>>(()=
   {
     //Task declaration and body go here
   });
   ```

4. Inside the `Lazy` initialization declaration, place the code to create to task. The entire statement should now look as the following code snippet:

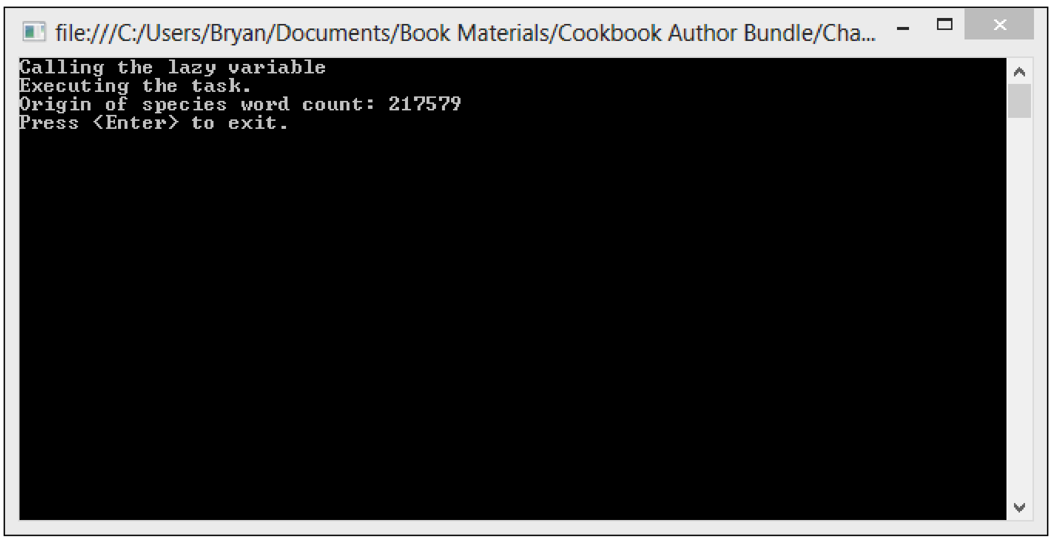
   ```
   Task<int>.Factory.StartNew(() =>
   {
       Console.WriteLine("Executing the task.");
       char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/',
   '\u000A' };
       const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
   Windows NT 6.1; Trident/6.0)";
       const string uri = "http://www.gutenberg.org/files/2009/2009.
   txt";
       var client = new WebClient();
       client.Headers.Add("user-agent", headerText);
   ```

```
        var words = client.DownloadString(uri);
        var wordArray = words.Split(delimiters, StringSplitOptions.
    RemoveEmptyEntries);
        return wordArray.Count();
    }));
```

5. Now we just need to write the result to the Console. Just add the following code to the end of your program:

```
Console.WriteLine("Calling the lazy variable");
Console.WriteLine("Origin of species word count: {0}",lazyCount.
Value.Result );
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

6. All done. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

System.Lazy<T> creates a thread safe Lazy initialization of an object. Lazy initialization is primarily used to improve performance and avoid computational overhead until necessary. You can pass a delegate (remember that System Threading Task is just a wrapper around a delegate) to the System.Lazy constructor, and as we have done in this recipe, you can use a lambda expression to specify a factory method for object creation. This keeps all of the initialization code in one place.

Lazy initialization occurs the first time the System.Lazy<T>.Value property is accessed.

Handling task exceptions using try/catch block

Let's face it; sometimes things just go wrong with our code. Even with the simplified parallel programming model provided by the TPL, we still need to be able to handle our exceptions.

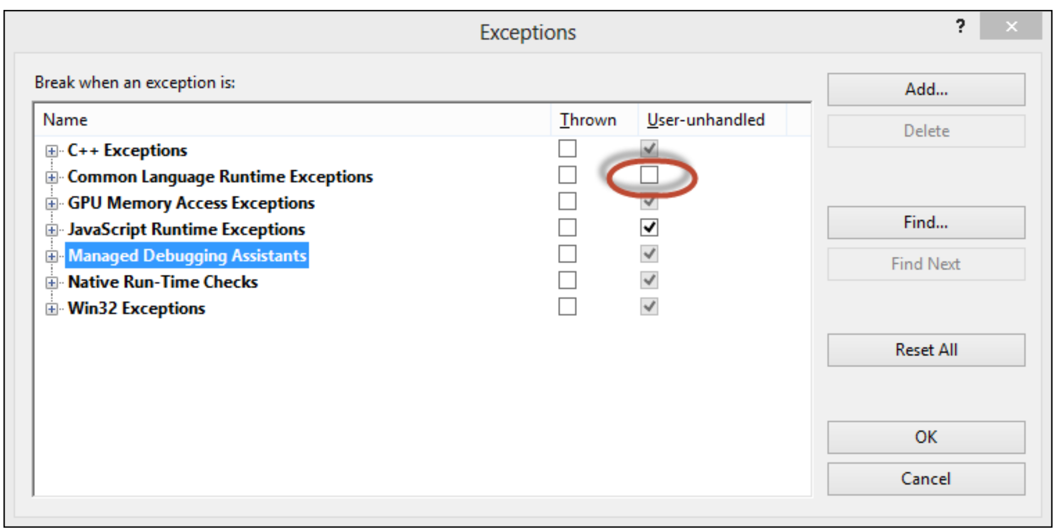
Tasks use `System.AggregateException` to consolidate multiple failures into a single exception object. In this recipe, we will take a look at the simplest way to handle `System.AggregateException` in our `tasks`: the `try/catch` blocks.

The try-catch statement consists of a try block followed by one of more catch blocks, which specify handlers for different exceptions. The try block contains the guarded code that may cause the exception.

Getting ready...

For this recipe we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime exception is thrown, and intercepts the exception before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **user-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Let's return to our WordCount solution so we can see how to handle an `AggregateException` thrown by a parallel task.

1. Start a new project using the C# **Console Application** project template and assign `WordCount6` as the **Solution name**.
2. Add the following `using` statements are at the top of your `Program` class:

```
using System;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
```

3. For this recipe, we will just need a single task. The task will be very similar to our other word count tasks, but in this one we will simulate a problem with the `System.Net.WebClient` by creating and throwing a `System.Net.WebException`. In the `Main` method of your `Program` class, create `System.Task` that looks as the following `Task`:

```
Task<int> task1 = Task.Factory.StartNew(() =>
{
    const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
    Console.WriteLine("Starting the task.");
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009/2009.txt");
    var ex = new WebException("Unable to download book contents");
    throw ex;
    return 0;
});
```

4. Just below the `Task`, let's put in our `try/catch` blocks as shown in the following code snippet. In the `catch` block, we will want to specifically catch `System.AggregateException`.

```
try
{
}
catch (AggregateException aggEx)
{
}
```

5. Now let's implement the body of our `try` block. The body of the `try` block should be as shown in the following code snippet. There are a couple of subtle but important concepts in here that will be explained later in the chapter.

```
try
{
    task1.Wait();
    if (!task1.IsFaulted)
    {
     Console.WriteLine("Task complete. Origin of Species word 
count: {0}",task1.Result);
    }
}
```

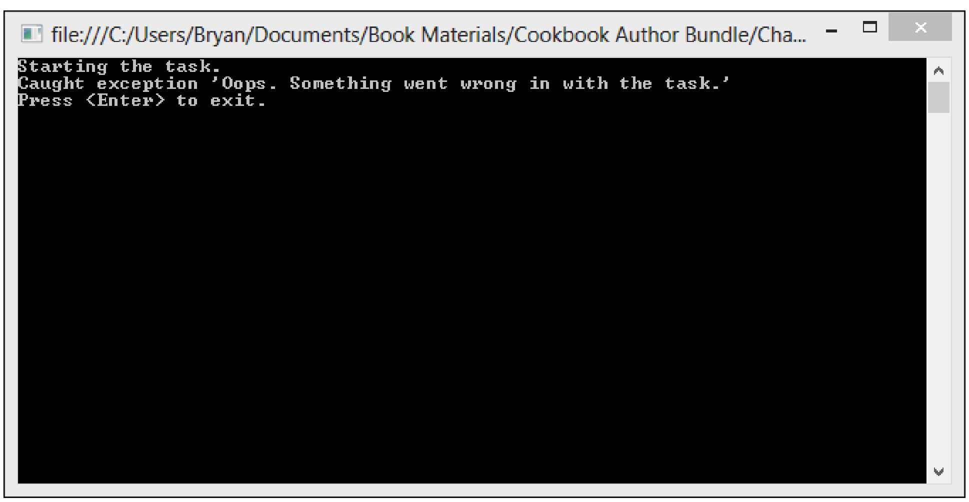
6. Next, let's implement the body of our `catch` block. It should look as shown in the following code snippet:

```
catch (AggregateException aggEx)
{
    foreach (var ex in aggEx.InnerExceptions)
    {
      Console.WriteLine("Caught exception: {0}", ex.Message);
    }
}
```

7. After the `catch` block, let's finish up by prompting the user to exit, and waiting on the user to hit *Enter*.

```
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

8. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

All of this stuff has been pretty self-explanatory so far, but handling exceptions in task involves a couple of subtleties that need to be pointed out.

The task itself is pretty straightforward. Other than throwing the `System.Net.WebException`, there is nothing out of the ordinary here.

Let's take a closer look at the try/catch blocks. The first statement in the `try` block `System.Threading.Task.Wait()` to wait on task completion. However, there is another purpose here. Unhandled exceptions thrown inside a `task` are swallowed by the runtime and wrapped up in `System.AggregateException`. It is your job to handle this.

The TPL also has the concept of `AggregateException` being observed. If `AggregateException` is raised by your task, it will only be handled if it is currently being observed. This is very important to understand. If you never take an action that causes the exceptions to be observed, you are going to have a problem. When the `Task` object is garbage collected, the `Finalize` method of the `task` will see that the `task` had unobserved exceptions, and it will throw`System.AggregateException`. You will not be able to catch an exception thrown by the finalizer thread and your process will be terminated.

So how to you observe an `AggregateException`, you ask? The `Systm.Threading.Task` class has a few methods and properties call triggers that cause `System.AggregateException` to be observed. A few of these are as follows:

- Task.Wait
- Task.WaitAny
- Task.WaitAll
- Task.Result

Using any of these `trigger` methods indicates to the runtime that you are interested in observing any `System.AggregateException` that occurs. If you do not use one of the `trigger` methods on the `Task` class, the TPL will not raise any `AggregateException`, and an unhandled exception will occur.

Now, let's take a look at the `catch` block. `System.AggregateException` can wrap many individual exception objects. In our `catch` block, we need to loop through `AggregateException.InnerExceptions` to take a look at all of the individual exceptions that occurred in a task.

It is important to note that there is really no way to correlate an exception from the `AggregateExcetion.InnerExceptions` collection back to the particular `task` that threw an exception. All you really know is that some operation threw an `Exception`.

`System.AggregateException` overrides the `GetBaseException` method of exception, and returns the innermost exception, which is the initial cause of the problem.

Handling task exceptions with AggregateException.Handle

In this recipe, we will look at another way to handle `System.AggregateException`, by using the `AggregateException.Handle` method. The `Handler` method invokes a handler function for each exception wrapped in `AggregateException`.

Getting ready...

For this recipe, we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime exception is thrown and intercepts the exception before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **user-unhandled** checkbox next to **Common Language Runtime Exceptions**.

How to do it...

Let's take a look at how we can use `AggregateException.Handle` to provide an alternate method to handling exceptions in a parallel application.

1. For this recipe, we will return to our **WordCount6** project, and modify it to handle our exceptions in a different way. Start Visual Studio 2012 and open the **WordCount6** project.
2. The first step is to define our handler function that will be invoked when we call `AggregateException.Handle`. Following the `Main` method of your `Program` class, add a new `private static handler` method that returns a bool. It should look as the following code snippet:

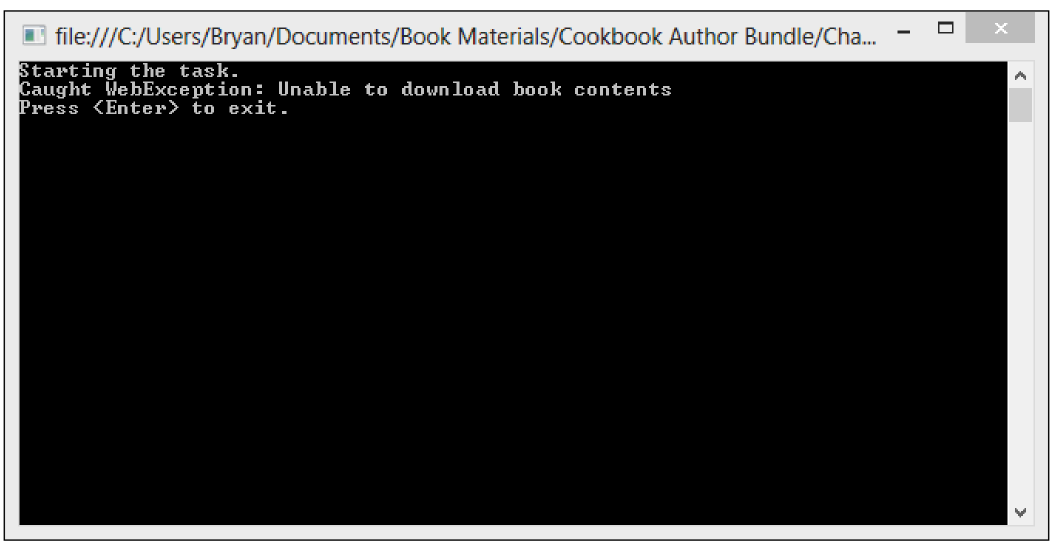
```
private static bool HandleWebExceptions(Exception ex)
{
    if (ex is WebException)
    {
      Console.WriteLine(("Caught WebException: {0}", ex.Message);
      return true;
    }
```

```
        else
        {
          Console.WriteLine("Caught exception: {0}", ex.Message);
          return false;
        }
    }
```

3. The only other step here is to replace the body of your `catch` block with a call to `System.AggregateException.Handle`, passing in the `HandleWebExceptions` predicate. The updated `try/catch` block should look as follows:

```
try
{
    task1.Wait();
    if (!task1.IsFaulted)
    {
      Console.WriteLine("Task complete. Origin of Species word
count: {0}",task1.Result);
    }
}
catch (AggregateException aggEx)
{
    aggEx.Handle(HandleWebExceptions);
}
```

4. Those are the only modifications necessary. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

`AggregateException.Handle()` takes a predicate that you supply, and the predicate will be invoked once for every exception wrapped in `System.AggregateException`.

The predicate itself just needs to contain the logic to handle the various exception types that you expect, and to return true or false to indicate whether the exception was handled.

If any of the exceptions went unhandled, they will be wrapped in a new `System.AggregateException` and thrown.

Cancelling a task

Up to this point, we have focused on creating, running, and handling exceptions in `tasks`. Now we will begin to take a look at using `System.Threading.CancellationTokenSource` and `System.Threading.CancellationToken` to cancel `tasks`.

This recipe will show how to cancel a single task.

How to do it...

Let's create a console application that shows how to cancel a parallel task.

1. Start a new project using the C# **Console Application** project template, and assign `WordCount7` as the **Solution name**.
2. Add the following `using` statements at the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. Let's start by creating `CancellationTokenSource` and getting our `CancellationToken`. In the `Main` method of your `Program` class, add the following statements:

   ```
   //Create a cancellation token source
   CancellationTokenSource tokenSource = new
   CancellationTokenSource();
   //get the cancellation token
   CancellationToken token = tokenSource.Token;
   ```

4. Now we need to create our `Task` and pass `CancellationToken` into the constructor. Right after the previous line, put in the following `Task` definition:

```
Task<int> task1 = Task.Factory.StartNew(() =>
{
    // The body of the task goes here
}, token);
```

5. In the body of our `Task`, we need to check the `IsCancellationRequested` property of `CancellationToken`. If it has been, we dispose of our resource and throw `OperationCancelledException`. If not, we do our usual work. Enter the following code into the body of `Task`:

```
//wait a bit for the cancellation
Thread.Sleep(2000);
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
var client = new WebClient();
client.Headers.Add("user-agent", headerText);
if(token.IsCancellationRequested)
{
    client.Dispose();
    throw new OperationCanceledException(token);
}
else
{
    var book = client.DownloadString(@"http://www.gutenberg.org/
files/2009/2009.txt");
    char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/',
'\u000A' };
    Console.WriteLine("Starting the task.");
    var wordArray = book.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
}
```

6. Right after the task, put in the following lines to write the cancellation status to the `Console`, and then call the `Cancel` method of `TokenSource`.

```
Console.WriteLine("Has the task been cancelled?: {0}", task1.
IsCanceled);
//Cancel the token source
tokenSource.Cancel();
```

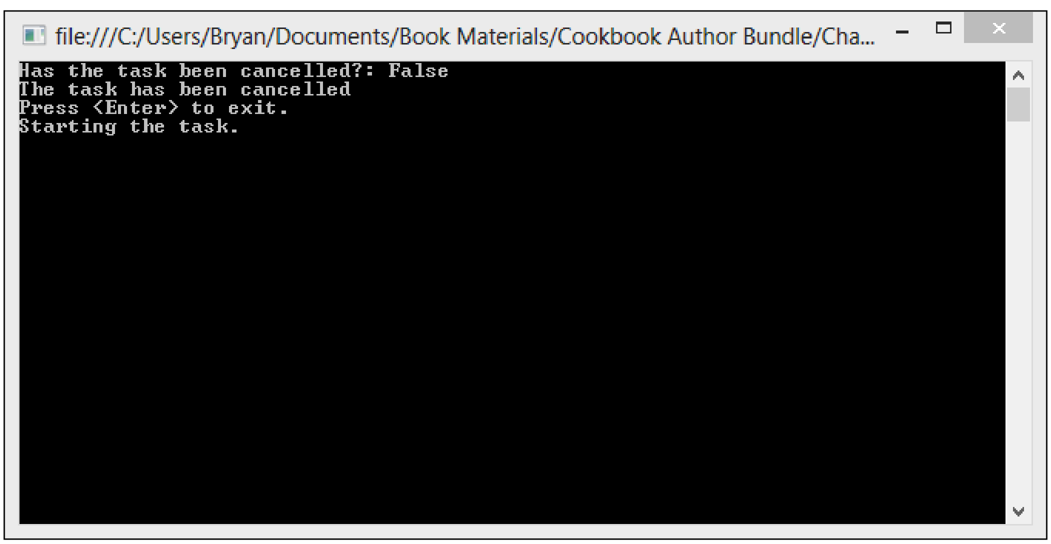
7. The following are the last statements put in a condition to check whether the task has been cancelled or faulted before we try to write out the results:

```
if (!task1.IsCanceled || !task1.IsFaulted)
{
    try
    {
        if (!task1.IsFaulted)
        {
          Console.WriteLine("Origin of Specied word count: {0}",
task1.Result);
        }
    }
    catch (AggregateException aggEx)
    {
    foreach (Exception ex in aggEx.InnerExceptions)
    {
          Console.WriteLine("Caught exception: {0}", ex.Message);
    }
    }
}
else
{
    Console.WriteLine("The task has been cancelled");
}
```

8. Lastly, we'll finish up by prompting the user to exit and waiting for the input.

```
Console.WriteLine("Press <Enter> to exit.");
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

The basic idea of cancelling `task` is that we create `CancellationTokenSource`, obtain `CancellationToken` from it, and then pass `CancellationToken` onto the `Task` constructor. Once we have done that, we can call the `Cancel` method on the `CancellationTokenSource` to cancel the task.

That's all easy enough. However, inside `task` we have a couple of options on how to handle the cancellation.

If your task has resources that need to be cleaned up (such as the `WebClient`), you need to check the cancellation tokens `IsCancellationRequested` property, then dispose of the resources, and throw a new `OperationCancelledException`.

The other option, if your task doesn't use resources which need to be explicitly cleaned up, is to use the token `ThrowIsCancellationRequested()`, which will ensure the task transitions to a status of cancelled in a single statement.

If you need to execute `task` and prevent it from being cancelled, you can obtain a special `CancellationToken` that is not associated with any `CancellationTokenSource` from the static `CancellationToken.None` property, and pass this token to `Task`. Since there is no associated `CancellationTokenSource`, it is not possible to call `Cancel` for this token, and any code that is checking the `CancellationToken.IsCancellationRequested` property will always get back a false.

There's more...

You can register one or more methods to be called when `CancellationTokenSource` is cancelled, by registering a `callback` method with the `CancellationToken.Register` method. You just need to pass `Action<Object>`, and optionally, a state value will be passed to `callback` and a `Boolean`, indicating whether to invoke the delegate using `SynchronizationContext` of the calling thread to the `Register` method. Passing in a value of false means the thread that calls `Cancel` will invoke the registered methods synchronously. If you pass true, the callbacks will be sent to `SynchronizationContext`, which determines which thread will invoke the callbacks.

```
var source = new CancellationTokenSource();
source.Token.Register(()=>
{
  Console.WriteLine("The operation has been cancelled.");
});
```

The `Register` method of `CancellationToken` returns a `CancellationTokenRegistration` object. To remove a registered callback from `CancellationTokenSource`, so it doesn't get invoked, call the `Dispose` method of the `CancellationTokenRegistration` object.

You can also create `CancellationTokenSource` by linking other `CancellationTokenSource` objects together. The new composite `CancellationTokenSource` will be cancelled, if any of the linked `CancellationTokenSource` objects are cancelled.

```
var source1 = new CancellationTokenSource();
var source2 = new CancellationTokenSource();
var linkedSource = CancellationTokenSource.
CreateLinkedTokenSource(source1.Token, source2.Token);
```

Cancelling one of many tasks

Now that we have seen how to cancel a task, let's take a look at how we can use `CancellationToken` to cancel multiple tasks with a single call to `CancellationTokenSource.Cancel()`.

How to do it...

Now let's return to our WordCount example and create a `Console` application that provides for the cancellation of multiple tasks with single `CancellationToken`.

1. Start a new project using the C# **Console Application** project template, and assign `WordCount8` as the **Solution name**.
2. Add the following `using` statements at the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. First, let's create a `helper` method to display errors and cancellation status. Since we have multiple tasks, it's better to have this logic all in one place. Following the `Main` method of your `Program` class, create a `static` method call `HandleExceptions` which will display the errors and task status to the user.

   ```
   private static void DisplayException(Task task, AggregateException
   outerEx, string bookName)
   ```

```
{
    foreach (Exception innerEx in outerEx.InnerExceptions)
    {
      Console.WriteLine("Handled exception for
{0}:{1}",bookName,innerEx.Message);
    }
    Console.WriteLine("Cancellation status for book {0}: {1}",
bookName, task.IsCanceled);
}
```

4. Next, at the top of the `Main` method, create `CancellationTokenSource` and get our `CancellationToken`.

```
//Create a cancellation token source
CancellationTokenSource tokenSource = new
CancellationTokenSource();
//get the cancellation token
CancellationToken token = tokenSource.Token;
```

5. Now we need to create our tasks and our array of `delimiters`. The tasks are the same as in the recipe for cancelling a single task. The key here is that we are passing the same `CancellationToken` in for all three tasks.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";

 Task<int> task1 = Task.Factory.StartNew(() =>
{
    // wait for the cancellation to happen
    Thread.Sleep(2000);
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    if (token.IsCancellationRequested)
    {
      client.Dispose();
      throw new OperationCanceledException(token);
    }
    else
    {
      var words = client.DownloadString(@"http://www.gutenberg.
org/files/2009/2009.txt");
      Console.WriteLine("Starting the task for Origin of
Species.");
```

```
      var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
      return wordArray.Count();
    }
},token);


 Task<int> task2 = Task.Factory.StartNew(() =>
{
    // wait for the cancellation to happen
    Thread.Sleep(2000);
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    if (token.IsCancellationRequested)
    {
      client.Dispose();
      throw new OperationCanceledException(token);
    }
    else
    {
      var words = client.DownloadString(@"http://www.gutenberg.
org/files/16328/16328-8.txt");
      Console.WriteLine("Starting the task for Beowulf.");
      var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
      return wordArray.Count();
    };
},token);

Task<int> task3 = Task.Factory.StartNew(() =>
{
    // wait for the cancellation to happen
    Thread.Sleep(2000);
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    if (token.IsCancellationRequested)
    {
      client.Dispose();
      throw new OperationCanceledException(token);
    }
    else
    {
```

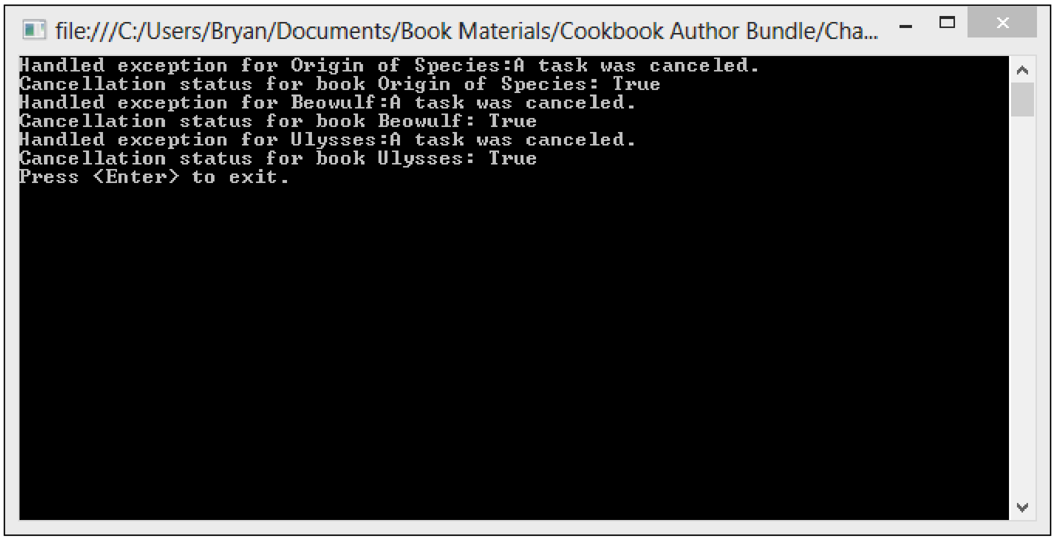
```
      var words = client.DownloadString(@"http://www.gutenberg.
org/files/4300/4300.txt");
      Console.WriteLine("Starting the task for Ulysses.");
      var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
      return wordArray.Count();
    };
},token);
```

6. OK, let's finish up by calling `CancellationTokenSource.Cancel()`, checking the results, and catching the exceptions. The remainder of the `Main` method should look as the following code snippet:

```
//Cancel the token source
tokenSource.Cancel();
try
{
    if (!task1.IsFaulted || !task1.IsCanceled)
    {
      Console.WriteLine("Origin of Specied word count: {0}",
task1.Result);
    }
}
catch(AggregateException outerEx1)
{
    DisplayException(task1, outerEx1, "Origin of Species");
}
try
{
    if (!task2.IsFaulted || !task2.IsCanceled)
    {
      Console.WriteLine("Beowulf word count: {0}", task2.Result);
    }
}
catch (AggregateException outerEx2)
{
    DisplayException(task2, outerEx2, "Beowulf");
}
try
{
    if (!task3.IsFaulted || !task3.IsCanceled)
    {
      Console.WriteLine("Ulysses word count: {0}", task3.Result);
    }
}
catch (AggregateException outerEx3)
{
   DisplayException(task3, outerEx3, "Ulysses");
```

```
    }
    Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

Functionally, cancelling multiple tasks is the same as cancelling a single task. In fact, the Parallel Extensions team has put a lot of work into making cancellation of various parallel structures very similar, as you will see as we go through the book.

All that is necessary to cancel multiple tasks is to create `CancellationToken`, then pass that token into all of the tasks you wish to cancel as shown in the following code snippet:

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;

Task<int> task1 = Task.Factory.StartNew(() =>
{
  ...
},token);
Task<int> task, = Task.Factory.StartNew(() =>
{
  ...
},token);

tokenSource.Cancel();
```

2

Implementing Continuations

In this chapter, we will cover:

- Continuing a task
- Passing task results to a continuation
- Continue "WhenAny" and "WhenAll"
- Specifying when a continuation will run
- Using a continuation for exception handling
- Cancelling a continuation
- Using a continuation to chain multiple tasks
- Using a continuation to update a UI

Introduction

When you are writing an application that has tasks and that execute in parallel, it is common to have some parallel tasks that depend on the results of other tasks. These tasks should not be started until the earlier tasks, known as antecedents, have been completed.

In fact, to write truly scalable software, you should not have threads that block. Calling `Wait` or querying `Task.Result`, when the task has not finished running, will cause your threads to block. Fortunately, there is a better way.

Prior to the introduction of the **Task Parallel Library** (**TPL**), this type of interdependent thread execution was done with callbacks, where a method was called, and one of its parameters was a delegate to execute when the task completed. This provided a viable solution to the dependency problems but quickly became very complex in the real-world application. This is especially true if, for example, you had a task that needed to run after several other tasks had completed.

With the TPL, a simpler solution exists in the form of continuation tasks. These tasks are linked to their antecedents, and are automatically started after the earlier tasks have been completed.

What makes continuations so powerful is that, you can create continuations that run when a task or a group of tasks completes throws an exception, or gets cancelled. As you will see in this chapter, continuations can even provide a means to synchronize the asynchronous method results with the user interface running on another thread.

We will start the chapter with a basic, simple continuation that runs when a single task completes. From there, we will look at using continuations to control a collection of tasks, using continuations to handle exceptions, and using continuations to chain multiple tasks together. We will finish the chapter by creating a **Windows Presentation Foundation** (**WPF**) application, using a continuation to marshal data created in a task back to the user interface.

Continuing a task

In its simplest form, a continuation is an action that runs asynchronously after a target task, called an antecedent, completes.

In the first recipe of this chapter, we will build a basic continuation. We will accomplish this by using the `Task.ContinueWith(Action<Task>)` method.

How to do it...

Let's go to Visual Studio and create a console application that runs a task continuation after our word count task completes. The steps to create a console application are as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation1` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Linq;
using System.Net;
using System.Threading;
using System.Threading.Tasks;
```

3. Now let's put a try/catch block and some basic exception handling. The `Main` method of the program class, at this point, should look as shown in the following code snippet:

```
static void Main()
{
  try
  {
  // The Task and Continuation will go here
  }
  catch (AggregateException aEx)
  {
    foreach (Exception ex in aEx.InnerExceptions)
    {
      Console.WriteLine("An exception has occured:
        {0}" + ex.Message);
    }
  }
}
```

4. Inside the `try` block, create a `WebClient` object and set the user-agent header as shown in the following code snippet:

```
var client = new WebClient();
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
  Windows NT 6.1; Trident/6.0)";
client.Headers.Add("user-agent", headerText);
```

5. Next, in the body of the `try` block, let's create an anonymous `Task` (no name), followed by a `.ContinueWith()` right after the closing parenthesis of the `Task`. The antecedent `Task` doesn't return any results in this recipe.

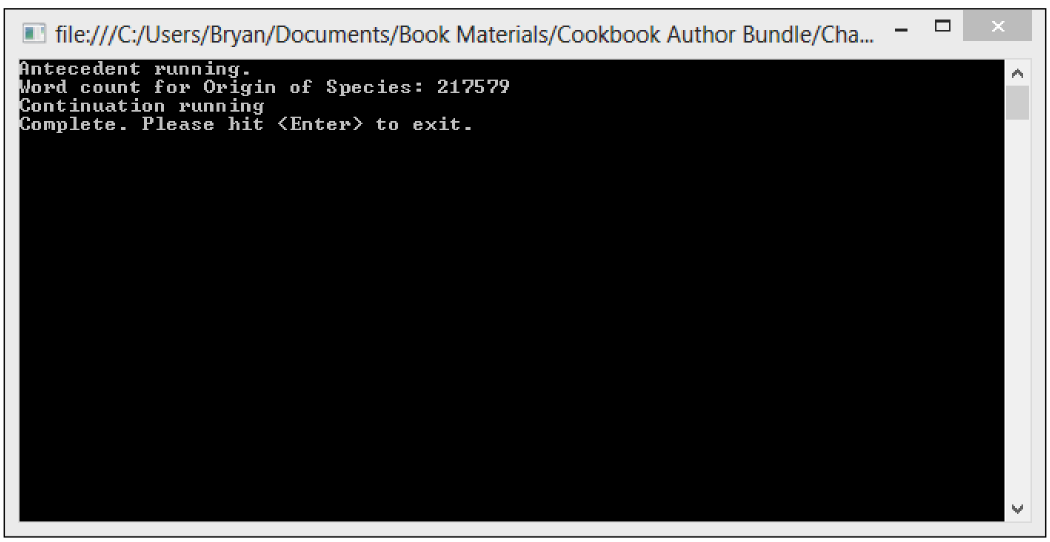
```
Task.Factory.StartNew(() =>
{
}).ContinueWith(obj =>
{
}).Wait();
```

6. Finally, we need to create the body of the `Task` and the continuation. The `Task` will execute one of our familiar word counts. The continuation will be used to clean up the reference to the `WebClient` object after the antecedent task completes. After the continuation, prompt the user to exit.

```
Task.Factory.StartNew(() =>
  {
    Console.WriteLine("Antecedent running.");
```

```
        char[] delimiters = { ' ', ',', '.', ';', ':', '-',
          '_', '/', '\u000A' };
        var words =
client.DownloadString(@"http://www.gutenberg.org/files/2009
  /2009.txt");
        var wordArray = words.Split(delimiters,
          StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Word count for Origin of Species:
          {0}", wordArray.Count());
      }
).ContinueWith(antecedent =>
  {
    Console.WriteLine("Continuation running");
    client.Dispose();
  }).Wait();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

There isn't a lot to explain about this basic continuation, but there are a couple of small points to note.

For this recipe, we created an anonymous `Task` and made the call to `ContinueWith` right after the closing parenthesis of the task as follows:

```
Task.Factory.StartNew(() =>
{
}).ContinueWith(obj =>
{
}).Wait();
```

We could just as well have created a named task and made the call to `ContinueWith` in a separate statement shown as follows:

```
Task task1 = Task.Factory.StartNew(() =>
{
});
task1.ContinueWith(obj =>
{
}).Wait();
```

Also, notice that we can wait for a continuation using the `Wait()` method; in the same way we could wait for a `Task` (however, you will not normally do this in practice. It causes the thread to block waiting for the continuation to complete. In general, you want to avoid causing your threads to block). In fact, tasks and continuations aren't much different and have many of the same instance methods and properties.

Passing task results to a continuation

In this recipe, we will see how we can pass the results returned from an antecedent `Task` to a continuation.

Our antecedent `Task` is going to read in the contents of a book as a string and display a word count to the user. The continuation, which will run after the antecedent completes, will take the string array returned by the antecedent and perform a LINQ query which will find the five most frequently used words.

How to do it...

Let's start Visual Studio and build a Console Application that shows how to pass results from the antecedent to a continuation. The steps are given as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation2` as the **Solution name**.

2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
```

3. To begin with, let's put some basic stuff in the class. We will need a character array of delimiters so that we can parse out the words properly. Also, we need a try/catch block and some basic exception handling. The `Main` method of the Program class, at this point, should look as follows:

```
static void Main()
{
    char[] delimiters = { ' ', ',', '.', ';', ':', '-',
      '_', '/', '\u000A' };
    try
    {
    // The Task and Continuation will go here
    }
    catch (AggregateException aEx)
    {
      foreach (Exception ex in aEx.InnerExceptions)
      {
        Console.WriteLine("An exception has occured: {0}" +
          ex.Message);
      }
    }
}
```

4. Now let's create a task called `task1` that returns an array of strings as its result. The purpose of `task1` will be to create `System.Net.WebClient` which will read in the text of the book as a string. Once the string is parsed and put into a string array, we will display the word count to the user by using the `Count` method of the array, and then return the array in the tasks result so that it can be used in our continuation. Create the task inside the `try` block. The body of the `try` block should now look something like the following code:

```
try
{
    Task<string[]> task1 = Task.Factory.StartNew(() =>
    {
      var client = new WebClient();
      const string headerText = "Mozilla/5.0 (compatible;
        MSIE 10.0; Windows NT 6.1; Trident/6.0)";
```

```
        client.Headers.Add("user-agent",headerText);
        var words =
client.DownloadString(@"http://www.gutenberg.org/files/2009
  /2009.txt");
        string[] wordArray = words.Split(delimiters,
          StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Word count for Origin of Species:
          {0}", wordArray.Count());
        Console.WriteLine();
        return wordArray;
    }
}
```

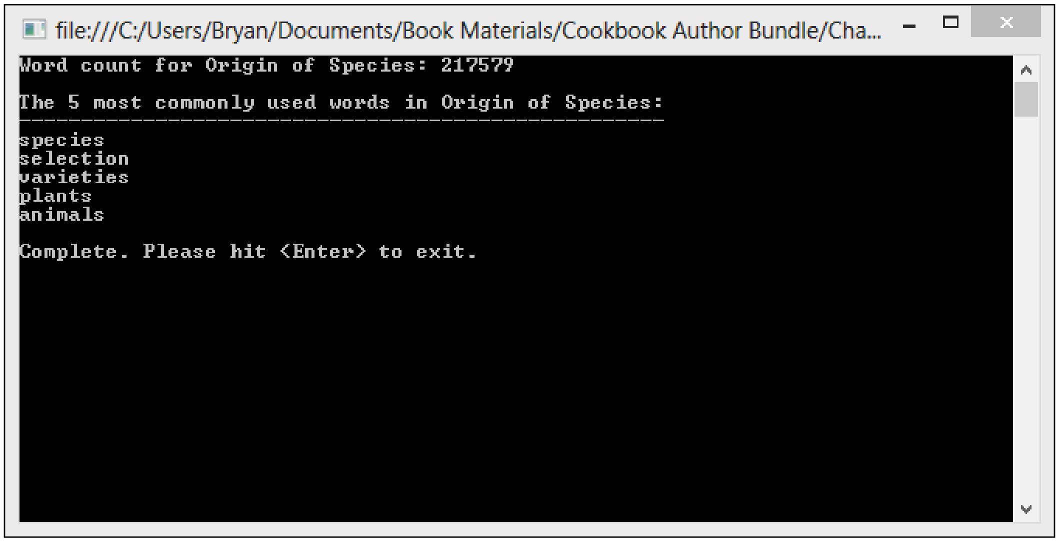
5. Next, we are going to create our continuation using the `Task.ContinueWith()` method. Our continuation will have a `Task<string[]> state` parameter. The body of the continuation will perform a Linq query on the string array to sort all of the words contained in the array by the number of times the words occur. We will then execute another Linq operation to take the top five most frequently used words and write them to the console. Finally, we will want to wait on the continuation to complete with the `Wait()` method. Create the task continuation right after the body of the antecedent task.

```
task1.ContinueWith(antecedent =>
{
  var wordsByUsage = antecedent.Result.Where(word =>
    word.Length > 5)
  .GroupBy(word => word)
  .OrderByDescending(grouping => grouping.Count())
  .Select(grouping => grouping.Key);
  var commonWords = (wordsByUsage.Take(5)).ToArray();
  Console.WriteLine("The 5 most commonly used words in
    Origin of Species:");
  Console.WriteLine("-----------------------------------
    ----------------");
  foreach (var word in commonWords)
  {
    Console.WriteLine(word);
  }
}).Wait();
```

6. OK, the last step for this recipe is to let the user know that our application is finished and prompt them to exit. Put that code right after the continuation. It should be the last lines in the `try` block.

```
Console.WriteLine();
Console.WriteLine("Complete. Please hit <Enter> to exit.");
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see the output similar to the following screenshot:



How it works...

The continuation in this recipe was created using the ContinueWith method of an existing task instance as we did in the previous recipe. In this recipe however, we use a Lambda expression to pass in a Task<string[]> parameter representing the antecedent Task.

```
Task<string[]> task1 = Task.Factory.StartNew(() =>
{
  //Task Action
});
task1.ContinueWith(antecedent =>
{
  //Continuation Action
});
```

Notice that the continuation accesses the result of the antecedent using the Task.Result property. If this looks familiar, it should. You access the results of a task in nearly the same way in a continuation as you would in any piece of your code, that is, by accessing the Result property of a Task. The Parallel Extensions team has made the coding experience very consistent across all parallel operations.

```
task1.ContinueWith(antecedent =>
{
```

```
    var wordsByUsage = antecedent.Result.Where(word => word.Length >
      5)
    .GroupBy(word => word)
    .OrderByDescending(grouping => grouping.Count())
    .Select(grouping => grouping.Key);
    var commonWords = (wordsByUsage.Take(5)).ToArray();
    Console.WriteLine("The 5 most commonly used words in Origin of
      Species:");
    Console.WriteLine("---------------------------------------------
      -------");
    foreach (var word in commonWords)
    {
      Console.WriteLine(word);
    }
  });
```

Lastly, we wait for the continuation to complete before prompting the user to exit.

Continue "WhenAny" and "WhenAll"

In this recipe we will move from continuing single tasks to setting up continuations for groups of tasks. The two methods we will be looking at are `WhenAny` and `WhenAll`. Both methods are static members of the `Task.Factory` class, and take an array of tasks and `Action<Task>` as their parameters.

First we will look at the `WhenAny` continuations. The basic idea here is that we have a group of tasks and we only want to wait for the first and fastest of the group to complete its work before moving on. In our case, we will be downloading the text of three different books, and performing a word count on each. When the first task completes we will display the word count of the winner to the user.

After that we will change to `WhenAll` and display the results of all three word counts to the user.

How to do it...

Let's build a solution that shows how to conditionally continue a task. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation3` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Net;
using System.Threading.Tasks;
```

3. First, in the `Main` method of your program class, let's create a character array of delimiters we can use to split our words with, a string constant for the user agent header of our web client, and a `Dictionary<string, string>` method to hold our book titles and URLs. The dictionary will serve as the state object parameter for our tasks, which will be created in a `foreach` loop.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_',
  '/', '\u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE
  10.0; Windows NT 6.1; Trident/6.0)";
var dictionary = new Dictionary<string, string>
{
  {"Origin of Species",
    "http://www.gutenberg.org/files/2009/2009.txt"},
    {"Beowulf",
      "http://www.gutenberg.org/files/16328/16328-8.txt"},
    {"Ulysses",
      "http://www.gutenberg.org/files/4300/4300.txt"}
};
```

4. Next, let's create a try/catch block with some basic error handling.

```
try
{
  // Loop to create and Continuation will go here
}
catch (AggregateException aEx)
{
  foreach (Exception ex in aEx.InnerExceptions)
  {
    Console.WriteLine("An exception has occured: {0}" +
      ex.Message);
  }
}
```

5. Inside the `try` block, let's create a new list of `Task<KeyValuePair<string, string>>`. Of course, this will be the list of our tasks. Each task will take a `KeyValuePair` from the dictionary we created in step 3 as their state parameters.

```
var tasks = new List<Task<KeyValuePair<string, int>>>();
```

6. Now let's create our task in a `foreach` loop. Each task will read the text of a book from a string, split the string into a character array, and do a word count. Our antecedent tasks return a `KeyValuePair<string, int>` with the book title and the word count for each book.

```
foreach (var pair in dictionary)
{
  tasks.Add(Task.Factory.StartNew(stateObj =>
  {
    var taskData = (KeyValuePair<string, string>)stateObj;
    Console.WriteLine("Starting task for {0}",
      taskData.Key);
    var client = new WebClient();
    client.Headers.Add("user-agent", headerText);
    var words = client.DownloadString(taskData.Value);
    var wordArray = words.Split(delimiters,
      StringSplitOptions.RemoveEmptyEntries);
    return new KeyValuePair<string, int>(taskData.Key,
      wordArray.Count());
  }, pair));
}
```

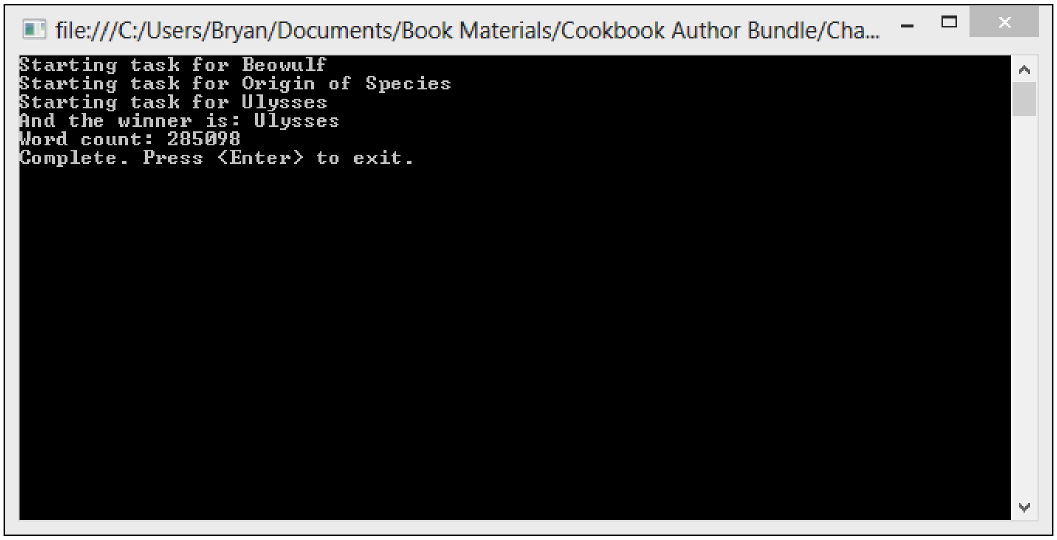
7. Now let's create the continuation by calling the `Task.Factory.WhenAny` method. The continuations will just display the title and word count of the winner to the user.

```
Task.Factory.ContinueWhenAny(tasks.ToArray(), antecedent =>
{
    Console.WriteLine("And the winner is: {0}", antecedent.Result.
Key);
    Console.WriteLine("Word count: {0}", antecedent.Result.Value);
}).Wait();
```

8. Lastly, after the catch block, prompt the user to exit and wait for the input.

```
Console.WriteLine("Complete. Press <Enter> to exit.");
Console.ReadLine();
```

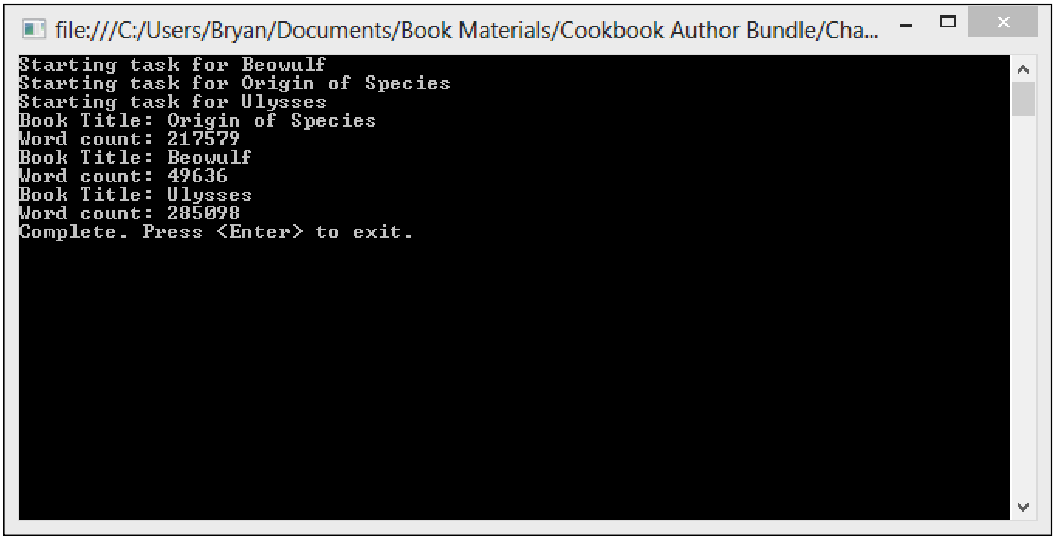
9. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following. Your winner may vary.



10. Before moving on, let's change our code a bit and continue when all of our tasks complete. All we need to do is change our method call from `Task.Factory.WhenAny` to `Task.Factory.WhenAll`, change the name of the continuation parameter from `antecedent` to `antecedents` to reflect plurality, and create a `foreach` loop in the body of the continuation to loop through the results.

```
Task.Factory.ContinueWhenAll(tasks.ToArray(), antecedents =>
{
    foreach (var antecedent in antecedents)
    {
        Console.WriteLine("Book Title: {0}", antecedent.Result.
Key);
        Console.WriteLine("Word count: {0}", antecedent.Result.
Value);
    }
}).Wait();
```

11. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

The continuations in this recipe are created a bit differently from the continuations that we have created in previous tasks. Instead of calling the instance method ContinueWith on a Task variable, we are calling the ContinueWhenAny and ContinueWhenAll static methods on Task.FactoryClass.

```
Task.Factory.ContinueWhenAll(tasks.ToArray(), antecedents =>
{
});
```

The ContinueWhenAny and ContinueWhenAll methods have a different parameter lists than Task.ContinueWith.

ContinueWhenAny takes an array of Task as its first parameter and a single Action<Task> delegate as its second parameter.

```
ContinueWhenAny(Task[], Action<Task>)
```

ContinueWhenAll takes the same array of Task as its first parameter and Action<Task[]> as its second parameter.

```
ContinueWhenAll(Task[], Action<Task[]>)
```

Specifying when a continuation will run

One of the most powerful features of task continuations is the ability to create multiple continuations for a task, and specify the exact conditions under which each continuation will be invoked by using the `Task.TaskContinuationOptions` enumeration.

When you create a continuation for a task, you can use `Task.ContinueWith` overload that takes the `TaskContinuationOptions` enumeration to specify that the continuation will only run if the antecedent `Task` completed, was cancelled, or is faulted. The enumeration also has members that specify when a continuation should not run.

In this recipe, we will be looking at two simple tasks, each with two continuations. One of the continuations for each task will run when the task completes, and one will run when the task is cancelled.

How to do it...

Now, let's create a console application that continues tasks conditionally. The steps to create a console application are as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation4` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. At the top of the `Main` method, create two `CancellationTokenSource` objects and get a `CancellationToken` from each one of them.

```
var tokenSource1 = new CancellationTokenSource();
var token1 = tokenSource1.Token;

 var tokenSource2 = new CancellationTokenSource();
 var token2 = tokenSource2.Token;
```

4. Next, let's create a try/catch block with some basic error handling.

```
try
{
    // Tasks and Continuations will go here
}
catch (AggregateException aEx)
```

```
{
    foreach (Exception ex in aEx.InnerExceptions)
    {
      Console.WriteLine("An exception has occured: {0}" +
ex.Message);
    }
}
```

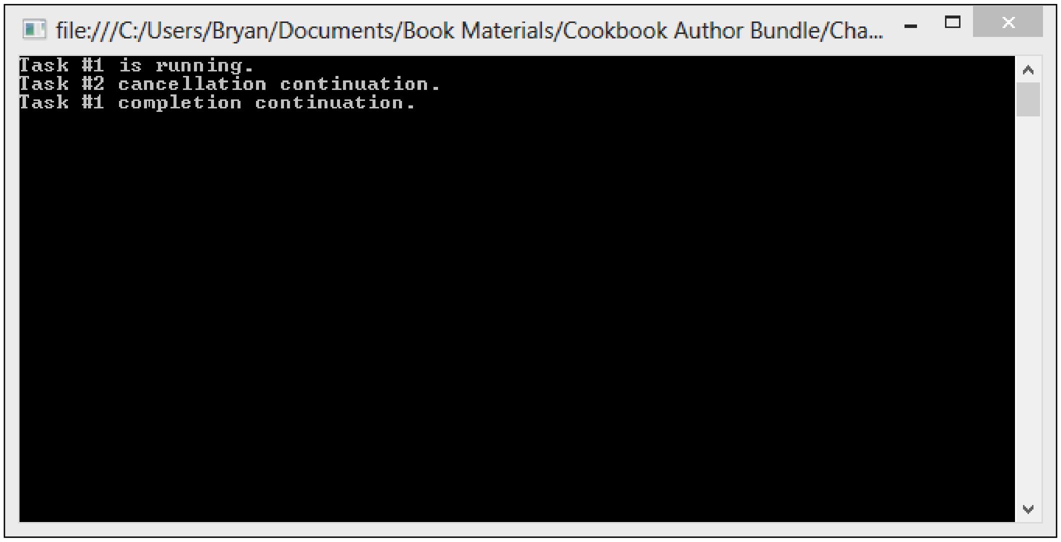
5. Inside the `try` block, let's create two simple tasks. Both tasks just write a message to the console. Also create two continuations for each task using `TaskContinuationOptions.OnlyOnRanToCompletion` and `Task ContinuationOption.OnlyOnFaulted`.

```
var task1 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Task #1 is running.");
    //wait a bit
    Thread.Sleep(2000);
}, token1);

task1.ContinueWith(antecedent => Console.WriteLine("Task #1
  completion continuation."),
  TaskContinuationOptions.OnlyOnRanToCompletion);
task1.ContinueWith(antecedent => Console.WriteLine("Task #1
  cancellation continuation."),
  TaskContinuationOptions.OnlyOnCanceled);

var task2 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Task #2 is running.");
    //wait a bit
    Thread.Sleep(2000);
}, token2);

task2.ContinueWith(antecedent => Console.WriteLine("Task #2
  completion continuation."),
  TaskContinuationOptions.OnlyOnRanToCompletion);
task2.ContinueWith(antecedent => Console.WriteLine("Task #2
  cancellation continuation."),
  TaskContinuationOptions.OnlyOnCanceled);
```

6. Lastly, after the `catch` block, let's cancel the token and wait for user input before exiting.

```
tokenSource2.Cancel();
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

In this very simple example, we started by creating two `CancellationTokenSource` objects and getting a cancellation token source from each. If we had created a `CancellationTokenSource` object and passed the token into both tasks, both tasks would have been cancelled when we cancelled the token. In our case, we just wanted to cancel one of the two tasks.

The tasks themselves are very simple. They just wait for a bit to give us some time to cancel the token and display a message to the console. We pass one `CancellationToken` into each task as shown in the following code snippet:

```
var task1 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Task #1 is running.");
    //wait a bit
    Thread.Sleep(2000);
}, token1);

var task2 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Task #2 is running.");
```

```
        //wait a bit
        Task2.Delay(2000);
    }, token2);
```

Both of the continuations just display a message to the console, and both are created with a member of the `Task.TaskContinuationOptions` enumerator. The first continuation is fired when the task runs to completion and the second continuation fires when the task is cancelled.

```
task1.ContinueWith(antecedent => Console.WriteLine("Task #1
  completion continuation."),
  TaskContinuationOptions.OnlyOnRanToCompletion);

task1.ContinueWith(antecedent => Console.WriteLine("Task #1
  cancellation continuation."),
  TaskContinuationOptions.OnlyOnCanceled);
```

We cancel the token for `task2`, but not for `task1` and the corresponding continuation for each executes, and we can see the message written on to the console.

There's more...

The `TaskContinuationOptions` enumeration has several members which control under which condition a continuation is triggered. The following table contains a list of these members. Note that this is not a complete list of continuation options. The complete list of continuation options can be found at `http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskcontinuationoptions.aspx`. The `OnlyOnFaulted` member will have its own recipe later in the chapter.

`NotOnRanToCompletion`	The continuation should not be scheduled if the task ran to completion.
`NotOnFaulted`	The continuation should not be scheduled if the task faulted.
`NotOnCancelled`	The continuation should not be triggered if the task was cancelled.
`OnlyOnRanToCompletion`	The continuation should be scheduled if the task ran to completion.
`OnlyOnFaulted`	The continuation should be scheduled if the task faulted.
`OnlyOnCancelled`	The continuation should be triggered if the task was cancelled.

The `TaskContinuationOptions` enumeration can be treated as a bit field and a bitwise combination can be performed on its members.

Using a continuation for exception handling

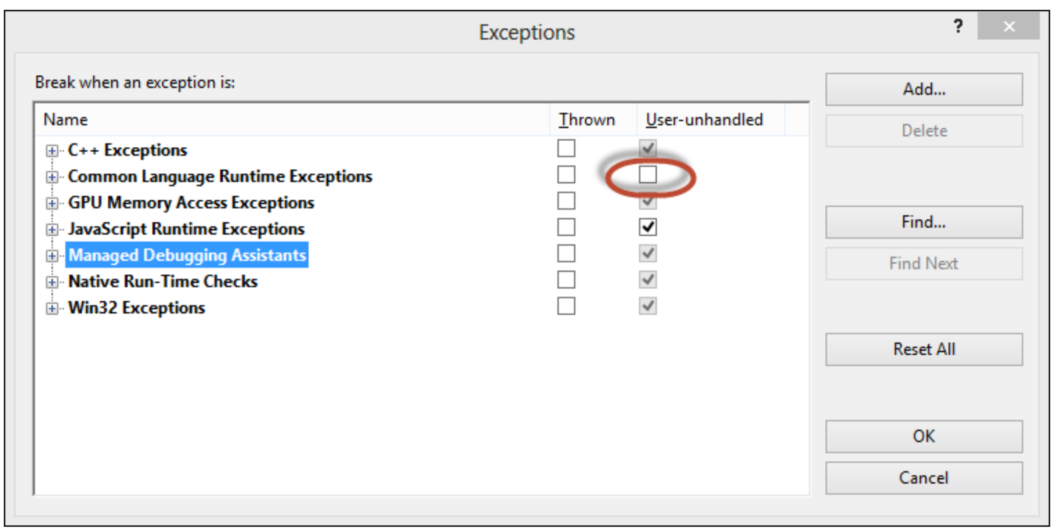
In the *Handling task exceptions using try/catch* recipe, in *Chapter 1*, *Getting Started With Task Parallel Library* we looked at how to handle exceptions in task. In addition to the techniques used in that recipe, you can also use continuations to handle task exceptions. By using a continuation, we can handle errors in a cleaner, less inline way. An exception handling continuation allows for centralizing exception handling logic in cases where you would want to provide logging or other exception related code.

The basic concept is to use the `Task.TaskContinuationOptions` enumeration so we can create a continuation that will be scheduled if the task ran to completion, and another continuation that will be scheduled if the task is put into a faulted state.

Getting Ready

For this recipe we need to turn off the **Visual Studio 2012 Exception Assistant**. The Exception Assistant appears whenever a run-time exception is thrown and intercepts the exception before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Now, let's go to Visual Studio and see how to use a continuation for exception handling. The steps are given as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation5` as the **Solution name**.
2. Add the following `using` directives at the top of your program class:

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. In the `Main` method of your program class, create a `Task`. The task doesn't need to accept a state parameter or return anything. In the body of the `Task`, create the try/finally blocks. In order to have a resource to dispose of, create a new WebClient in the `try` block, and then throw an exception. In the `finally` block, call the dispose method of the WebClient. Other than that, the exact details don't matter much.

```
Task task1 = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Starting the task.");
    var client = new WebClient();
    const string headerText = "Mozilla/5.0 (compatible;
      MSIE 10.0; Windows NT 6.1; Trident/6.0)";
    client.Headers.Add("user-agent", headerText);
    try
    {
        var book = client.DownloadString(@"http://www.gutenberg.
org/files/2009
  /2009.txt");
        var ex = new WebException("Unable to download book
          contents");
        throw ex;
    }
    finally
    {
        client.Dispose();
        Console.WriteLine("WebClient disposed.");
    }
});
```

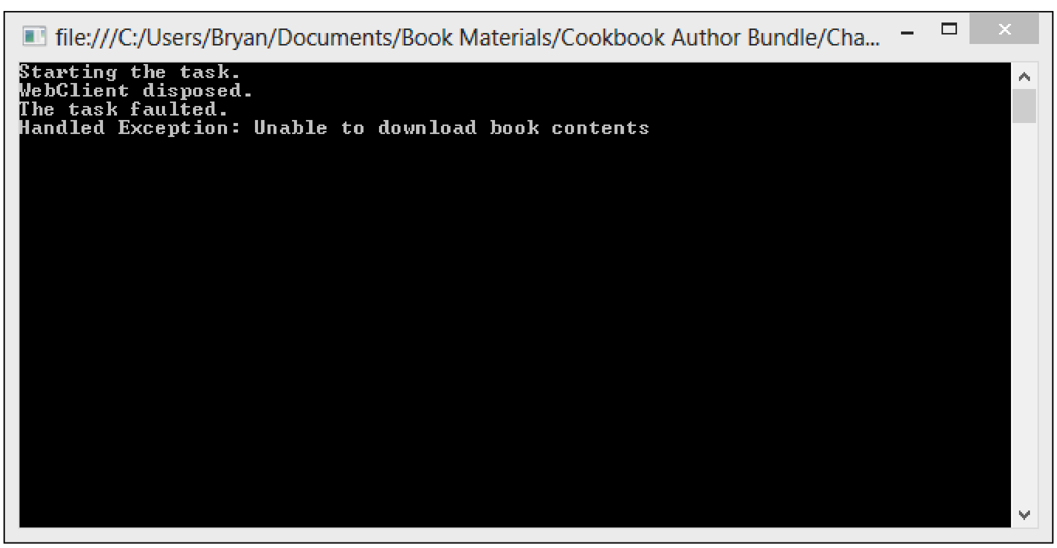
4. Immediately following the `Task`, use `TaskContinuationOptions.OnlyOnRanToCompletion` to create a trivial continuation to run when the task completes successfully. This continuation only needs to write a message to the console.

```
task1.ContinueWith(antecedent=>
{
    Console.WriteLine("The task ran to   completion."),
}, TaskContinuationOptions.OnlyOnRanToCompletion);
```

5. Next use `TaskContinuationOptions.OnlyOnFaulted` to create a continuation that only runs when `task1` throws a fault. After the continuation, add `Console.Readline` to wait for user input before exiting.

```
task1.ContinueWith(antecedent =>
{
    Console.WriteLine("The task faulted.");
    var aEx = antecedent.Exception;
    if (aEx != null)
    foreach (var ex in aEx.InnerExceptions)
    {
        Console.WriteLine("Handled Exception: {0}",ex.Message);
    }
}, TaskContinuationOptions.OnlyOnFaulted);
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the one shown in the following screenshot:



How it works...

Creating a continuation that will run when a `Task` is in a faulted state, works the same as setting any of the other enumerations in `TaskContinuationOptions` on a continuation.

In order to properly clean up resources used by the `Task`, we created `try/finally` blocks in our task and disposed of the WebClient in the `finally` block:

```
finally
{
    client.Dispose();
    Console.WriteLine("WebClient disposed.");
}
```

Our exception handling continuation checks to see if the `AggregateException` is null before looping through the `InnerExceptions` collection, and writing the result to the console. The null check isn't strictly necessary because the antecedent task needs to be in a faulted state before the continuation is scheduled, but it is a good defensive coding practice none the less:

```
task1.ContinueWith(antecedent =>
{
    Console.WriteLine("The task faulted.");
    var aEx = antecedent.Exception;
    if (aEx != null)
        foreach (var ex in aEx.InnerExceptions)
        {
            Console.WriteLine("Handled Exception: {0}",ex.Message);
        }
}, TaskContinuationOptions.OnlyOnFaulted);
Console.ReadLine();
```

Cancelling a continuation

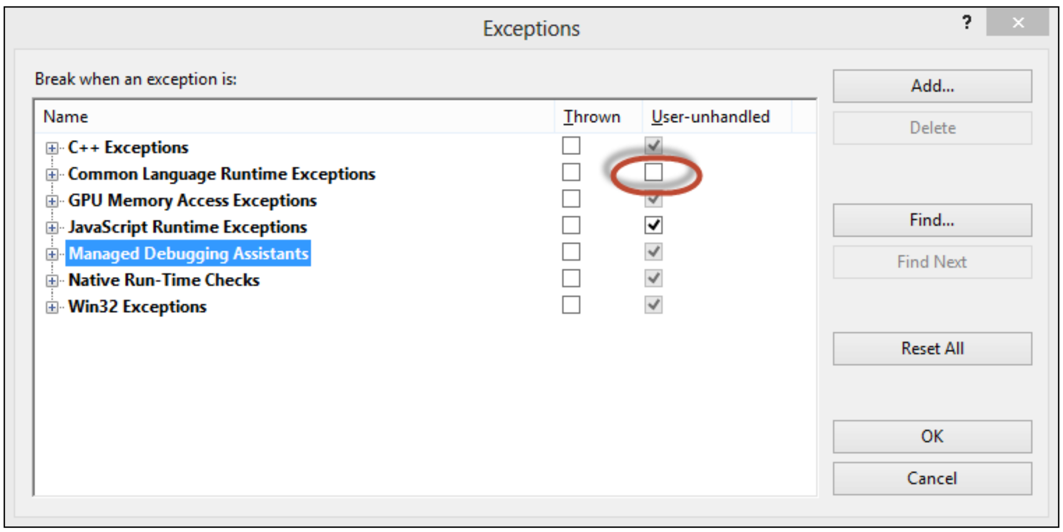
Cancelling a continuation follows the same basic rules as cancelling a `Task`. If a `Task` and its continuation are two parts of the same operation, you can pass the same cancellation token to both the `Task` and the continuation.

In this recipe we will have a simple `Task` that creates a list of numbers and a continuation that squares the numbers and return a result. After a few seconds of running, we will use the token to cancel both the `Task` and the continuation.

Getting Ready

Since cancelling a `Task` or continuation raises and `OperationCanceledException` we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime exception is thrown, and intercepts the exception before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Now, let's build a console application so that we can see how to cancel a continuation. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation6` as the **Solution name**.
2. Add the following `using` directives to the top of your program class.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

3. In the `Main` method of your program class, let's start by creating our `CancellationTokenSource` and getting a token. We will pass this token to both the antecedent `Task` and the continuation.

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
```

4. Next let's add try/catch/finally blocks to the `Main` method, just under the previous lines. Add some basic error handling to the `catch` block and dispose of the `CancellationTokenSource` in the `finally` block.

```
try
{
//Task and Continuation go here
}
catch (AggregateException aEx)
{
    foreach (var ex in aEx.InnerExceptions)
    {
        Console.WriteLine("An exception has occured: " +
          ex.Message);
    }
}
finally
{
    tokenSource.Dispose();
}
```

5. Inside the `try` block, create a task that accepts an object state parameter. The parameter will determine the size of our number list. We will cast it to `Int32` and create a `for` loop to add numbers to our list. Also, pass the token created in step 1 to the task constructor.

```
var task1 = Task.Factory.StartNew(state =>
{
    Console.WriteLine("Task has started.");
    var result = new List<Int32>();
    for (var i = 0; i < (Int32) state; i++)
    {
        token.ThrowIfCancellationRequested();
        result.Add(i);
        Thread.Sleep(100); //sleep to simulate some work
```

```
        }
        return result;
    }, 5000,token);
```

6. After the `Task`, let's create our continuation. The continuation will receive the results from the antecedent `Task`, loop through the list, and square the numbers. Pass the same `CancellationToken` into the continuations constructor.

```
task1.ContinueWith(antecedent =>
{
    Console.WriteLine("Continuation has started.");
    var antecedentResult = antecedent.Result;
    var squares = new List<int>();
    foreach (var value in antecedentResult)
    {
        token.ThrowIfCancellationRequested();
        squares.Add(value*value);
        Thread.Sleep(100);//sleep to simulate some more work
    }
    return squares;
},token);
```

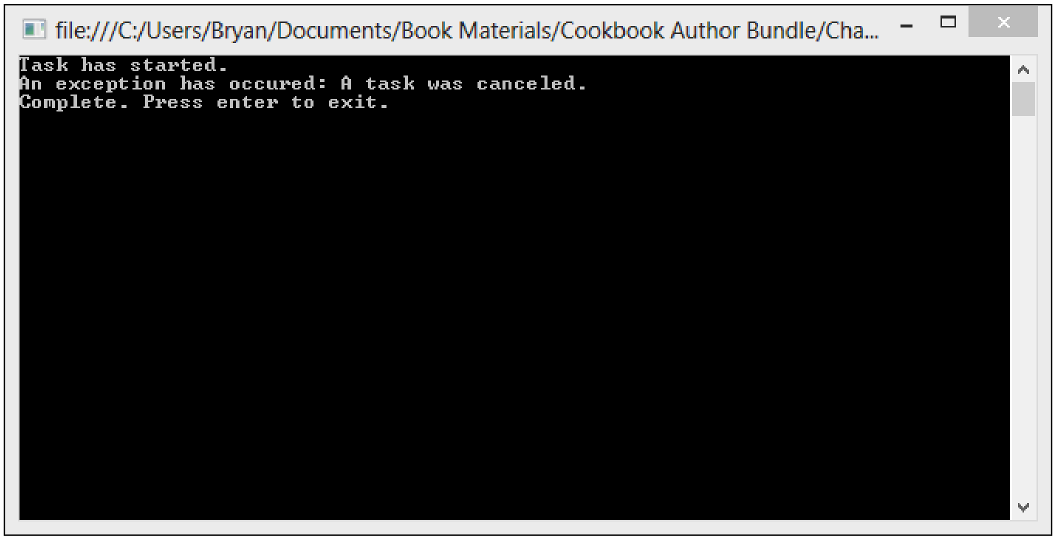
7. At the end of the `try` block, we need to sleep the thread a bit to give the `Task` and continuation some time to run, and then we will cancel the token. Finally we will call the `Wait` method on `task1`.

```
Thread.Sleep(2000); //wait for 2 seconds
tokenSource.Cancel();
task1.Wait();
```

8. Last, after the end of the `finally` block, write a message to the console that we are finished and wait for the user input.

```
Console.WriteLine("Complete. Press enter to exit.");
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

When an antecedent task throws an `OperationCancelledException` in response to a cancellation request, as long as the continuation uses the same `CancellationToken`, the cancellation request will be treated as an acknowledgement of co-operative cancellation and both the antecedent task and the continuation will go into a cancelled state.

This is pretty easy to accomplish. We just need to get a `CancellationToken` from a `CancellationTokenSource`, and pass the token to the constructors for both the antecedent `Task` and the continuation.

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var task1 = Task.Factory.StartNew(state =>
{
    // Task body
}, 5000,token);

task1.ContinueWith(antecedent =>
{
    //Continuation body
},token);
```

Inside the body of the loops in our `Task` and the continuation, we need to poll for cancellation and throw an `OperationCancelledException` if the token gets cancelled. This can be done in one line of code with the `ThrowIfCancellationRequested` method of the `CancellationToken` object.

```
foreach (var value in antecedentResult)
{
    token.ThrowIfCancellationRequested();
    squares.Add(value*value);
    Thread.Sleep(100);//sleep to simulate some more work
}
```

Lastly, we just need to make sure we are handling `AggregateExceptions` in our `catch` block.

Using a continuation to chain multiple tasks

Another feature of continuations is that you can continue continuations in order to chain tasks together to any length. The pipeline pattern can be implemented with a series of tasks and continuations. You can think of a pipeline as an assembly line in a factory. At the frontend of a pipeline, a producer task generates the data to be operated on, and each of the chained consumer stages operates on or changes the produced data.

In this recipe we will return to our word count example to create a simple three stage pipeline using continuations with `TaskContinuationOptions.OnlyOnRanToCompletion`.

How to do it...

Open up Visual Studio, and let's see how to chain tasks together into a pipeline. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `Continuation7` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   ```

3. Let's start this application by adding try/catch blocks in the `Main` method of the program class. In the `catch` block add some handling for any `AggregateException` raised by the tasks. At the end of the `catch` block, write a message to the console to tell the user we are finished and wait for input to exit.

```
try
{
//Task and continuations go here
}
catch (AggregateException aEx)
{
    foreach (var ex in aEx.InnerExceptions)
    {
        Console.WriteLine("An exception has occured: {0}",
          ex.Message);
    }
}
Console.WriteLine();
Console.WriteLine("Complete. Please hit <Enter> to exit.");
Console.ReadLine();
```

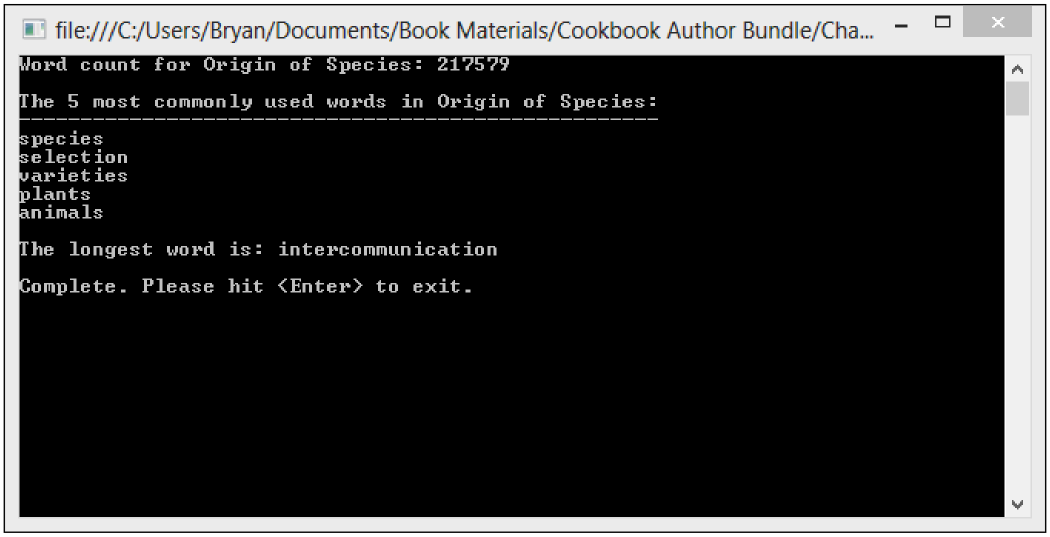
4. Now we need to create a `producer` task that reads in the text of a book, and returns a string array, which the consumer continuations will consume.

```
var producer = Task.Factory.StartNew(() =>
{
    char[] delimiters = { ' ', ',', '.', ';', ':', '-',
      '_', '/', '\u000A' };
    var client = new WebClient();
    const string headerText = "Mozilla/5.0 (compatible;
      MSIE 10.0; Windows NT 6.1; Trident/6.0)";
    client.Headers.Add("user-agent", headerText);
    try
    {
        var words =
client.DownloadString(@"http://www.gutenberg.org/files/2009
  /2009.txt");
        var wordArray = words.Split(delimiters,
          StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Word count for Origin of
          Species: {0}", wordArray.Count());
        Console.WriteLine();
        return wordArray;
    }
    finally
    {
        client.Dispose();
    }
});
```

5. The first consumer will perform a Linq query on the results of the producer to find the five most commonly used words.

```
Task<string[]> consumer1 = producer.ContinueWith(antecedent =>
{
    var wordsByUsage =
      antecedent.Result.Where(word => word.Length > 5)
        .GroupBy(word => word)
        .OrderByDescending(grouping => grouping.Count())
        .Select(grouping => grouping.Key);
    var commonWords = (wordsByUsage.Take(5)).ToArray();
    Console.WriteLine("The 5 most commonly used words in
      Origin of Species:");
    Console.WriteLine("-----------------------------------
      ----------------");
    foreach (var word in commonWords)
    {
        Console.WriteLine(word);
    }
    Console.WriteLine();
    return antecedent.Result;
}, TaskContinuationOptions.OnlyOnRanToCompletion);
The second consumer will perform another Linq query to find the
longest word used.
Task consumer2 = consumer1.ContinueWith(antecedent =>
{
    var longestWord =
       (antecedent.Result.OrderByDescending(w =>
      w.Length)).First();
    Console.WriteLine("The longest word is: {0}",
      longestWord);
}, TaskContinuationOptions.OnlyOnRanToCompletion);
consumer2.Wait();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

The task and continuations we used in this example are pretty much the same as the tasks we have created in other recipes. The primary difference is how we chained them together and the length of the chain. Our antecedent task produces and returns a string array, and then we have a continuation that finds the five most commonly used words, finally we continue the continuation to find the longest word.

Note that we also use `TaskContinuationOptions.OnlyOnRanToCompletion` because we only want the consumers to be scheduled to run when the previous task succeeded. To be a more complete solution, we would want to use `TaskContinuationOptions.OnlyOnFaulted` to set up a continuation for the failure path as well.

Using a continuation to update a UI

A common challenge when developing multithreaded WPF applications is that the UI controls have thread affinity, meaning they can only be updated by the thread that created them. This is usually the main thread of the application.

The TPL, however, offers a clean way to marshal the results from a TPL task to the correct thread for updating the UI. It accomplishes this with the `TaskScheduler.FromCurrentSynchronizationContext` method which creates a `TaskScheduler` associated with the current `SyncronizationContext`.

In this recipe we are going to create a WPF application which will start a task to get the word count of a book. The task will have a continuation that is created in the correct synchronization context by calling `TaskScheduler.FromCurrentSynchronizationContext`. The continuation will perform the UI update.

How to do it...

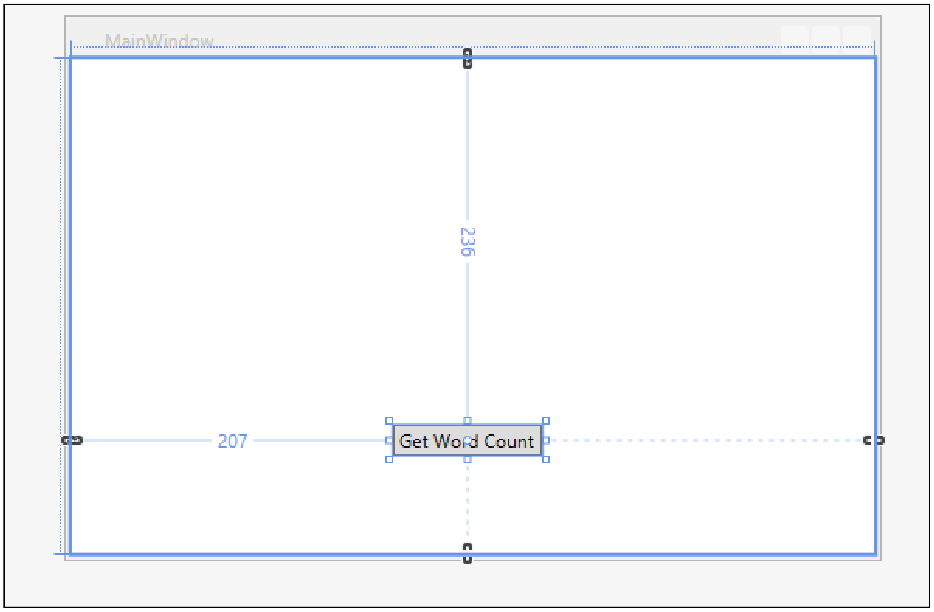
Let's create a WPF application and see how we can use the TPL marshal data to the UI thread.

1. Start a new project using the **WPF Application** project template and assign `Continuation8` as the **Solution name**.
2. Open the `MainWindow.xaml.cs` file and ensure the following `using` directives to the top of your `MainWindow` class:

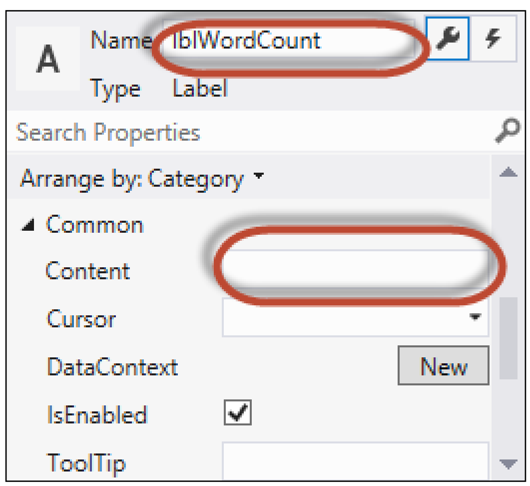
```
using System;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.Windows;
```

3. Go back to `MainWindow.xaml` and replace the XAML with the following code to create the UI layout:

```
<Window x:Class="Continuation8.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Get Word Count"
            HorizontalAlignment="Left"
            Margin="207,236,0,0"
            VerticalAlignment="Top"
            Width="96"
            Click="Button_Click_1"/>
        <Label x:Name="lblWordCount"
            Content=""
            HorizontalAlignment="Left"
            Margin="121,148,0,0"
            VerticalAlignment="Top"
            RenderTransformOrigin="0.094,0.923"
            Width="278"/>

    </Grid>
</Window>
```



4. Now add a `Label` from the toolbox to your window. Change the **Name** property to `lblWordCount` and remove the default value from the **Content** property.

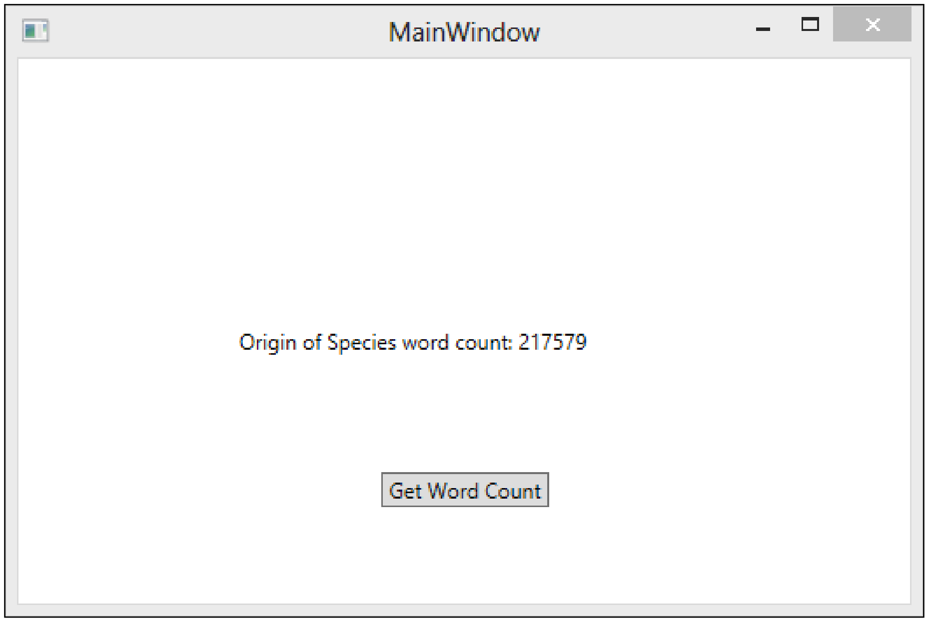


5. OK, now double click on the **Get Word Count** button on your form to open up `MainWindow.xaml.cs` in the `Button_Click_1` event handler. This is where we will create our task and continuation.

6. In the `Button_Click_1` event handler, create a `Task` that reads the content of a book into a string array. The `Task` will return a string array result which will be used in a continuation to display the word count to the UI. The `Task` will be continued with a continuation created with `TaskScheduler.FromCurrentSynchronizationContext` called in the constructor:

```
private void Button_Click_1(object sender, RoutedEventArgs e)
{
    Task.Factory.StartNew(() =>
    {
        char[] delimiters = {' ', ',', '.', ';', ':', '-',
          '_', '/', '\u000A'};
        var client = new WebClient();
        const string headerText = "Mozilla/5.0 (compatible;
          MSIE 10.0; Windows NT 6.1; Trident/6.0)";
        client.Headers.Add("user-agent", headerText);
        try
        {
            var words = client.DownloadString(@"http://www.
gutenberg.org/files/2009
  /2009.txt");
            var wordArray = words.Split(delimiters,
              StringSplitOptions.RemoveEmptyEntries);
            return wordArray;
        }
        finally
        {
            client.Dispose();
        }
    }).ContinueWith(antecedent =>
    {
        lblWordCount.Content = String.Concat("Origin of
          Species word count: ",
          antecedent.Result.Count().ToString());
    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

7. In Visual Studio 2012, press *F5* to run the project. Your application windows should look something as shown in the following screenshot:



How it works...

Tasks run on instances of the `TaskScheduler` class. Two implementations of `TaskScheduler` are included as part of the .NET Framework 4.5. One is the default scheduler, which is integrated with the .NET ThreadPool. The other is the type of `TaskScheduler` returned from the static method `TaskScheduler.FromCurrentSynchronizationContext`.

`SynchronizationContext` provides two methods, `Send` and `Post`, both of which accept a delegate to be executed. `Send` synchronously invokes the delegate, and `Post` asynchronously invokes the delegate.

UI controls should only be accessed by the thread that created them, usually the main UI thread. So, if a thread working in the background wants to update something in the UI, it needs to marshal that data back to the UI thread so that the controls can be accessed safely. In WPF, you would do this with the target thread's `Dispatcher` and corresponding `Invoke/BeginInvoke` methods. With the .NET 4.5 TPL, a new type may be derived from `SynchronizationContext` such that its `Send` method synchronously marshals a delegate to the right thread for execution, and `Post` does the same but asynchronously.

UI frameworks like WPF publish an instance of their `SynchronizationContext` derived class to `SynchronizationContext.Current`. Your code can then get `SynchronizationContext.Current` and use it to marshal work.

`TaskScheduler.FromCurrentSynchronizationContext` creates a `TaskScheduler` that wraps the `SynchronizationContext` returned from `SynchronizationContext.Current`. In doing so, this gives you a `TaskScheduler` that will execute `Tasks` on the current `SynchronizationContext`. This means you can create tasks that are able to access UI controls safely by running them on the right scheduler.

Since we can create a `Task` or continuation with a derived `TaskScheduler`, we can create the continuation with a scheduler that will execute the continuation on the proper context to update the UI.

```
Task.Factory.StartNew(() =>
{
}).ContinueWith(antecedent =>
{
}, TaskScheduler.FromCurrentSynchronizationContext());
```

3
Learning Concurrency with Parallel Loops

In this chapter, we will cover:

- Creating a basic parallel for loop
- Creating a basic parallel foreach loop
- Breaking a parallel loop
- Stopping a parallel loop
- Cancelling a parallel loop
- Handling exceptions in a parallel loop
- Controlling the degree of parallelism in a loop
- Partitioning data in a parallel loop
- Using Thread Local Storage

Introduction

Most developers frequently write sequential code in the form of loops where they are doing something to each of the items in a collection of data. Loops are often an ideal place to introduce parallelism, because most of the time, the items in the collections are not related to each other, and we usually want to perform the same independent operation on all the items in a collection. However, parallelism comes with overhead. The individual loop iterations must perform enough work to justify the overhead of parallelism.

The **.NET 4.5 Parallel Extensions** includes methods that simulate both parallel `For` and parallel `ForEach` loops, and both look very much like the loop syntax you already use for sequential loops. In fact, it is quite easy to change a sequential loop into a parallel loop which can complete faster on a computer with multiple cores.

In this chapter, we will be taking a look at how to use parallel `For` and parallel `ForEach` loops in your programs.

Creating a basic parallel for loop

In this recipe, we will take a look at the syntax of a basic parallel `for` loop and compare its performance against a sequential `for` loop.

For our comparison we will create a console application with two methods. Both methods will loop over a very large array of numbers and use the `Math.Sqrt()` method to calculate the square root of each number in the array. One of our methods will use a sequential `for` loop to process the array, the other will use a parallel `for` loop.

Our program will time both operations and will display the results to the console when both loops finish.

How to do it...

Now, let's open up Visual Studio and create some parallel loops. The steps for creating the parallel `for` loops are as follows:

1. Start a new project using the **C# Console Application** project template and assign `ParallelFor` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:
   ```
   using System;
   using System.Diagnostics;
   using System.Linq;
   using System.Threading.Tasks;
   ```
3. First, let's implement the `Main` method of the program class. We are going to create a `StopWatch` object to perform the timing, create a large array of random numbers, and then call the methods to run the loops.
   ```
   var stopWatch = new Stopwatch();
   var random = new Random();
   var numberList numberArray = Enumerable.Range(1,
     10000000).OrderBy(i => random.Next(1,
     10000000)).ToArray();
   ```

```
stopWatch.Start();
SequentialLoop(numberArraynumberList.ToArray());
stopWatch.Stop();
Console.WriteLine("Time in milliseconds for sequential
  loop: {0}", stopWatch.ElapsedMilliseconds.ToString());

stopWatch.Reset();
stopWatch.Start();
ParallelLoop(numberArraynumberList.ToArray());
stopWatch.Stop();
Console.WriteLine("Time in milliseconds for parallel loop:
  {0}", stopWatch.ElapsedMilliseconds.ToString());

Console.Write("Complete. Press <ENTER> to exit.");
Console.ReadKey();
```

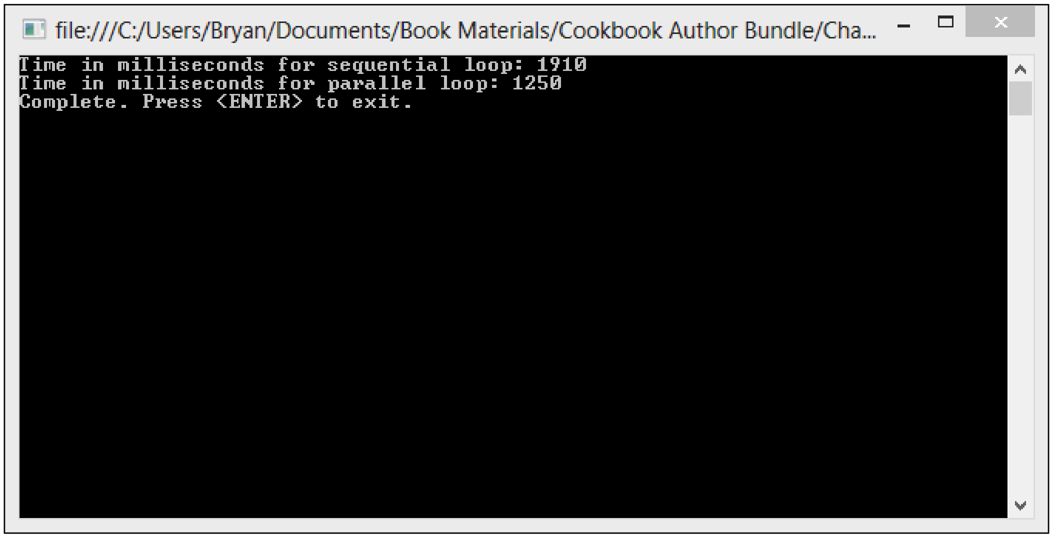
4. Next let's create our `SequentialLoop` method which, as you might have guessed, executes a sequential `for` loop that calculates the square root of each number in the array.

```
private static void SequentialLoop(Int32[] array)
{
    for (var i = 0; i < array.Length; i++)
    {
        var temp = Math.Sqrt(array[i]);
    }
}
```

5. Now we just need to create our `ParallelLoop` method which uses a parallel `for` loop to calculate the square root of each number in the array.

```
private static void ParallelLoop(Int32[] array)
{
    Parallel.For(0, array.Length, i =>
    {
        var temp = Math.Sqrt(array[i]);
    });
}
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

As you can see from the preceding results, on my quad-core machine there were some performance improvements with the parallel loop, but not as much as you might have expected. It is possible that on your machine the sequential loop might have even outperformed the parallel loop. If this is the case, it probably means that the overhead of creating the threads and performing the context switches to execute the threads on the CPU outweighed the benefits of parallelizing the loop. As the chapter continues, we will look at how we can improve the performance of our loops a bit more. For now, we are just focusing on the basics of the parallel for loop syntax.

In this recipe, we used the basic overload of the static `For` method of the `Parallel` class to create our loop. At this level the syntax looks very much like that of a sequential for loop.

```
Parallel.For(int fromInclusive, int toExclusive, Action<int> body );
```

The first two parameters forms the range the loop will iterate over. Note that `from` is an inclusive parameter and `to` is exclusive. So, if our first parameter is 0 and our second parameter is 10, our loop will iterate from 0 to 9.

The third parameter is a delegate of type `Action<int>` which always returns `void`. In this recipe, we use a Lambda expression for the delegate.

```
Parallel.For(0, array.Length, i =>
{
    var temp = Math.Sqrt(array[i]);
});
```

Creating a basic parallel foreach loop

In this recipe we will take a look at the syntax of a basic parallel foreach loop and compare its performance against that of a sequential foreach loop.

For our comparison, like the previous recipe, we will create a Console Application with two methods which both loop over a very large array of numbers and use the `Math.Sqrt()` method to calculate the square root of each number in the array. One of our methods will use a sequential foreach loop to process the array, the other will use a parallel foreach loop.

Our program will time both operations and we will display the results to the console when both loops finish.

How to do it...

Now, let's go to Visual Studio and see how to create parallel for loops. The steps to create parallel ForEach loops are as follows:

1. Start a new project using the **C# Console Application** project template and assign `ParallelForEach` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Diagnostics;
   using System.Linq;
   using System.Threading.Tasks;
   ```

3. Let's start off by putting some code in the `Main` method of the program class to create a `StopWatch` object to perform the timing, create a large array of random numbers, and then call the two methods to run the loops.

   ```
   var stopWatch = new Stopwatch();

   var random = new Random()();
   var numberList = Enumerable.Range(1, 10000000).OrderBy(i =>
     random.Next(1, 10000000));
   ```

```
stopWatch.Start();
SequentialLoop(numberList);
stopWatch.Stop();
Console.WriteLine("Time in milliseconds for sequential
  loop: {0}", stopWatch.ElapsedMilliseconds.ToString());

stopWatch.Reset();
stopWatch.Start();
ParallelForLoop(numberList);
stopWatch.Stop();
Console.WriteLine("Time in milliseconds for parallel loop:
  {0}", stopWatch.ElapsedMilliseconds.ToString());

Console.Write("Complete. Press <ENTER> to exit.");
Console.ReadKey();
```

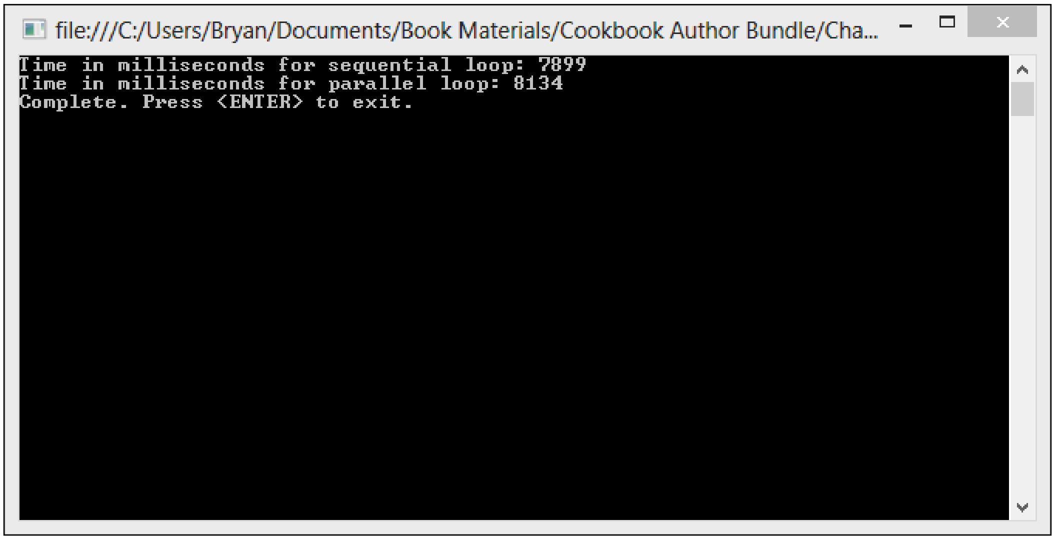
4. Next we need to create our `SequentialLoop` method to execute a sequential `foreach` loop that calculates the square root of each number in the array.

```
private static void SequentialLoop(IEnumerable<int>
  numberList)
{
    foreach (var currentNumber in numberList)
    {
        var temp = Math.Sqrt(currentNumber);
    }
}
```

5. Now let's create our `ParallelLoop` method which uses a parallel `ForEach` loop to calculate the square root of each number in the array.

```
private static void ParallelForLoop(IEnumerable<int>
  numberList)
{
    Parallel.ForEach(numberList, currentNumber =>
    {
        var temp = Math.Sqrt(currentNumber);
    });
}
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

This time the sequential loop outperformed the parallel loop by a bit, at least on my machine.

In this recipe, we used the basic overload of the static ForEach method of the Parallel class to create our loop:

```
ForEach<TSource>(IEnumerable<TSource>, Action<TSource>)
```

Basically, this is just a source that implements IEnumerable<T> and a delegate. In our case we used a Lambda expression for the delegate.

```
Parallel.ForEach(numberList, currentNumber =>
{
    var temp = Math.Sqrt(currentNumber);
});
```

Other than the type of loop we used and the parameter types, the code in this project is very much like that of our parallel for loop recipe.

Breaking a parallel loop

Occasionally when writing loops, we will want to break out of the loop under certain conditions. In a sequential loop we would accomplish this breakout with a C# `break` statement. However, a break statement is only valid when enclosed within an iteration statement like a foreach loop. When we run a parallel foreach, we are not running an iteration statement. It is actually a delegate running in a method.

In this recipe we will we learn how to use a TPL class called `ParallelLoopState` to break out of a parallel ForEach loop. `ParallelLoopState` is a class that allows concurrently running loop bodies to interact with each other. It also provides us with a way to break out of a loop. When the loop breaks or completes, we will check the completion status of our loop using the `ParallelLoopResult` structure.

We are going to create a Console Application to download the contents of a book and split the individual words into a list of strings. We will then loop through the list of strings looking for a specific word. When we find the word we are looking for, we will use `ParallelLoopState` to break out of the loop.

How to do it...

Now let's take a look at how to cancel a parallel loop. The steps to cancel a parallel loop are as follows:

1. Start a new project using the **C# Console Application** project template and assign `BreakALoop` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   ```

3. First, let's add some code to the `Main` method of our program class to use a `WebClient` to download the contents of the book and split the words of the book into a string array.

   ```
   char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_',
     '/', '\u000A' };
    var client = new WebClient();
   const string headerText = "Mozilla/5.0 (compatible; MSIE
     10.0; Windows NT 6.1; Trident/6.0)";
   client.Headers.Add("user-agent", headerText);
   ```

```
var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009
  /2009.txt");
 var wordList = words.Split(delimiters,
  StringSplitOptions.RemoveEmptyEntries).ToList();
```

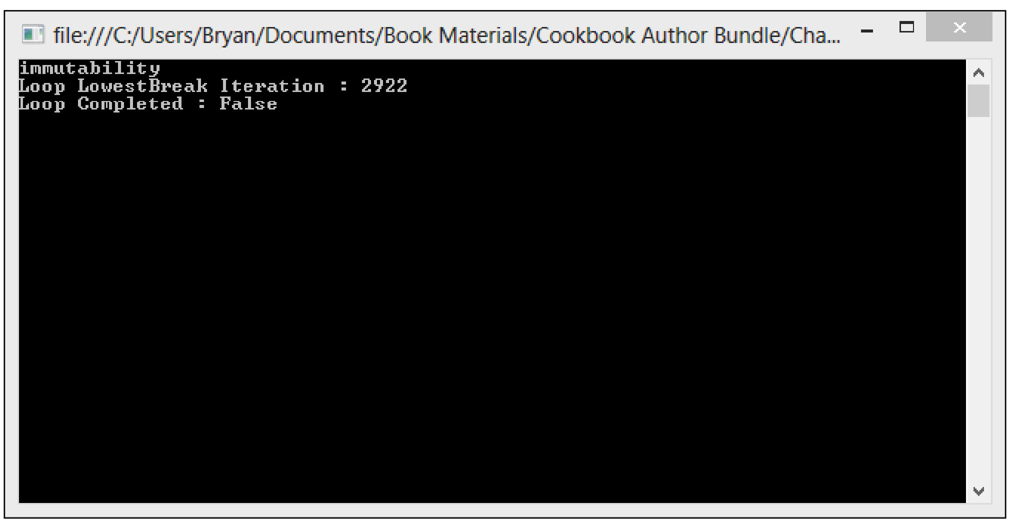
4. Next, let's create our loop. The loop needs to process the list of strings looking for the word "immutability". When we find it, use `ParallelLoopState.Break` to break out of the loop.

```
var loopResult = Parallel.ForEach(wordList, (currentWord,
  loopState) =>
{
    if (currentWord.Equals("immutability"))
    {
        Console.WriteLine(currentWord);
        loopState.Break();
    }
});
```

5. We will finish adding a couple of lines to display the loop iteration we broke on, the completion status of the loop, and wait for the user to exit.

```
Console.WriteLine("Loop LowestBreak Iteration : {0}",
  loopResult.LowestBreakIteration.ToString());
Console.WriteLine("Loop Completed : {0}",
  loopResult.IsCompleted.ToString());
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

In this recipe, we used a different overload of `Parallel.ForEach`. This overload takes an `Action` delegate with the source and a loop state parameter.

```
ForEach<TSource>(IEnumerable<TSource>, Action<TSource,
ParallelLoopState>)
```

In the body of our loop, we used the loop state parameter to cancel the loop.

```
var loopResult = Parallel.ForEach(wordList, (currentWord,
  loopState) =>
{
    if (currentWord.Equals("immutability"))
    {
        Console.WriteLine(currentWord);
        loopState.Break();
    }
});
```

Note that we didn't instantiate the `ParallelLoopState` parameter that we passed into the loop. It was created and provided to us by the `Parallel` class. We just have to change our Lambda expression to indicate that we want to use the loop state parameter.

The `Parallel.ForEach` method returns a `ParallelLoopResult` structure (`var loopResult`). This structure has a couple of very useful properties. One of which is `IsCompleted` that gets the loop completion status. A value of true indicates that all iterations of the loop were executed and the loop didn't receive a request to end prematurely. `LowestBreakIteration` gets the index of the lowest iteration from which `Break` was called.

There is an important difference between breaking from a parallel loop and a sequential loop. When breaking a sequential loop, the break statement will immediately terminate the loop. `ParallelLoopState.Break` has a different behavior. What we are actually doing is signaling that we would like the loop terminated at the system's earliest convenience. The issue is that we are not processing a single element at a time. If we call `ParallelLoopState.Break` in one of our threads, other threads are likely to still be executing. Some code will continue to run for a short time after you request to terminate the loop.

Stopping a parallel loop

When you break from a parallel loop, the application will actually continue to process any elements of the collection that were found prior to the element that was being processed when the `ParallelLoopState.Break` method was called. Sometimes this behavior is not desirable and we want to end the loop immediately, without processing any loop iterations that are currently running.

In this recipe, we will look at how to use the `ParallelLoopState.Stop` method to request that the processing of elements terminate as soon as possible without guaranteeing that any other elements will be processed. We will again be using `WebClient` to download the contents of a book, and splitting the words into a sorted list of strings. We will loop through the list looking for the word "immutability". When we find it, we will stop the loop.

How to do it...

Now, let's start Visual Studio and see how to stop a parallel loop. The steps to stop a parallel loop are as follows:

1. Start a new project using the **C# Console Application** project template and assign `StopALoop` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading.Tasks;
   ```

3. First, let's add some code to the `Main` method of our program class to use a `WebClient` to download the contents of the book, and split the words of the book into a string array.

   ```
   char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_',
     '/','"','(',')', '\u000A' };
   var client = new WebClient();
   const string headerText = "Mozilla/5.0 (compatible; MSIE
     10.0; Windows NT 6.1; Trident/6.0)";
   client.Headers.Add("user-agent", headerText);
   var words = client.DownloadString(@"http://www.gutenberg.org/
   files/2009
     /2009.txt");
   var wordList = words.Split(delimiters, StringSplitOptions.
   RemoveEmptyEntries).Where(word =>
     word.Length > 5).ToList();
   wordList.Sort();
   ```

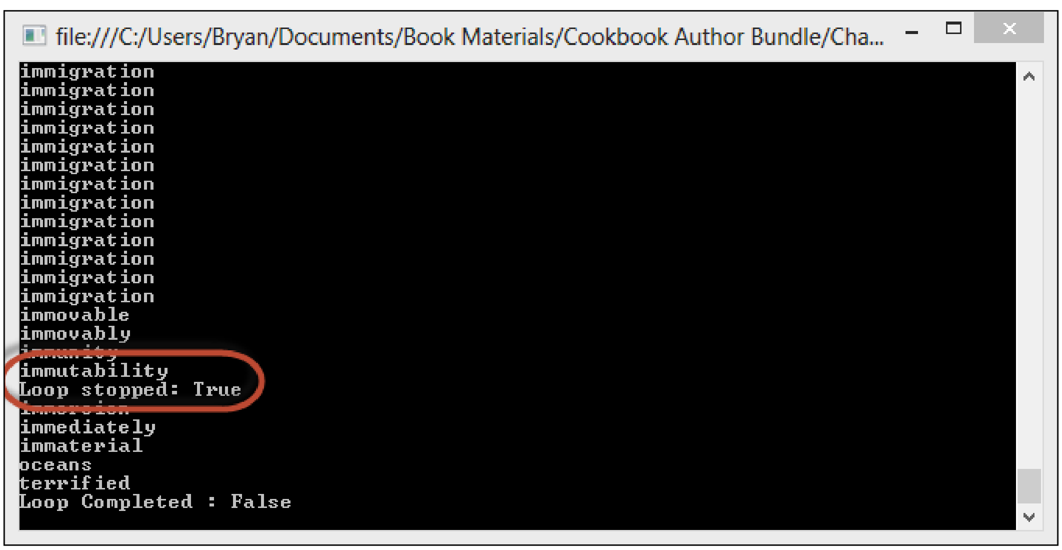
4. Next, let's create our loop. The loop needs to process the list of strings looking for the word "immutability". When we find it, use `ParallelLoopState.Stop` to stop the loop.

```
var loopResult = Parallel.ForEach(wordList, (currentWord,
loopState) =>
{
    if (!currentWord.Equals("immutability"))
        Console.WriteLine(currentWord);
    else
    {
        loopState.Stop();
        Console.WriteLine(currentWord);
        Console.WriteLine("Loop stopped: {0}", loopState.
IsStopped.ToString());
    }
});
```

5. We will finish adding a couple of lines to display the loops' completion status and wait for user's input to exit.

```
Console.WriteLine("Loop Completed : {0}",
  loopResult.IsCompleted.ToString());
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

As in the previous recipe, we used the overload of `Parallel.ForEach` that takes an `Action` delegate with the source and a loop state parameter.

```
ForEach<TSource>(IEnumerable<TSource>, Action<TSource,
ParallelLoopState>)
```

In the body of our loop, we used the loop state parameter to stop the loop. We also used the `ParallelLoopState.IsStopped` property to display the status of our loop.

```
var loopResult = Parallel.ForEach(wordList, (currentWord, loopState)
=>
{
    if (!currentWord.Equals("immutability"))
        Console.WriteLine(currentWord);
    else
    {
        loopState.Stop();
        Console.WriteLine(currentWord);
        Console.WriteLine("Loop stopped: {0}", loopState.IsStopped.
ToString());
    }
});
```

As you can see from the preceding results, the elements of the list that were currently in process when we stopped the loop, still get written to the console. However, `ParallelLoopState.Stop` does stop the loop more quickly than `ParallelLoopState.Break` and is better to use in a situation where you are searching for an element of a condition in the collection.

Both `ParallelLoopState.Break` and `ParallelLoopState.Stop` behave differently from a break statement in a sequential loop. We are asking the application to process more than one thing at a time and we can no longer count on the sequence. It is easy to parallelize loops with the TPL, but it should be approached with caution because we can no longer rely on the order of the results.

Cancelling a parallel loop

As we've seen in previous recipes, to create a task that can be cancelled, you pass in a cancellation token from a `CancellationTokenSource` object. If you then make a call to the `CancellationTokenSource.Cancel` method, the token signals all of the tasks that use it should terminate. The linked tasks detect this signal via the token and stop their activity in a safe manner.

Parallel loops support the same cancellation token mechanism as parallel tasks. In a parallel loop, you supply the `CancellationToken` to the method in the `ParallelOptions` parameter.

This recipe will download the contents of a book and split the words into a list of strings. We will then use a parallel loop to iterate through the words writing each to the console. However, we will create a separate task that sleeps for a few seconds and then calls the `CancellationTokenSource.Cancel` method which will cancel the loop.

How to do it...

Let's create a Console Application in Visual Studio so that we can see how to break a loop. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `BreakALoop` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Linq;
using System.Net;
using System.Threading;
using System.Threading.Tasks;
```

3. First, in the `Main` method of the program class, let's create a `CancellationTokenSource` and then add the `CancellationToken` to a `ParallelOptions` object.

```
var tokenSource = new CancellationTokenSource();
var options = new ParallelOptions
{
    CancellationToken = tokenSource.Token
};
```

4. Next, just below the previous lines, create a simple task that sleeps for a few seconds and then calls the `Cancel` method on the `CancellationTokenSource`.

```
Task.Factory.StartNew(() =>
{
    Thread.Sleep(new TimeSpan(0,0,5));
    tokenSource.Cancel();
});
```

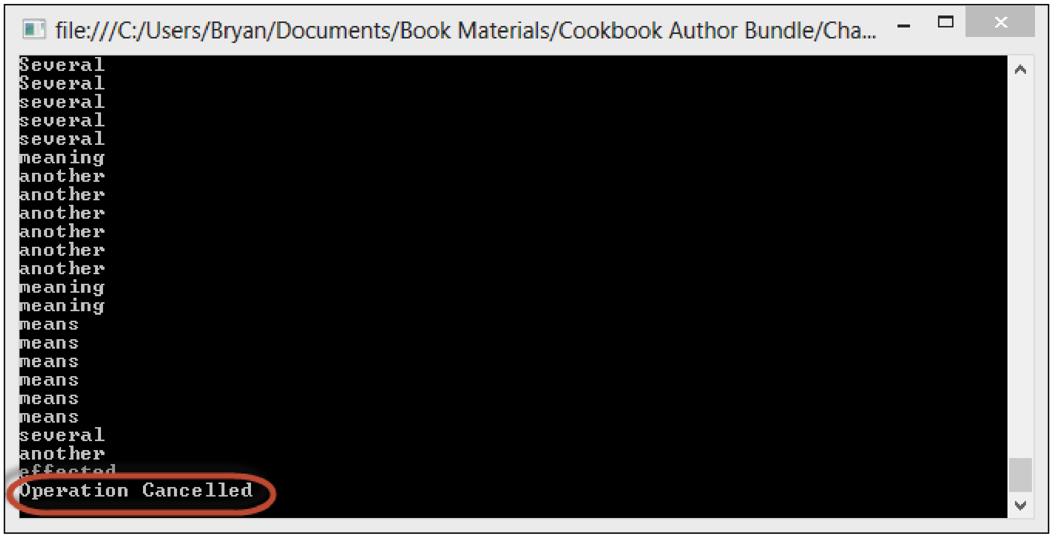
5. Now create a `WebClient` to download the text of a book, and split the words from the book into a list of strings.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_',
  '/', '"', '(', ')', '\u000A' };
var client = new WebClient();
const string headerText = "Mozilla/5.0 (compatible; MSIE
  10.0; Windows NT 6.1; Trident/6.0)";
client.Headers.Add("user-agent", headerText);
 var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009
  /2009.txt");
var wordList = words.Split(delimiters,
  StringSplitOptions.RemoveEmptyEntries).Where(word =>
  word.Length > 5).ToList();
wordList.Sort();
```

6. Finally, let's create a simple parallel `foreach` loop that writes the strings to the console. The loop should be in a try/catch and we should be catching `OperationCancelledException` and `AggregateException`.

```
try
{
    var loopResult = Parallel.ForEach(wordList, options,
(currentWord, loopState) => Console.WriteLine(currentWord));
    Console.WriteLine("Loop Completed : {0}", loopResult.
IsCompleted.ToString());
}
catch (OperationCanceledException)
{
    Console.WriteLine("Operation Cancelled");
}
catch (AggregateException)
{
    Console.WriteLine("Operation Cancelled");
}
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

In this recipe we are using another overload of the `Parallel.ForEach` method that accepts an `IEnumerable` source, a `ParallelOptions` object, and an `Action` delegate.

```
ForEach<TSource>(IEnumerable<TSource>, ParallelOptions,
Action<TSource>)
```

The difference between cancelling a task and cancelling a parallel loop is how we pass in the `CancellationToken`. With a task, a `CancellationToken` is passed directly into the constructor of the task. For a parallel loop, we set the `CancellationToken` property of a `ParallelOptions` object with our `CancellationToken`, and then pass the `ParallelOptions` object into the parallel loop method.

If the token that signals the cancellation is the same token that is set on the `ParallelOptions` instance, then the parallel loop will throw an `OperationCanceledException` on cancellation. If a different token causes cancellation, the loop will throw an `AggregateException` with an `OperationCanceledException` as an `InnerException`. Both should be handled in your `catch` blocks.

Handling exceptions in a parallel loop

When a sequential loop throws an exception, the normal flow of the loop is interrupted. Control will be passed to a `catch` block or, if left unhandled, the exception is passed to the .NET runtime, and the process is aborted.

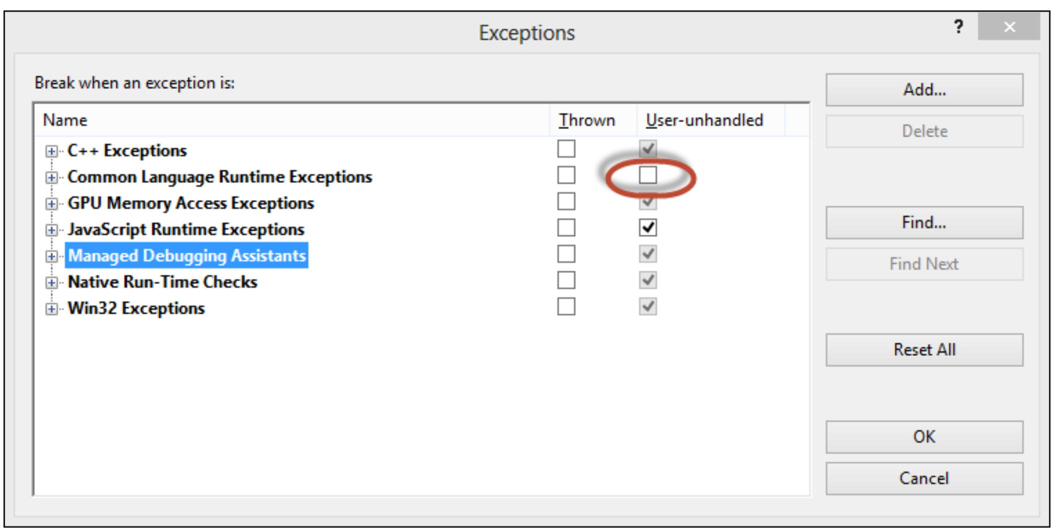
Parallel `For` and `ForEach` loops are similar in that they do not have any special mechanism to handle exceptions that might be thrown. It is up to us to handle any exceptions which might be thrown on one or multiple threads by wrapping all exceptions from the loop in an `AggregateException`.

In this recipe, we will create a simple parallel `For` loop that loops through a range of numbers, writing the values to the console. If the number being processed is higher than a set number, we will throw a new `ArgumentException` which we will then store in a queue and later throw as part of an `AggregateException`.

Getting ready...

For this recipe we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime exception is thrown, and intercepts the exception before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Let's take a look at how to handle exceptions in parallel loops. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `LoopExceptions` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Collections.Concurrent;
   using System.Threading.Tasks;
   ```

3. In the `Main` method of the program class, create a try/catch block. The `catch` block should have some code for looping through `AggregateException.InnerExceptions` and displaying the wrapped exception messages to the console.

   ```
   try
   {
       // Parallel for loop
   }
   catch (AggregateException aggregate)
   {
       // Loop through the exceptions and display to console
       foreach (var ex in aggregate.InnerExceptions)
       {
         Console.WriteLine("An exception was caught:
     {0}",ex.InnerException.Message);
       }
   }

   // Wait for user input before exiting
   Console.ReadLine();
   ```

4. Inside the `try` block, define a variable of type `ConcurrentCollection<Exception>`. This will be the container to hold our exceptions until we are ready to wrap them in an `AggregateException`.

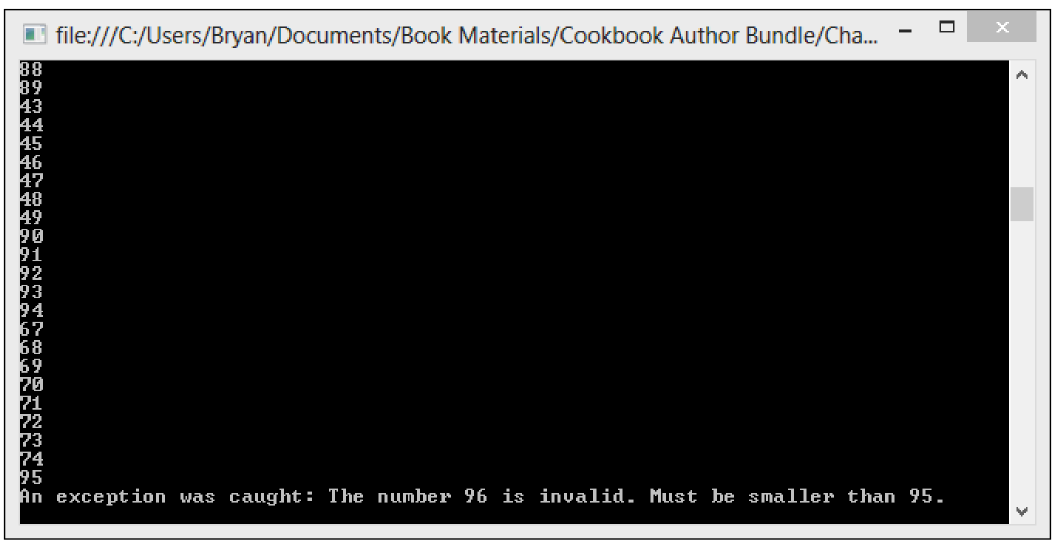
   ```
   var exceptionQueue = new ConcurrentQueue<Exception>();
   ```

5. Finally, let's create a simple parallel `ForEach` loop that loops from 0 to 100. If the loop encounters a number greater than 95, it should throw an `ArgumentException`. The body of the loop needs a try/catch block to catch the argument exception to enqueue it.

   ```
   Parallel.For(0, 100, number =>
   {
   ```

```
    try
    {
        if (number > 95)
        {
          throw new ArgumentException(String.Format("The
            number {0} is invalid. Must be smaller than
            95.",number.ToString()));
        }
        Console.WriteLine(number.ToString());
    }
    catch (Exception ex)
    {
        exceptionQueue.Enqueue(ex);
    }
    if(exceptionQueue.Count > 0)
        throw new AggregateException(exceptionQueue);
});
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

In this recipe, the outer `catch` block handles `AggregateException` which can wrap many individual exception objects. Any one or all the actual threads created by calling `Parallel.For` could throw an `AggregateException`. In our `catch` block, we need to loop through `AggregateException.InnerExceptions` to process individual exceptions that occurred.

```
catch (AggregateException aggregate)
{
    foreach (var ex in aggregate.InnerExceptions)
    {
      Console.WriteLine("An exception was caught:
        {0}",ex.InnerException.Message);
    }
}
```

We have also created a try/catch block in the body of the parallel loop. This `catch` block catches any type of exception thrown in the loop and simply enqueues it in a `ConcurrentQueue<Exception>`. `ConcurrentQueue<T>` is a thread-safe first-in-first-out collection that implements `IEnumerable<T>`. `AggregateException` has a constructor overload that accepts `IEnumerable<Exception>`, so we can wrap our exception collection in `AggregateException` by passing the `ConcurrentCollection` to the constructor.

```
catch (Exception ex)
{
    exceptionQueue.Enqueue(ex);
}
if(exceptionQueue.Count > 0)
    throw new AggregateException(exceptionQueue);
```

Controlling the degree of parallelism in a loop

By default, `Parallel.For` and `Parallel.ForEach` utilize as many threads as the underlying thread scheduler will provide. Usually it is good enough to let the system manage how iterations of a parallel loop are mapped to your computer's cores. Sometimes, however, you might want more control over the maximum number of threads that are used. For example, if you know that an algorithm you are using won't scale beyond a certain number of cores; you might want to limit the cores used in order to avoid wasting cycles.

The number of tasks created by `Parallel.For` and `Parallel.ForEach` is often greater than the number of available cores. However, you can limit the maximum number of tasks used concurrently by specifying the `MaxDegreeOfParallelism` property of a `ParallelOptions` object.

In this recipe, we are going to create a large array of integers. We will then pass this array to a couple of parallel `For` loops. One loop will run with the default degree of parallelism, and the other will be limited to four threads. We will display the time it takes to run each loop to see if there is any performance difference between the two.

How to do it...

Let's take a look at how we can control the degree of parallelism in a parallel loop. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `DegreeOfParallelism` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
```

3. First, in the program class, let's create a method named `DefaultParallelism` that takes an array of `Int32` as a parameter. The method calls `Parallel.For` and loops through the array calculating the square root of each element.

```
private static void DefaultParallelism(Int32[] array)
{
    Parallel.For(0, array.Length, i =>
    {
        var temp = Math.Sqrt(array[i]);
    });
}
```

4. Next, let's create another method named `LimitedParallelism` that takes the same type of parameter. This method will also call `Parallel.For` and loop through the array calculating the square root of each element. The only difference is that this method will also create a `ParallelOptions` object with the `MaxDegreeOfParallelism` property set to `4`.

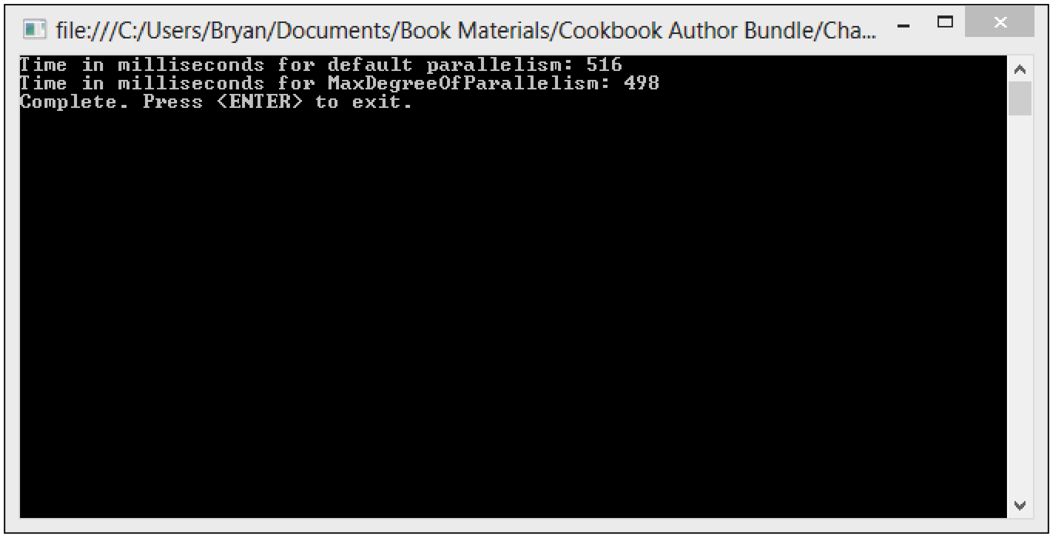
```
private static void LimitedParallelism(Int32[] array)
{
    var options = new ParallelOptions()
```

```
        {
            MaxDegreeOfParallelism = 4
        };

        Parallel.For(0, array.Length, options, i =>
        {
            var temp = Math.Sqrt(array[i]);
        });
    }
```

5. Finally, in the `Main` method, we need to create a large array of `Int32` and initialize the array elements to random numbers. We also need to set up a `StopWatch` object so we can capture some time and call the two methods.

```
static void Main()
{
    var stopWatch = new Stopwatch();

    var random = new Random();
    var numberList numberArray = Enumerable.Range(1,
      1000000).OrderBy(i => random.Next(1,
      1000000)).ToArray();

    stopWatch.Start();
    DefaultParallelism(numberListnumberArray.ToArray());
    stopWatch.Stop();
    Console.WriteLine("Time in milliseconds for default
      parallelism: {0}",
      stopWatch.ElapsedMilliseconds.ToString());

    stopWatch.Reset();
    stopWatch.Start();
    LimitedParallelism(numberList.ToArray());
    stopWatch.Stop();
    Console.WriteLine("Time in milliseconds for
      MaxDegreeOfParallelism: {0}",
      stopWatch.ElapsedMilliseconds.ToString());

    Console.Write("Complete. Press <ENTER> to exit.");
    Console.ReadKey();
}
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

This recipe doesn't have an algorithm that benefits from controlling the degree of parallelism, but in certain long running loop bodies, the ThreadPool's heuristics will be unable to determine the right number of threads to utilize, and could end up injecting many more than is appropriate.

The degree of parallelism is controlled by creating a `ParallelOptions` object and setting the `MaxDegreeOfParallelism` property.

```
var options = new ParallelOptions()
{
        MaxDegreeOfParallelism = 4
};
```

Once created, the `ParallelOptions` object can be passed into one of the many overloads of `Parallel.For` or `Parallel.ForEach` that accept `ParallelOptions`.

```
Parallel.For(0, array.Length, options, i =>
{
    var temp = Math.Sqrt(array[i]);
});
```

Partitioning data in a parallel loop

When creating a `Parallel.For` or `Parallel.ForEach` loops, we are effectively queuing up a delegate of work that will eventually be run on a ThreadPool worker thread. The amount of time taken to create and swap out these delegate payloads can have a very adverse effect on performance, especially when we create loops with small delegate bodies.

There is a default `Partitioner<T>` class that uses a default partitioning algorithm that takes into account the number of cores on your system and other factors, but default portioning may or may not yield the best results.

The .NET 4.5 Parallel Extensions also allows us to create our own custom partitioning chunks so that the workload of a `Parallel.For` or `Parallel.ForEach` is broken up into of a size that we specify in our code. We are effectively creating a custom partitioning algorithm.

In this recipe we are going to create three parallel loops that each iterate over a large array of integers. One of the loops will use no data partitioning, one will use the default partitioner, and one will use a custom partition. We will capture the time it takes each loop to iterate over the array and display the results.

How to do it...

Now, let's see how we can partition data for a parallel loop. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `PartitionData` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Collections.Concurrent;
   using System.Diagnostics;
   using System.Linq;
   using System.Threading.Tasks;
   ```

3. First, in the program class, let's create a method called `NoPartitioning` that takes an array of integers as a parameter. As the name indicates, this method will use no partitioning and will just iterate over the elements in the array calculating the square root of each element.

   ```
   private static void NoPartitioning(Int32[] numbers)
   {
       Parallel.ForEach(numbers, currentNumber =>
       {
          var temp = Math.Sqrt(currentNumber);
   ```

```
    });
}
```

4. Next, we need to create a method called `DefaultPartition`. Like the other method, this one will take an array of integers as its parameter and will iterate over the array calculating the square root of each element in the array. This method will use the `Partitioner.Create` method to create a partition for the data.

```
private static void DefaultPartitioning(Int32[] numbers)
{
    var partitioner = Partitioner.Create(numbers);
    Parallel.ForEach(partitioner, currentNumber =>
    {
        var temp = Math.Sqrt(currentNumber);
    });
}
```

5. Now let's create a method called `CustomPartitioning`. This method will use a different overload of `Partitioner.Create` which allows us to specify the range size we want to use.

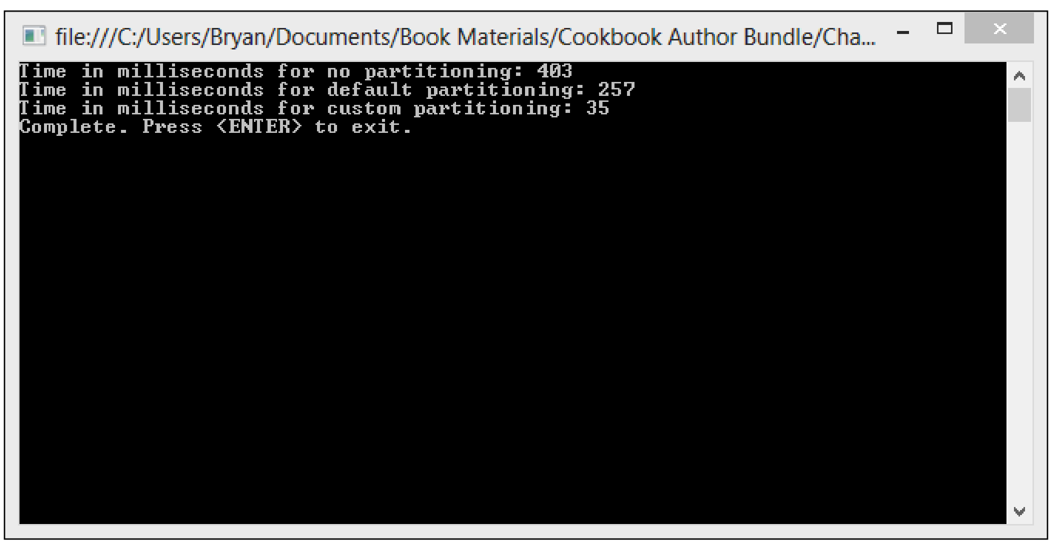
```
private static void CustomPartitioning(Int32[] numbers)
{
    var partitioner = Partitioner.Create(0,
      numbers.Count(), 100000);
    Parallel.ForEach(partitioner, range =>
    {
        for (var index = range.Item1; index < range.Item2;
          index++)
        {
            var temp = Math.Sqrt(numbers[index]);
        }
    });
}
```

6. Finally, in the `Main` method, we need to create a large array of `Int32` and initialize the array elements to random numbers. We also need to set up a `StopWatch` object so we can capture some time and call the three methods.

```
static void Main()
{
    var stopWatch = new Stopwatch();

    var random = new Random();
    var numberArray = Enumerable.Range(1, 10000000).OrderBy(i =>
random.Next(1, 10000000)).ToArray();
```

```
        stopWatch.Start();
        NoPartitioning(numberArray);
        stopWatch.Stop();
        Console.WriteLine("Time in milliseconds for no partitioning:
    {0}", stopWatch.ElapsedMilliseconds.ToString());

        stopWatch.Reset();
        stopWatch.Start();
        DefaultPartitioning(numberArray);
        stopWatch.Stop();
        Console.WriteLine("Time in milliseconds for default
    partitioning: {0}", stopWatch.ElapsedMilliseconds.ToString());

        stopWatch.Reset();
        stopWatch.Start();
        CustomPartitioning(numberArray);
        stopWatch.Stop();
        Console.WriteLine("Time in milliseconds for custom
    partitioning: {0}", stopWatch.ElapsedMilliseconds.ToString());

        Console.Write("Complete. Press <ENTER> to exit.");
        Console.ReadKey();
    }
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

As you can see from the preceding screenshot, there was quite a performance improvement realized from using custom partitioning.

In this recipe, the `NoPartitioning` method iterates through the array performing the `Math.Sqrt` operation on each element with no partitioning of the array at all. The body of the loop is handed to the `Parallel.ForEach` method as a delegate, and the body of a parallel loop is so small that the cost of the delegate invocation on each loop's iteration is very significant. As a result, the performance is not very good.

In the `DefaultPartitioning` method, we used one of the `Create` method overloads of the `Partitioner` class to create an `IEnumerable<T>` of range partitions over the source array. The benefit of doing this is that the delegate invocation cost is incurred only once per range, rather than once per element. As you can see, that resulted in a pretty nice performance improvement.

```
private static void DefaultPartitioning(Int32[] numbers)
{
    var partitioner = Partitioner.Create(numbers);
    Parallel.ForEach(partitioner, currentNumber =>
    {
        var temp = Math.Sqrt(currentNumber);
    });
}
```

In the `CustomPartitioning` method, we use a different overload of `Partitioner.Create` to create a custom partition that chunks a range that we specified.

```
public static OrderablePartitioner<Tuple<int, int>> Create(
    int fromInclusive,
    int toExclusive,
    int rangeSize
)
```

Basically we told the partitioner to create a partition from 0 to `numbers.Count()` with a chunk size of 1,00,000. In other words, we partitioned the array into ten equal parts. The performance improvement was pretty substantial.

The key lesson here is that the overhead of delegate invocation, particularly in loops with small bodies, can be very significant. Consider using either the default partitioning scheme, or a custom chunk partitioner to improve the performance of your parallel loops.

Using Thread Local Storage

Computers are pretty good at counting. Sometimes we need to create loops that accumulate a running count of the occurrence of some piece data. How would we manage something like that when using `Parallel.For` or `Parallel.Foreach` loops? We could have any number of threads counting at the same time.

What we need to accomplish this is **Thread Local Storage**. Thread Local Storage gives us the ability to store and retrieve states in each separate task that is created by a `Parallel.For` or `Parallel.ForEach` loop, and avoid the overhead of synchronizing accesses to a shared state variable.

Thread Local Storage is a programming method that uses static memory local to a thread. This is sometimes needed because normally all threads in a process share the same address space. In other words, data in a static or global variable is normally always located at the same memory location, when referred to from the same process. TLS variables are on the call and are local to threads, because each thread has its own stack.

In this recipe, we are going to see how we can use the thread-local variables to store the value of a word count in each thread created by `Parallel.ForEach` loop. After the loops finish, we will then write the final result only once to a shared variable.

How to do it...

Ok, let's take a look at how we can use the Thread Local Storage in our parallel loops. The steps are as follows:

1. Start a new project using the **C# Console Application** project template and assign `ThreadLocalStorage` as the **Solution name**.
2. Add the following `using` directives to the top of your program class:

   ```
   using System;
   using System.Linq;
   using System.Net;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. First, let's add some code to the `Main` method of the program class to use `WebClient` to download the text of a book, and split the words of the book into an array of strings. Also, we will create an `integer` variable which will hold our word count.

   ```
   char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_',
     '/', '\u000A' };
   var client = new WebClient();
   ```

```
const string headerText = "Mozilla/5.0 (compatible; MSIE
  10.0; Windows NT 6.1; Trident/6.0)";
client.Headers.Add("user-agent", headerText);
var words = client.DownloadString(@"http://www.gutenberg.org/
files/2009
  /2009.txt");
var wordList = words.Split(delimiters,
  StringSplitOptions.RemoveEmptyEntries).ToList();

//word count total
Int32 total = 0;
```

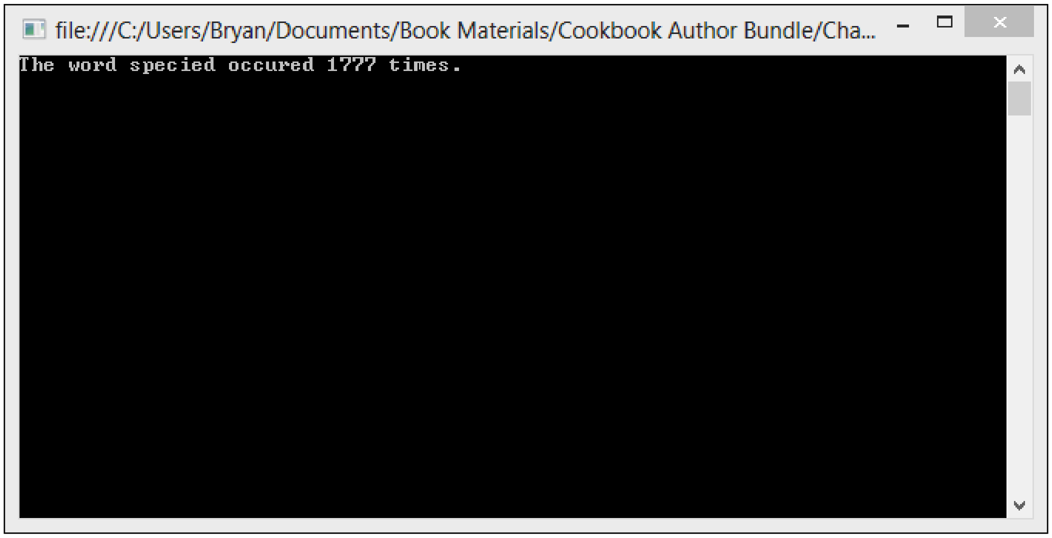
4. Next, let's create a parallel `ForEach` loop that takes an array of `String`, has an `Int32` thread-local variable, and passes its `Int32` result to an `Interlocked.Add` method.

```
Parallel.ForEach<String, Int32>(wordList, () => 0,
    (word, loopstate, count) =>  // method invoked on each
iteration of loop
    {
        if (word.Equals("species"))
        {
            count++; // increment the count
        }
        return count;
    },(result)=>Interlocked.Add(ref total, result));
 // executed when all loops have completed
```

5. Let's finish up by displaying the result and waiting for user input.

```
Console.WriteLine("The word species occured {0}
  times.",total.ToString());
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

Using a thread-local variable in a parallel `ForEach` loop means that we have to use the `Parallel.ForEach` method overload that takes two type parameters and two function parameters. Our first parameter is the type of the elements in our source (`String`). The second parameter specifies the type of our thread-local variable (`Int32`). The third parameter is a `Func<TSource, ParallelLoopState, TLocal, TLocal>` delegate that is invoked on each loop iteration. The fourth parameter is an `Action<T>` delegate that the method will invoke when all loops are finished.

```
ForEach<TSource, TLocal>(IEnumerable<TSource>, Func<TLocal>,
Func<TSource, ParallelLoopState, TLocal, TLocal>, Action<TLocal>)
```

In the body of our loop, the loop passes our input parameters to our function delegate. The parameters are the current element, a `ParallelLoopState` variable and the thread local variable.

```
(word, loopstate, count) =>  // method invoked on each iteration
  of loop
{
if (word.Equals("species"))
{
   count++; // increment the count
```

```
}
return count;
}
```

After the loop completes, we return our thread-local variable and it gets passed to the `Action<T>` delegate where we add it to our shared state variable using `Interlocked.Add()`:

```
(result)=>Interlocked.Add(ref total, result));
```

4

Parallel LINQ

In this chapter, we are going to cover the following recipes:

- Creating a basic parallel query
- Preserving order in parallel LINQ
- Forcing parallel execution
- Limiting parallelism in a query
- Processing query results
- Specifying merge options
- Range projection with parallel LINQ
- Handling exceptions in parallel LINQ
- Cancelling a parallel LINQ query
- Performing reduction operations
- Creating a custom partitioner

Introduction

Language Integrated Query (**LINQ**) offers developers syntax for performing queries on collection of data. Using LINQ you can traverse, filter, sort, and return projected sets of items. When you use LINQ to objects, all of the items in your data collection are processed sequentially by a single thread.

Parallel LINQ is a parallel implementation of LINQ to objects, which can turn your sequential queries into parallel queries, potentially improving performance.

Internally, parallel LINQ uses tasks queued to default `TaskScheduler` to extend the processing of the source collection's items across available processors, so that multiple items are processed concurrently.

In this chapter, we are going to see how parallel LINQ can potentially improve query performance for large collections of items, or for long compute-bound processing of items.

Creating a basic parallel query

In this recipe, we will take a look at creating a basic parallel query by using the `AsParallel` method of the `System.Linq.ParallelEnumerable` class.

We are going to create a `Console` application that initializes a collection of employees, and then queries the employee collection looking for a specific job title.

How to do it...

Now, let's go to Visual Studio and start creating some parallel LINQ queries.

1. Start a new project using the **C# Console Application** project template and assign `SimplePLINQ` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   ```

3. First, we need to create an `Employee` class just below the `Program` Class. Create an `Employee` class definition with `Id`, `Title`, `FirstName`, and `LastName` properties.

   ```
   public class Employee
   {
       public int Id { get; set; }
       public string Title { get; set; }
       public string FirstName { get; set; }
       public string LastName { get; set; }
   }
   ```

4. Now, in the `Main` method of the `Program` class, let's create and initialize an array of `employees`.

   ```
   var employees = newList<Employee>
   {
     new Employee{Id=1, Title="Developer", FirstName="Mark",
   LastName="Smith"},
   ```

```
  new Employee{Id=2, Title="Director", FirstName="Kate",
LastName="Williams"},
  new Employee{Id=3, Title="Manager", FirstName="Karen",
LastName="Davis"},
  new Employee{Id=4, Title="Developer", FirstName="Maria",
LastName="Santos"},
  new Employee{Id=5, Title="Developer", FirstName="Thomas",
LastName="Arnold"},
  new Employee{Id=6, Title="Tester", FirstName="Marcus",
LastName="Gomez"},
  new Employee{I =7, Title="IT Engineer", FirstName="Simon",
LastName="Clark"},
  new Employee{Id=8, Title="Tester", FirstName="Karmen",
LastName="Wright"},
  new Employee{Id=9, Title="Manager", FirstName="William",
LastName="Jacobs"},
  new Employee{Id=10, Title="IT Engineer", FirstName="Sam",
LastName="Orwell"},
  new Employee{Id=11, Title="Developer", FirstName="Tony",
LastName="Meyers"},
  new Employee{Id=12, Title="Developer", FirstName="Karen",
LastName="Smith"},
  new Employee{Id=13, Title="Tester", FirstName="Juan",
LastName="Rodriguez"},
  new Employee{Id=14, Title="Developer", FirstName="Sanjay",
LastName="Bhat"},
  new Employee{Id=15, Title="Manager", FirstName="Abid",
LastName="Naseem"}
};
```

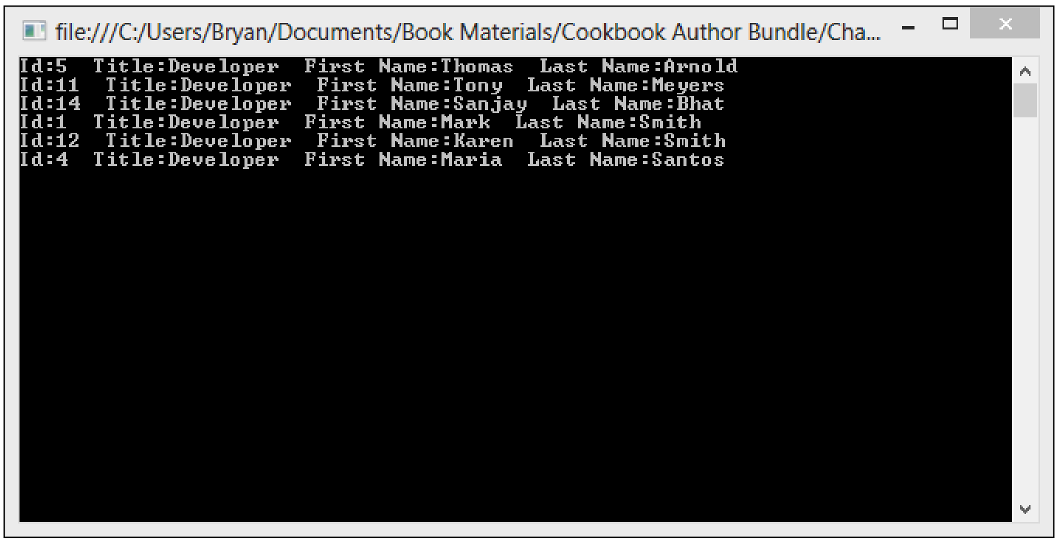
5. Next, we will create a parallel LINQ query that selects all employees where their title is `Developer`.

```
var results = from e in employees.AsParallel()
                where e.Title.Equals("Developer")
                select e;
```

```
Finally, let's loop through the results and display them to the
Console, then wait for user input to exit.
foreach (var employee in results)
{
    Console.WriteLine("Id:{0}  Title:{1}  First Name:{2}  Last
Name:{3}",
```

```
        employee.Id, employee.Title, employee.FirstName, employee.
LastName);
}
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

The small collection of employees we created for this example is too small to benefit from parallelizing the query, but the key thing to notice in the example is the use of the `AsParallel extension` method which binds the query to parallel LINQ, and specifies that the rest of the query should be parallelized if possible.

```
var results = from e in employees.AsParallel()
                where e.Title.Equals("Developer")
                select e;
```

The `System.Linq.ParallelEnumerable` class implements all of the parallel LINQ functionality, and exposes parallel versions of `Select`, `Single`, `Skip`, `OrderBy`, and so on. All of these methods are extension methods that extend `ParallelQuery<TSource>`. The `AsParallel` extension method converts your sequential query based on `IEnumerable<T>` to a parallel query based on `ParallelQuery<T>`.

Preserving order in parallel LINQ

By default, PLINQ does not preserve the order out of a source collection. Because PLINQ processes items in a data collection concurrently using multiple threads, the items are returned unordered. This is by design, because maintaining the original ordering of a sequence adds overhead, and in most cases, that overhead may not be necessary.

However, when you need to preserve order, PLINQ provides a simple way to accomplish it. In this recipe, we are going to create a `Console` application that creates two collections of numbers, performs an ordered query on one collection, and the default unordered query on the other collection, and looks at the results.

How to do it...

Let's open up Visual Studio and see how to preserve order on parallel LINQ queries.

1. Start a new project using the **C# Console Application** project template, and assign `PreserveOrder` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

3. Let's start off by creating a `UnorderedQuery` method just below the `Main` method of the `Program` class. This method will query a large range of integers for numbers that are evenly divisible by 5, and will take the first 10 of those numbers as the result.

```
private static void UnorderedQuery(IEnumerable<int> source)
{
    Console.WriteLine("Unordered results");
    var query = (from numbers in source.AsParallel()
        where numbers%5 == 0
        select numbers).Take(10);

    foreach (var number in query)
        Console.WriteLine(number);
}
```

4. Next we need to create our `OrderedQuery` method which will perform the same query as the previous step, except it will use the `AsOrdered` extension method to preserve the original order.

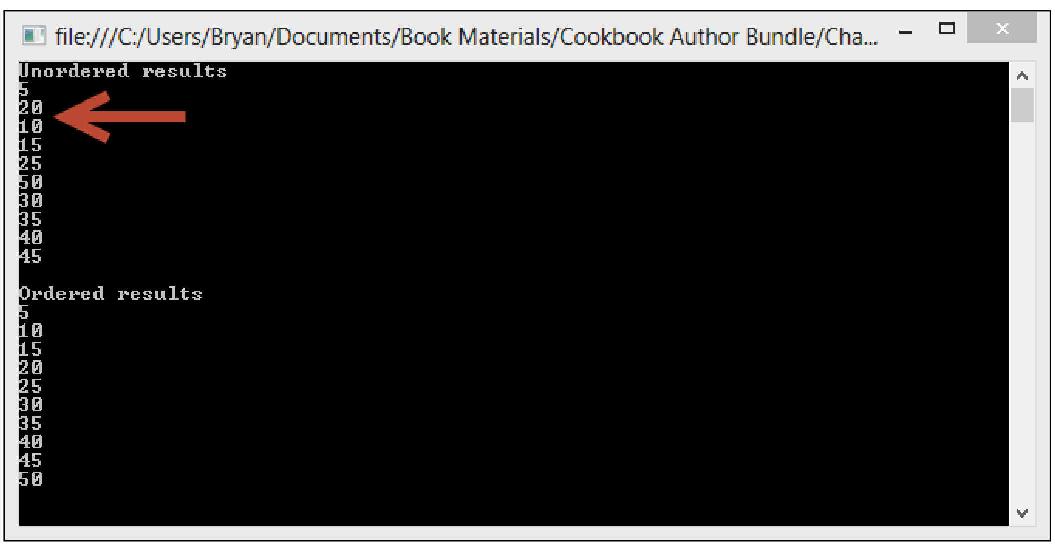
```
private static void OrderedQuery(IEnumerable<int> source)
{
```

```
        Console.WriteLine("Ordered results");
        var query = (from numbers in source.AsParallel().AsOrdered()
            where numbers % 5 == 0
            select numbers).Take(10);

        foreach (var number in query)
            Console.WriteLine(number);
    }
```

5. Now let's add some code to the `Main` method of the `Program` class to create your source lists of numbers and to call each of the methods.

```
var source1 = Enumerable.Range(1, 100000);
UnorderedQuery(source1);

Console.WriteLine();

var source2 = Enumerable.Range(1, 100000);
OrderedQuery(source2);

Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

If you need to have parallel LINQ, preserve the order of items as they are processed, then you can call the `AsOrdered` method of the `ParallelEnumerable` class.

```
var query = (from numbers in source.AsParallel().AsOrdered()
        where numbers % 5 == 0
        select numbers).Take(10);
```

When you call this method, the threads spawned for the query will process the items of a collection in groups, then the groups are merged back together preserving order but hurting performance.

The operators `Distinct`, `Intersect`, `Union`, `Join`, `Except`, `GroupBy`, `GroupJoin`, and `ToLookup` produce unordered operations. If you need to enforce ordering after one of these operations, you just need to call the `AsOrdered` method.

Conversely, the operators `OrderBy`, `OrderByDescending`, `ThenBy`, and `ThenByDescending` produce ordered operations. If you need to go back to unordered processing and improve performance, PLINQ provides a `AsUnordered` method you can call.

Forcing parallel execution

Parallel LINQ looks for opportunities to parallelize a query, but not all queries run faster in parallel. For example, a query that contains a single delegate that does only a little bit of work will usually run faster sequentially, because the overhead of parallelizing outweighs the benefits gained from parallelizing it.

For the most part, parallel LINQ does a really good job of determining what should be parallelized and what should run sequentially, based on its examination of the shape of the query. However, the algorithm it uses is not perfect, and you might need to instruct PLINQ to run your query in parallel.

In this recipe, we will build a `Console` application that creates a query which PLINQ will determine whether it needs to be executed sequentially. We will then force the query to run in parallel using the `WithExecutionMode` method. Finally, we will capture the time it takes for both queries to run and compare the results.

How to do it...

Now, let's see how to force a PLINQ query to execute in parallel.

1. Start a new project using the **C# Console Application** project template, and assign `ForceParallelism` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Diagnostics;
using System.Linq;
using System.Threading;
```

3. First, let's create a method in your `Program` class called `NoForcedParallelism` that creates a PLINQ query with a small delegate of work that PLINQ will evaluate and determine what needs to be executed sequentially.

```
private static void NoForcedParallelism()
{
    Enumerable.Range(0, 1000).AsParallel()
                .Where(x =>
                {
                    Thread.SpinWait(1000000);
                    return true;
                })
                .Select((x, i) => i)
                .ToArray();
}
```

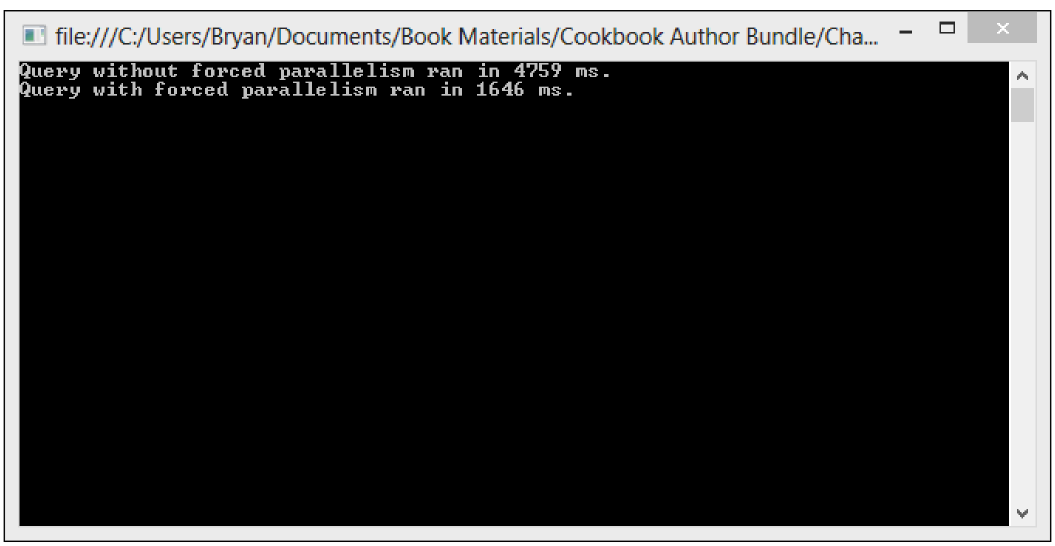
4. Next, let's create a method in the `Program` class called `ForcedParallism` which runs the same query, but forces PLINQ to execute it in parallel by calling the `WithExecutionMode` method and passing it a `ParallelExecutionMode.ForceParallelism` enumeration.

```
private static void ForcedParallelism()
{
    Enumerable.Range(0, 1000).AsParallel()
            .WithExecutionMode(ParallelExecutionMode.
ForceParallelism)
            .Where(x =>
            {
                Thread.SpinWait(1000000);
                return true;
            })
            .Select((x, i) => i)
            .ToArray();
}
```

5. We will finish up by adding some code to the `Main` method to create `stopWatch` to capture the timing of the two methods, then run the methods and compare the results.

```
private static void Main()
{
   var stopWatch = new Stopwatch();
   stopWatch.Start();
   NoForcedParallelism();
   stopWatch.Stop();
   Console.WriteLine("Query with no forced parallelism ran in {0}
ms.",
             stopWatch.ElapsedMilliseconds);
   stopWatch.Reset();
   stopWatch.Start();
   ForcedParallelism();
   stopWatch.Stop();
   Console.WriteLine("Query with forced parallelism ran in {0}
ms.",
             stopWatch.ElapsedMilliseconds);
   Console.ReadLine();
}
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

In this recipe, we can see that PLINQ wrongly decided that the query would run faster sequentially. When we forced the query to run in parallel, the performance improvement was significant.

We instructed PLINQ not to fall back to sequential execution when it detects certain query shapes by calling the `WithExecutionMode` method of `System.Linq.ParallelQuery`, and passing it `ParallelExecutionMode.ForceParallelism` enumeration value.

```
Enumerable.Range(0, 1000).AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .Where(x =>
    {
        Thread.SpinWait(1000000);
        return true;
    })
    .Select((x, i) => i)
    .ToArray();
```

Why did PLINQ determine that the query should be executed sequentially in the first place? It is mainly a factor to determine the shape of the query which has a single delegate of work. It also has to do with this query using the `positional Select` operator. Positional-related operators may require `ForceParallelism` in PLINQ that includes `positional Select`, `positional Where`, `positional SelectMany`, `Take`, `Skip`, `TakeWhile`, and `SkipWhile`.

Limiting parallelism in a query

By default, parallel LINQ will try to take advantage of all of the processor cores offered by your CPU. Usually, this is what you want. However, there could be situations where you want to limit the number of threads used to run queries and keep some cores available for other work.

In this recipe, we are going to create a query that uses the `WithDegreeOfParallelism` method to explicitly set the number of threads that a parallel query uses.

How to do it...

Now, let's see how to limit the degree of parallelism of a query.

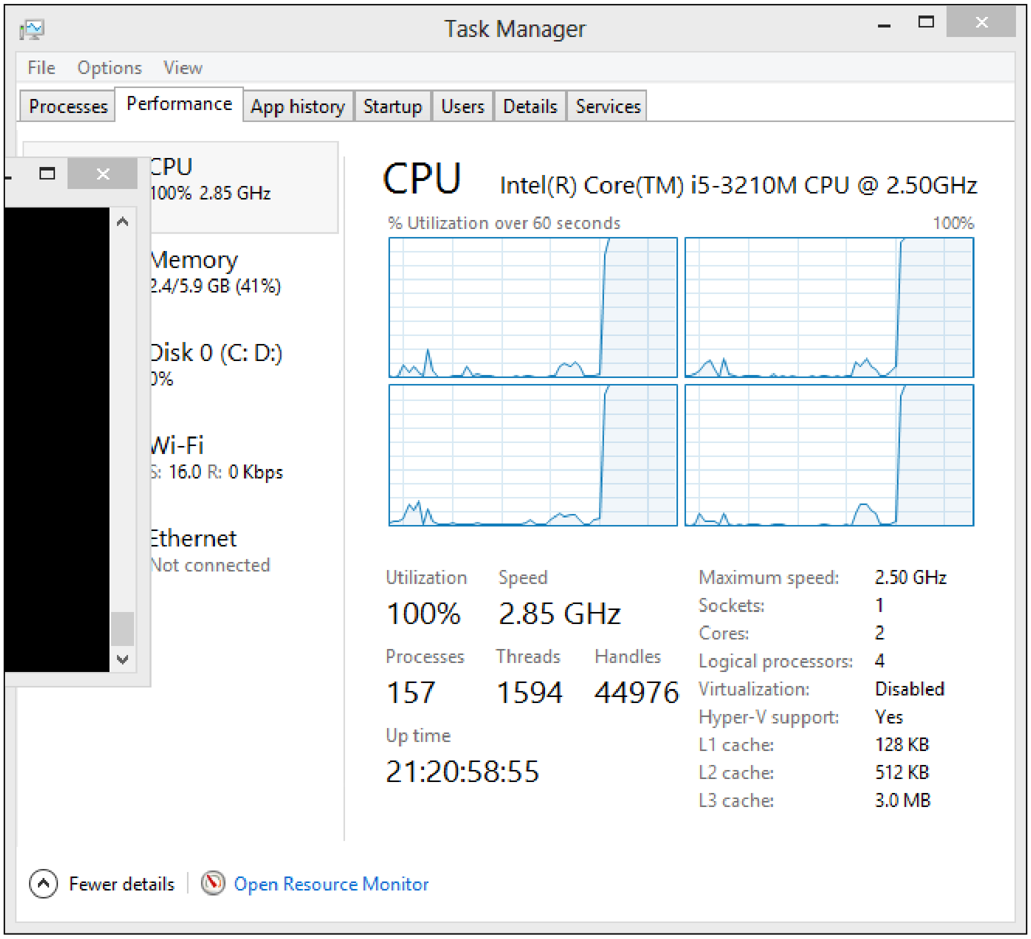
1. Start a new project using the **C# Console Application** project template, and assign `LimitParallelism` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
using System.Threading;
```

3. Let's add a PLINQ query to the `Main` method of the `Program` class that works the processors. For now, we will use the `WithExecutionMode` method to force the query to run in parallel, but will not set a limit on the parallelization.

```
private static void Main()
{

    var result = Enumerable.Range(0, 10000).AsParallel()
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .Where(x =>
      {
                Thread.SpinWait(1000000);
                return true;
            })
        .Select((x, i) => i)
        .ToArray();

        foreach (var number in result)
            Console.WriteLine("Result: {0}",number);

        Console.ReadLine();
}
```

4. If you have a multi-core processor on your machine, start up the **Task Manager**, click on the **Performance** tab, and watch CPU usage. In Visual Studio 2012, press *F5* to run the project. You should see processor usage as shown in the following screenshot:

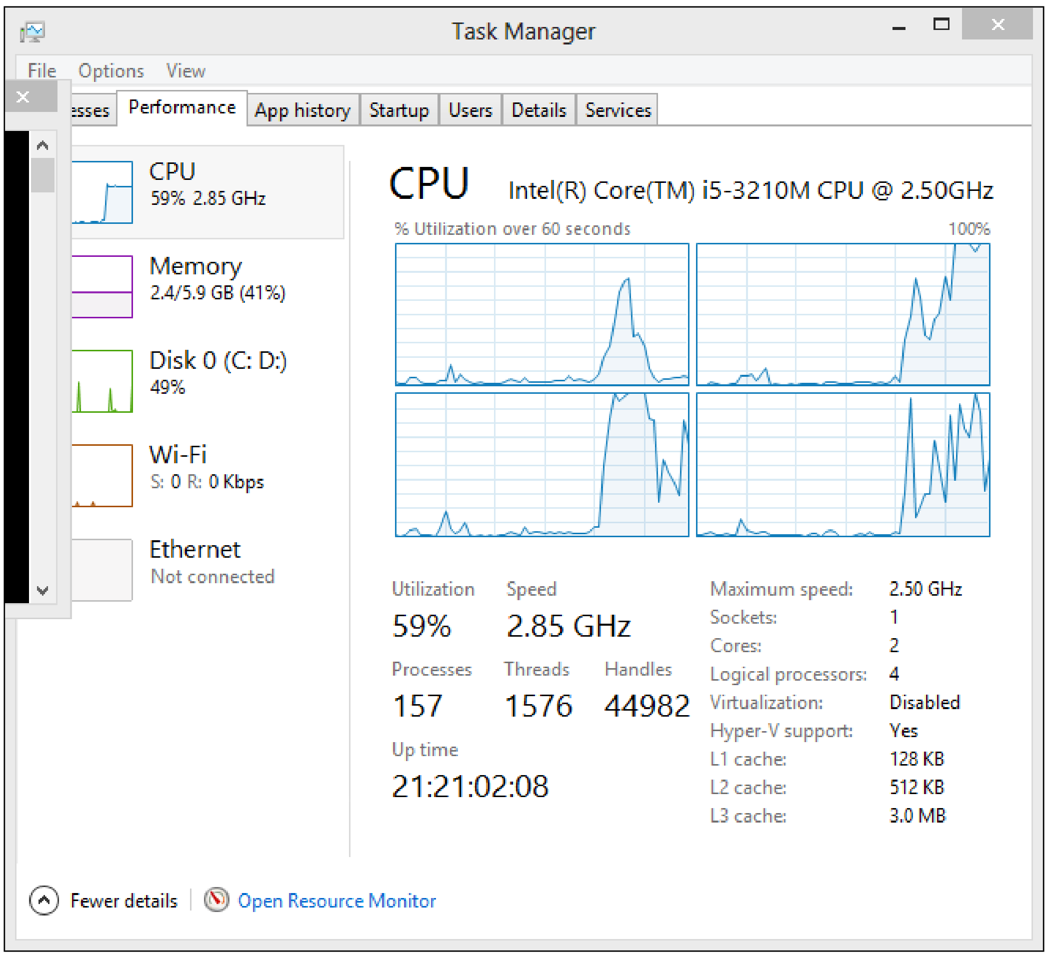


5. Now let's edit the PLINQ query to call the `WithDegreeOfParallelism` method to limit the number of processor cores used. You might want to change the value you pass into the method to be a number that is relevant to the number of processor cores available to you. You can specify a number greater than the number of processor cores available on your machine, but this will likely lead to more context switching.

```
var result = Enumerable.Range(0, 10000).AsParallel()
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .WithDegreeOfParallelism(2)
```

```
.Where(x =>
    {
        Thread.SpinWait(1000000);
        return true;
    })
.Select((x, i) => i)
.ToArray();
```

6. Now start up the **Task Manager** again, click on the **Performance** tab, and watch CPU usage. In Visual Studio 2012, press *F5* to run the project. You should see reduced processor usage. Note that the threads created will not necessarily spend all of their time on a single core, but the overall usage will go down.



How it works...

The `WithDegreeOfParallelism` method probably isn't something that you will use very often. You might want to use it in situations where you need to leave some CPU time available to perform other tasks. You could also pass a number that is greater than the number of cores on your machine, in cases where the query will be performing synchronous I/O, because the threads will be blocking.

Setting the degree of parallelism is simply a matter of calling the method and passing in the number of threads you want PLINQ to use.

```
var result = Enumerable.Range(0, 10000).AsParallel()
        .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
        .WithDegreeOfParallelism(2)
        …
        .Select((x, i) => i)
        .ToArray();
```

The default value of the `WithDegreeOfParallism` method is the processor count of your machine.

Processing query results

One of the nice features of parallel LINQ is that it collates the results from a parallelized query into a single output sequence. Often though, all your program does with the query's output data is run a function over each element using a `foreach` loop or similar. In such cases, particularly in cases where you don't care about the order in which elements are processed, you can improve performance by using the ParallelEnumerable's `ForAll` method to process the results in parallel.

In this recipe, we will perform a query on a range of numbers, and then use `ParallelForAll` to iterate over the results in parallel, calculating the square of each number.

How to do it...

Now, let's open up Visual Studio and see how to process the results of a parallel query.

1. Start a new project using the **C# Console Application** project template, and assign `ProcessResults` as the **Solution name**.

2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

3. First, in the `Main` method of the `Program` class, let's create a concurrent collection of integers to hold the result of our calculation, and generate a range of numbers for the query.

```
var result = new ConcurrentBag<Int32>();
var source = Enumerable.Range(1, 100000);
```

4. Next, just below the previous lines, create a PLINQ query that queries the first 100 numbers that are evenly divisible by 5 out of the source range.

```
var query = (from numbers in source.AsParallel()
        where numbers % 5 == 0
        select numbers).Take(100);
```

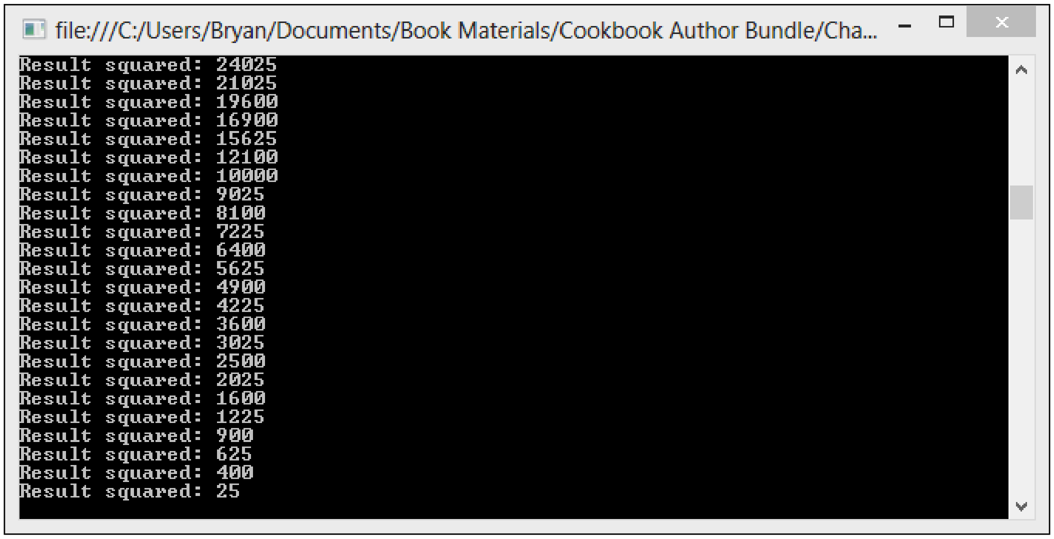
5. Now let's call the `ParallelEnumerable.ForAll` method to process our query results in parallel. We are just going to calculate the square of each number and add the result to our collection.

```
query.ForAll(r =>
    {
        result.Add(r * r);
    });
```

6. Finally, let's loop through the collection and print the results to `Console`.

```
foreach (var value in result)
{
     Console.WriteLine("Result squared: {0}", value );
}
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

In this recipe, we are using ParallelEnumerable.ForAll<TSource> to iterate through and process the results of our query in parallel. The ForAll method runs a delegate over every output element of a ParallelQuery.

```
query.ForAll(r =>
    {
        result.Add(r * r);
    });
```

The ForAll method hooks right into parallel LINQ's internals, bypassing the steps of collating, and enumerating the results which can save considerable processing time.

You might wonder why we used the ForAll method to calculate the square of our results and add them to the collection; just to use a sequential foreach loop to write the results to Console. Besides the obvious answer that this is just a simple example, you wouldn't want to write to Console inside a ForAll method because .NET serializes all access to Console, and would force the whole thing to run sequentially.

Specifying merge options

When parallel LINQ executes a query, it partitions the source data and assigns each partition to a separate thread. If the results are consumed by a single thread, such as a `foreach` loop, then the results from each partition must be merged back into one result set. The kind of merge that is performed depends on the operators used in the query. For operators that produce ordered results, the results from all the threads are completely buffered before being merged back together. Your application's consuming thread might have to wait for a while before seeing the final result. If you don't care about order, or want to use a different buffering scheme to improve results, your query can use the `WithMergeOptions` extension method to provide a hint to PLINQ about how you would like the results to be buffered.

In this recipe, we will query some numbers out of a range, and loop through the results using a couple of different buffering options, and observe the effects.

How to do it...

Now, let's take a look at how to specify the merge option of a PLINQ query.

1. Start a new project using the **C# Console Application** project template, and assign `MergeOptions` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
using System.Threading;
```

3. Let's start by creating our source number range in the `Main` method of the `Program` class.

```
var numbers = ParallelEnumerable.Range(0, 1000);
```

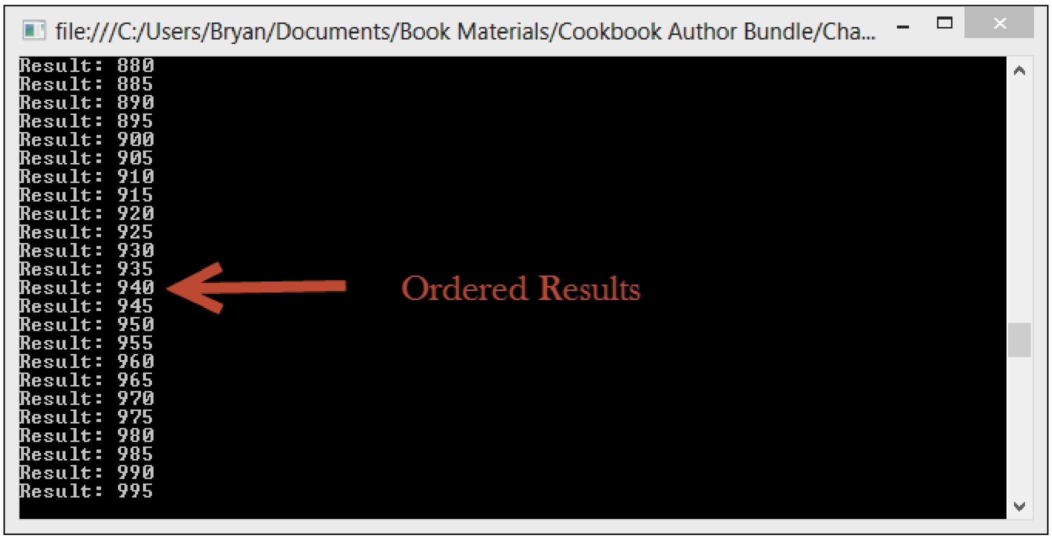
4. Now let's create a PLINQ query to select the numbers in the range that are evenly divisible by 5, and use the `WithMergeOption` method to specify fully buffered merging of the results.

```
var result = numbers.AsParallel()
        .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
        .Where(number => number % 5 == 0);
```

5. Next, let's use a `foreach` loop to iterate through the results and print them to `Console`.

```
foreach (var number in result)
{
    Console.WriteLine("Result: {0}",number);
}
Console.ReadLine();
```

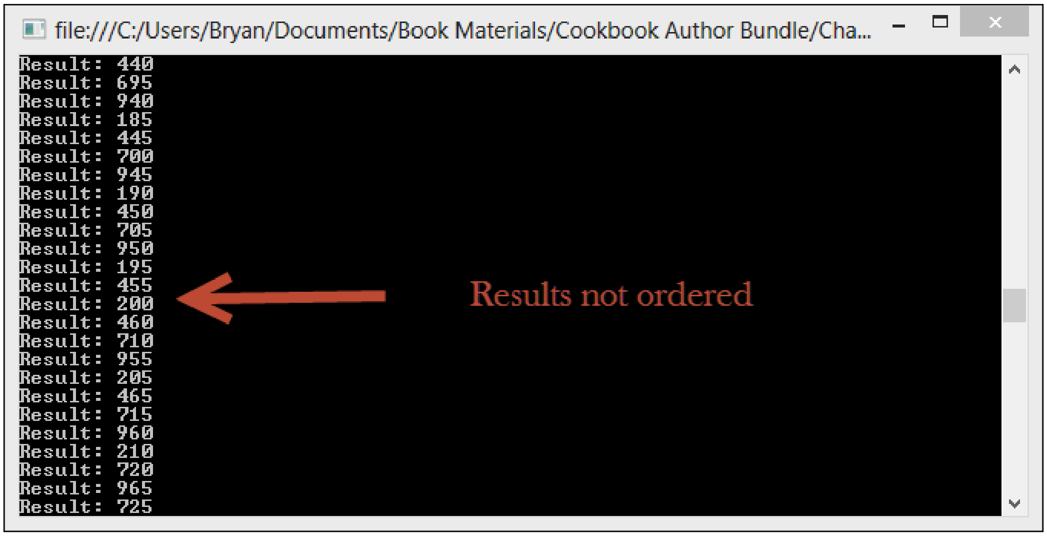
6. In Visual Studio 2012, press *F5* to run the project. Notice the **Ordered Results** shown in the following screenshot:



7. Now let's use the `WithMergeOption` method to specify no buffering for the results.

```
var result = numbers.AsParallel()
        .WithMergeOptions(ParallelMergeOptions.NotBuffered)
        .Where(number => number % 5 == 0);
```

8. In Visual Studio 2012, press *F5* to run the project. Notice the unordered results in the following screenshot:



How it works...

The `WithMergeOptions` method takes the `ParallelMergeOptions` enumeration as a parameter. You can specify one of the following options for how the query output is yielded and when the results can be consumed:

- **Not Buffered**: Each processed element of the query is returned from each thread as soon as it is produced. If the `AsOrdered` operator is present, ordering is preserved, but if `AsOrdered` is not present, then the results are yielded as soon as they are available. This option yields the fastest results.
- **Auto Buffered**: The elements are collected into a buffer and periodically yielded to the consuming thread. This is a middle ground approach to buffering.
- **Fully Buffered**: All of the elements are collected in a buffer before any elements are yielded to the consuming thread.

When you use `WithMergeOptions`, you are giving PLINQ a hint about the buffering scheme you want it to use. If a particular query cannot support the requested option, your request will be ignored.

Range projection with parallel LINQ

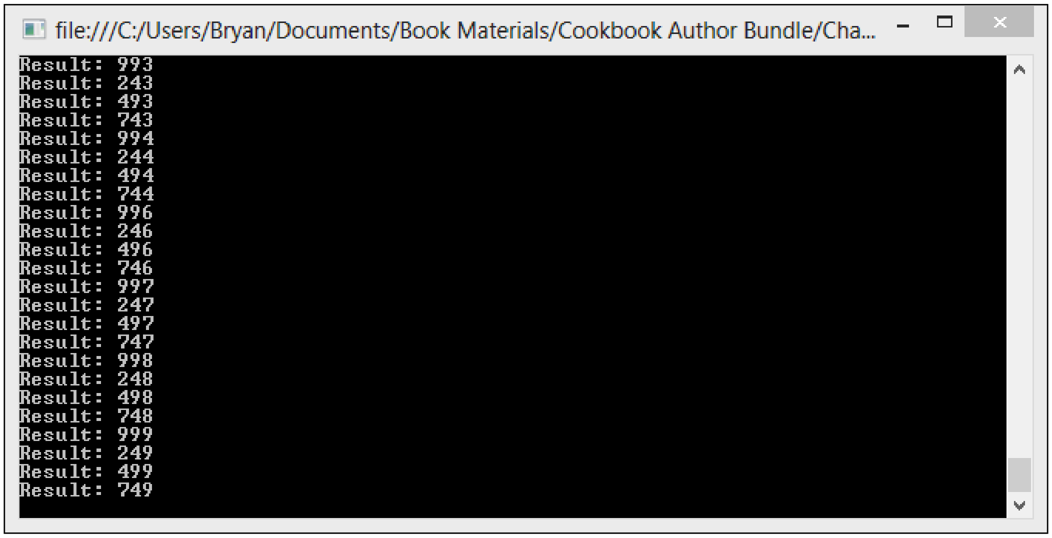
While using sequential LINQ, it is very common to use range projection to obtain a range of values. Parallel LINQ provides us with a way to do this too. If you need to generate a very large range of numbers which do not necessarily need to be in sequence, you can use the `Range` method of `ParallelEnumerable` to create the sequence.

In this short recipe, we will use `ParallelEnumerable.Range` to generate some numbers over a very large range.

How to do it...

Now, let's go to Visual Studio and see how to use parallel LINQ to generate a range of numbers.

1. Start a new project using the **C# Console Application** project template, and assign `RangeProjection` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:
```
using System;
using System.Linq;
```
3. First, in the `Main` method, let's use `ParallelEnumerable.Range` to create a range of numbers between 1 and 1000 that are divisible by 5.
```
var numbers = ParallelEnumerable.Range(1, 1000)
        .Where(x => x % 5 != 0)
        .Select(i => i);
```
4. Now let's just loop through the results to display them to the `Console` and wait for user input before exiting.
```
foreach (var number in numbers)
{
    Console.WriteLine("Result: {0}",number);
}
Console.ReadLine();
```
5. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

The code for this recipe is fairly easy to understand. One important point to note is that the implicit cast to `ParallelQuery<int>` creates a parallel execution instead of a sequential one, and there is no particular ordering of the numbers in the result.

Of course, order can be preserved by calling `AsOrdered` on the query, but if ordering is important to you, just generate the range using sequential LINQ projection, and avoid the overhead of parallel execution.

Handling exceptions in parallel LINQ

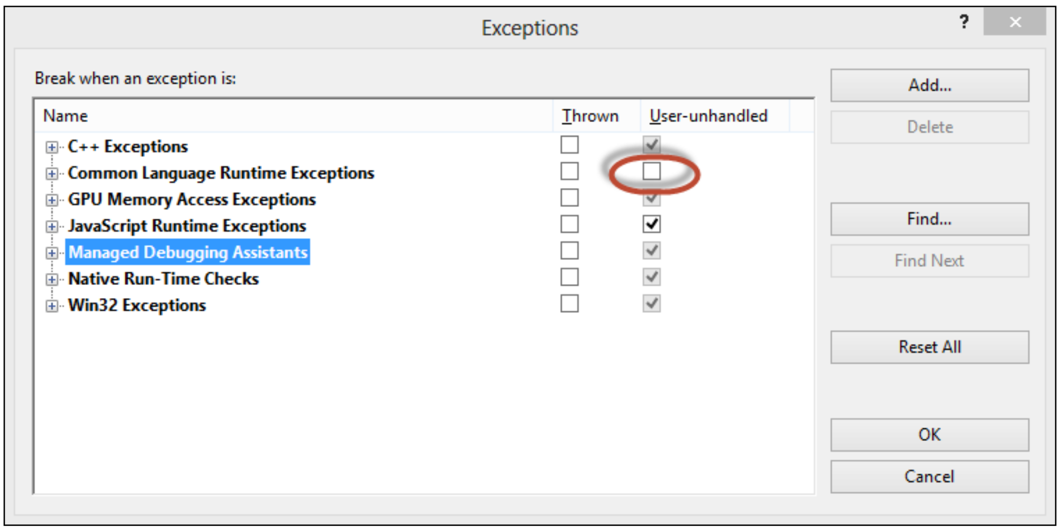
Handling exceptions in parallel LINQ is not much different from handling exceptions in tasks, continuation, or anywhere else in your parallel code. You need to use `try/catch` and make sure to catch `AggregateException`. With parallel LINQ, the really important part is to use the try/catch around where you enumerate or use your results.

In this recipe, we are going to create a simple parallel LINQ query that returns a list of employees and throws `InvalidOperationException`, which we will handle when we iterate through the results.

Getting ready...

For this recipe, we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime `Exception` is thrown, and intercepts the `Exception` before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and click on **Exception**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Now, let's take a look at how to handle exceptions in a parallel LINQ query.

1. Start a new project using the **C# Console Application** project template, and assign `HandleExceptions` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
```

3. First, let's create an `Employee` class. Add the following class definition to your `Program.cs` file, just below the `Program` class:

```
public class Employee
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

4. Now, in the `Main` method of the `Program` class, let's create and initialize an array of `employees`.

```
var employees = new[]
{
  new Employee{Id=1, Title="Developer", FirstName="Mark",
LastName="Smith"},
  new Employee{Id=2, Title="Director", FirstName="Kate",
LastName="Williams"},
  new Employee{Id=3, Title="Manager", FirstName="Karen",
LastName="Davis"},
  new Employee{Id=4, Title="Developer", FirstName="Maria",
LastName="Santos"},
  new Employee{Id=5, Title="Developer", FirstName="Thomas",
LastName="Arnold"},
  new Employee{Id=6, Title="Tester", FirstName="Marcus",
LastName="Gomez"},
  new Employee{I =7, Title="IT Engineer", FirstName="Simon",
LastName="Clark"},
  new Employee{Id=8, Title="Tester", FirstName="Karmen",
LastName="Wright"},
  new Employee{Id=9, Title="Manager", FirstName="William",
LastName="Jacobs"},
  new Employee{Id=10, Title="IT Engineer", FirstName="Sam",
LastName="Orwell"},
  new Employee{Id=11, Title="Developer", FirstName="Tony",
LastName="Meyers"},
  new Employee{Id=12, Title="Developer", FirstName="Karen",
LastName="Smith"},
  new Employee{Id=13, Title="Tester", FirstName="Juan",
LastName="Rodriguez"},
  new Employee{Id=14, Title="Developer", FirstName="Sanjay",
LastName="Bhat"},
  new Employee{Id=15, Title="Manager", FirstName="Abid",
LastName="Naseem"}
  new Employee{Id=16, Title="Developer",FirstName="Kevin",
LastName="Strong"}
};
```

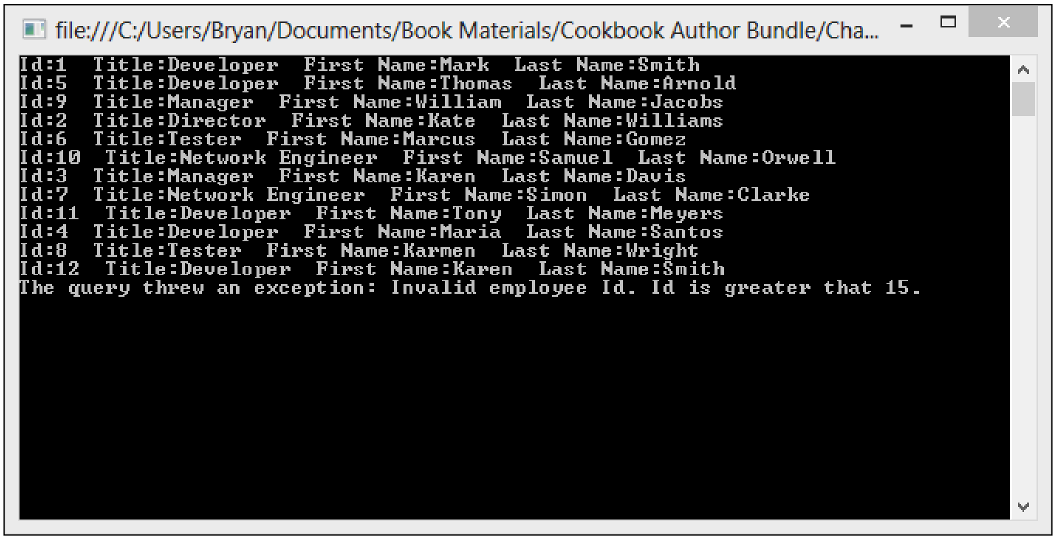
5. Next, create a PLINQ query that selects all employees and throws `InvalidOperationException` when it encounters an `employee Id` that is greater than `15`.

```
var results = employees.AsParallel()
    .Select(employee =>
    {
    if (employee.Id > 15)
    throw new InvalidOperationException("Invalid employee. Id > 
15.");
    return employee;
    });
```

6. Finally, let's create a `try/catch` block. In the `try` block, create a `foreach` loop to iterate through the results. In the `catch` block, you need to handle `AggregateException` and display the `Exception` to `Console`. Finish up by waiting for user input before exiting the program.

```
try
{
    foreach (var employee in results)
    {
        Console.WriteLine("Id:{0}  Title:{1}  First Name:{2}  Last 
Name:{3}",
            employee.Id, employee.Title, employee.FirstName, 
employee.LastName);
    }
}
catch (AggregateException aggregateException)
{
    foreach (var exception in aggregateException.InnerExceptions)
    {
        Console.WriteLine("The query threw an exception: {0}", 
exception.Message);
    }
}
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

As you can see, the steps for handling exceptions that occur in a parallel LINQ query are very consistent with handling exceptions that occur in other parts of our parallel code. Any `Exception` that occurs will be added to the `InnerExceptions` collection of an `AggregateException`. So, we need to be ready to catch `AggregateException` and examine the individual `Exception` items in the `InnerExceptions` collection.

In this recipe, we just placed a `try/catch` block around the loop that iterates through our results, and handled `AggregateException` in the `catch` block.

```
try
{
    foreach (var employee in results)
    {
        ...
    }
}
catch (AggregateException aggregateException)
{
    foreach (var exception in aggregateException.InnerExceptions)
    {
        Console.WriteLine("The query threw an exception: {0}",
exception.Message);
    }
}
```

Cancelling a parallel LINQ query

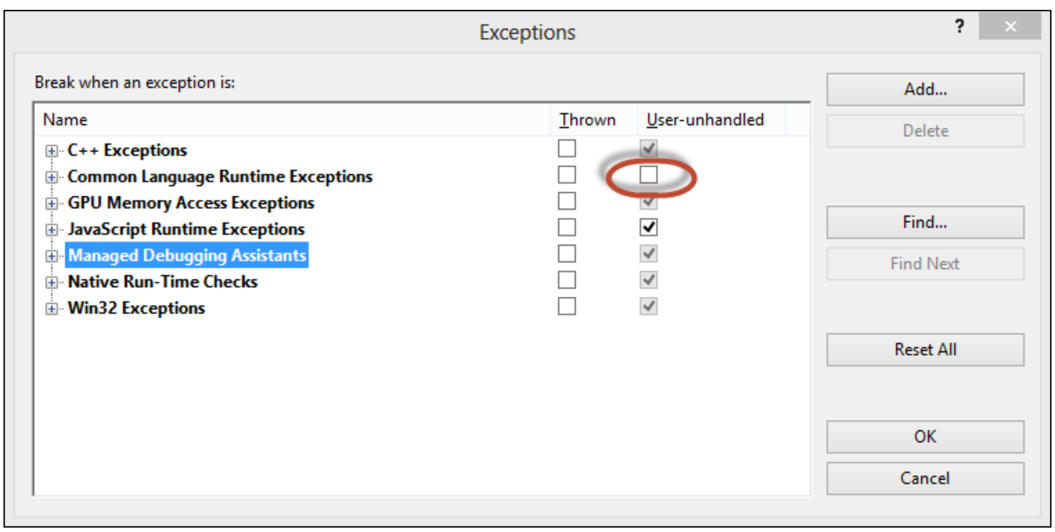
Like tasks and continuations, parallel LINQ queries are cancelled by using `CancellationToken` which you obtain from `CancellationTokenSource`. A minor difference in cancelling a parallel LINQ query is in how you register a `CancellationToken` with the `WithCancellation(tokenSource.Token)` extension method.

In this recipe, we are going to create a cancellable parallel query that selects the square of numbers from a large range of random numbers. We are then going to create a separate task to cancel the query from.

Getting ready...

For this recipe, we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime `Exception` is thrown and intercepts the `Exception` before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exception**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Now, let's take a look at how to cancel a parallel LINQ query.

1. Start a new project using the **C# Console Application** project template and assign `CancelQuery` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. First, let's add some code to the `Main` method of the `Program` class to create our `CancellationTokenSource` object. We also need to initialize a large range of random numbers that will be the source for the query.

```
var tokenSource = new CancellationTokenSource();
var random = new Random();
var numberList = ParallelEnumerable.Range(1, 100000).OrderBy(i =>
random.Next());
```

4. Next, let's create a parallel LINQ query that uses the `WithCancellation` extension method to accept a cancellation token and uses the `Math.Pow` method to select the square of each number.

```
var results = numberList
    .AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .WithCancellation(tokenSource.Token)
    .Select(number => Math.Pow(number, 2));
```

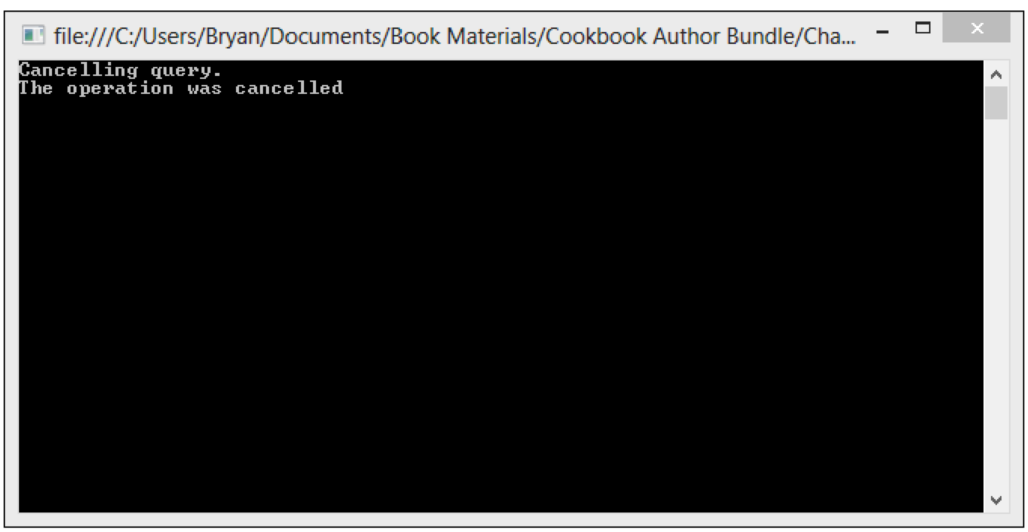
5. Now let's create a task that will sleep for 1 second to give the query time to start, and then it will call the `CancellationTokenSource.Cancel` method to cancel the token.

```
Task.Factory.StartNew(() =>
{
    Thread.Sleep(1000);
    tokenSource.Cancel();
    Console.WriteLine("Cancelling query.");
});
```

6. Finally, let's create a `try` block and a couple of `catch` blocks. The `try` block will just contain a `foreach` loop, to loop through the results of the query. The first `catch` block needs to catch `OperationCancelledException` and display a message to `Console`. The second `catch` block needs to catch `AggregateException` and display all `InnerException` messages to the `Console`. Lastly, let's wait for user input before exiting.

```
try
{
    foreach (var number in results)
    {
        Console.WriteLine("Result: {0}", number);
    }
}
catch (OperationCanceledException)
{
    Console.WriteLine("The operation was cancelled");
}
catch (AggregateException aggregateException)
{
    foreach (var exception in aggregateException.InnerExceptions)
    {
        Console.WriteLine("Handled exception: {0}",exception.
Message);
    }
}
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output as shown in the following screenshot:



How it works...

We used two separate `catch` blocks in this recipe because the parallel LINQ framework does not roll `OperationCanceledExceptions` into `AggregateException`; the `OperationCanceledException` must be handled in a separate `catch` block or it will be left unhandled.

If you have one or more delegates, throw `OperationCanceledException` by using `CancellationToken`, but don't throw any other exception. Then, parallel LINQ will just throw a single `OperationCanceledException` rather than `System.AggregateException`. However, if a delegate throws `OperationCanceledException` and another delegate throws another `Exception` type, then both exceptions will be rolled into `AggregateException`. So, whenever you create a PLINQ query using the `WithCancellation` extension method, it is recommended that you create two `catch` blocks: one for `OperationCancelledException` and one for `AggregateException`.

Performing reduction operations

Like sequential LINQ, parallel LINQ provides many common aggregation operations such as sum, average, min, and max. It is pretty easy to perform aggregate operations by using one of LINQs extension methods.

Sometimes, however, we need to perform a custom aggregation of our source data, either because we need to perform a calculation that isn't provided in one of the standard aggregation extension methods, or because we need to apply custom logic to the calculation.

For such cases, parallel LINQ provides us with a `aggregate` method which can apply a custom accumulator function in parallel over a sequence of data.

In this recipe, we are going to create a custom aggregation operation that calculates the average of a large range of numbers.

How to do it...

Now, let's see how to perform custom aggregation with PLINQ.

1. Start a new project using the **C# Console Application** project template, and assign `Average` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
```

3. First, let's add some code to the `Main` method to create a range of random numbers.

```
var random = new Random();
var numbers = ParallelEnumerable.Range(1, 1000).OrderBy(i =>
random.Next()).ToArray();
```

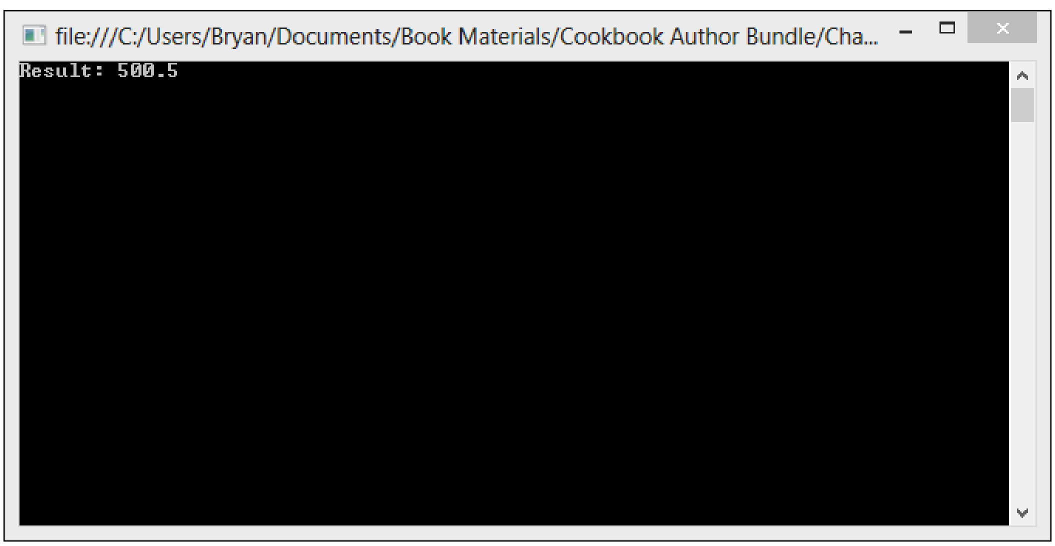
4. Now let's create a PLINQ query that calls the `Aggregate` extension method of `ParallelEnumerable` passing delegate to calculate the average to the `intermediateReduceFunc`, `finalReduceFunc`, and `resultSelector` parameters. Display the results to `Console` and wait for user input before exiting.

```
var result = numbers.AsParallel().Aggregate(() => new double[2],
    (accumulator, elem) => { accumulator[0] += elem;
accumulator[1]++; return accumulator; },
    (accumulator1, accumulator2) => { accumulator1[0] +=
accumulator2[0]; accumulator1[1] += accumulator2[1]; return
accumulator1; },
    accumulator => accumulator[0] / accumulator[1]);
```

5. Finish up by displaying the results to the `Console` and waiting for the user input before exiting.

```
Console.WriteLine("Result: {0}",result);
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

An aggregation operation is an operation that iterates over a sequence of input data elements, maintaining an accumulator that contains an intermediate result. At each step, a reduction function takes the current element and accumulator value as inputs, and yields a value that will overwrite the accumulator. The final accumulator value is the result of the computation.

The `ParallelEnumerable` class provides several overloads of the `aggregate` extension method. We are using the following overload:

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
  this ParallelQuery<TSource> source,
  Func<TAccumulate> seedFactory,
  Func<TAccumulate, TSource, TAccumulate> updateAccumulatorFunc,
  Func<TAccumulate, TAccumulate, TAccumulate> combineAccumulatorsFunc,
  Func<TAccumulate, TResult> resultSelector
)
```

The `seedFactory` function returns the initial accumulator value.
The `updateAccumularorFunc` parameter is an `accumulator` function to be invoked on each element in a partition. The `combineAccumulatorsFunc` parameter is an `accumulator` function to be invoked on the yielded accumulator result from each partition. Finally, the `resultSelector` parameter is a function to transform the final accumulator value into the result value.

We have provided our own delegate for each of these function parameters.

```
numbers.AsParallel().Aggregate(() => new double[2], //Seed factory
    (accumulator, elem) =>  //Update accumulator function
  {
    accumulator[0] += elem;
    accumulator[1]++;
    return accumulator;
  },
    (accumulator1, accumulator2) =>  //Combine accumulator function
  {
    accumulator1[0] += accumulator2[0];
    accumulator1[1] += accumulator2[1];
    return accumulator1;
  },
    accumulator => accumulator[0] / accumulator[1]); //Result selector
function
```

Creating a custom partitioner

To parallelize an operation on a data source, one of the essential steps is to partition the source into multiple sections that can be accessed concurrently by multiple threads. Parallel LINQ provides default partitioners that work quite well for most parallel queries. However, for more advanced scenarios, you can also create your own partitioner.

For the recipe, we will create a custom static partitioner which will split our data source into a variable number of partitioned chunks. The exact number of partitions will be specified by TPL itself, and will be made available to our custom partitioner by overriding the `Partitioner<T>` method. We will then test the performance of a query that uses default partitioning against the performance of a query using our custom partitioner.

How to do it...

Let's take a look at how to partition data for a parallel query.

1. Start a new project using the **C# Console Application** project template, and assign `CustomPartitioner` as the **Solution name**.
2. Add a new class the project and name it `CustomPartitioner.cs`.
3. Add the following `using` directives to the top of your `CustomPartitioner` class:

```
using System.Collections.Concurrent;
using System.Collections.Generic;
```

4. Apply a generic type parameter to the `CustomPartitioner` class, and declare `Partitioner<T>` as its base class. Optionally, mark the class visibility as internal.

```
internal class CustomPartitioner<T> : Partitioner<T>
{

}
```

5. Create a private source field of type array of `T` and initialize the source data with the `Class` constructor.

```
internal class CustomPartitioner<T> : Partitioner<T>
{
  private readonly T[] _source;

    // Class constructor. Initializes source data to array
  public CustomPartitioner(T[] sourceData)
```

```
    {
        _source = sourceData;
    }
}
```

6. Override the `SupportsDynamicPartitions` property of the base class to return `false`. This partitioner can only allocate partitions statically.

```
public override bool SupportsDynamicPartitions
{
    get
    {
        return false;
    }
}
```

7. Add a `GetItems` method that returns `IEnumerator<T>` for the items in the source.

```
internal IEnumerator<T> GetItems(int start, int end)
{
    for (var index = start; index < end; index++)
        yield return _source[index];
}
```

8. Finish up the `CustomPartitioner` class by overriding the `GetPartitions` method of the base class. This method will return `List<IEnumerable<T>>` which is a list of our partitioned data.

```
public override IList<IEnumerator<T>> GetPartitions(int
partitionCount)
{
    IList<IEnumerator<T>> partitionedData = new
List<IEnumerator<T>>();
    var items = _source.Length / partitionCount;
    for (var index = 0; index < partitionCount - 1; index++)
    {
        partitionedData.Add(GetItems(index * items, (index + 1) *
items));
    }
    partitionedData.Add(GetItems((partitionCount - 1) * items, _
source.Length));
    return partitionedData;
}
```

That's it for the `CustomPartitioner` class. Now let's go to the `Program` class:

1. Add the following `using` directives to the top of the class:

```
using System;
using System.Diagnostics;
using System.Linq;
```

2. In the `Main` method of the `Program` class add some code to create a large array of random numbers that we will use for the source of our query. Also, create a `stopWatch` object, which we will use to capture our performance numbers.

```
var stopWatch = new Stopwatch();
var random = new Random();
var source = Enumerable.Range(1, 10000000).OrderBy(i => random.
Next()).ToArray();
```

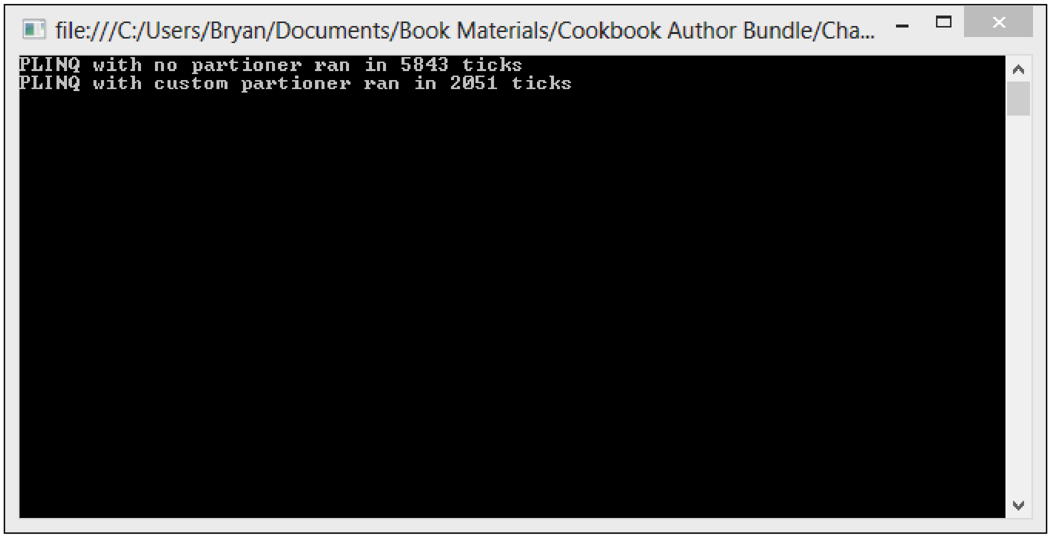
3. Next, let's start `stopWatch` and run a query against the source data to select the square of each number. This query uses default partitioning.

```
stopWatch.Start();
source.AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .Select(item => Math.Sqrt(item));
    stopWatch.Stop();
    Console.WriteLine("PLINQ with no partioner ran in {0} ticks",
stopWatch.ElapsedTicks );
```

4. Finally, let's reset `stopWatch`, run the query with our custom partitioner, display the results, and wait for user input before exiting.

```
var partitioner = new CustomPartitioner<int>(source);

stopWatch.Reset();
stopWatch.Start();

partitioner.AsParallel()
    .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
    .Select(item => Math.Sqrt(item));

stopWatch.Stop();
Console.WriteLine("PLINQ with custom partioner ran in {0} ticks",
stopWatch.ElapsedTicks);

Console.ReadLine();
```

5. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

In some cases, it might be worthwhile to create your own partitioner, but for the most part, the default partitioning works pretty well.

To create a basic custom partitioner, derive a class from `Partitioner<TSource>` which is located in `System.Collections.Concurrent`, and override a couple of virtual methods and a virtual property getter.

We provided an override for the `SupportsDynamicPartitions` property to indicate that our simple custom partitioner only supports static partitions by returning `true`.

```
public override bool SupportsDynamicPartitions
{
    get
    {
        return false;
    }
}
```

If we had indicated that this partitioner does support dynamic partitions, we would want to provide an override for the `GetDynamicPartitions` method, which can be called instead of `GetPartitions` for dynamic partitioners.

In our case, we just had to provide an override for `GetPartitions`. This method returns `IList(IEnumerator(TSource))` which represents our actual partitioned data.

```
public override IList<IEnumerator<T>> GetPartitions(int
partitionCount)
{
    IList<IEnumerator<T>> partitionedData = new
List<IEnumerator<T>>();
    var items = _source.Length / partitionCount;
    for (var index = 0; index < partitionCount - 1; index++)
    {
        partitionedData.Add(GetItems(index * items, (index + 1) *
items));
    }
    partitionedData.Add(GetItems((partitionCount - 1) * items, _
source.Length));
    return partitionedData;
}
```

Lastly, we provided a `GetItems` method which is a helper that `GetPartitions` uses to get an enumerator for the items in our source data.

```
internal IEnumerator<T> GetItems(int start, int end)
{
    for (var index = start; index < end; index++)
        yield return _source[index];
}
```

5
Concurrent Collections

In this chapter, we are going to cover the following recipes:

- Adding and removing items to `BlockingCollection`
- Iterating a `BlockingCollection` with `GetConsumingEnumerable`
- Performing LIFO operations with `ConcurrentStack`
- Thread safe data lookups with `ConcurrentDictionary`
- Cancelling an operation in a concurrent collection
- Working with multiple producers and consumers
- Creating object pool with `ConcurrentStack`
- Adding blocking and bounding with `IProducerConsumerCollection`
- Using multiple concurrent collections to create a pipeline

Introduction

Although `System.Collections` namespace offers a wide range of collections; the only thing which limits our use of them in a multi-threaded or parallel environment is that they are not thread safe. A non thread safe collection could lead to race conditions, which is a condition that occurs when two or more threads can access shared data and try to change it at the same time, producing unexpected errors.

Concurrent collections in .NET Framework 4.5 allow the developers to create type safe as well as thread safe collections. These collection classes form an essential part of the parallel programming feature and are available under the namespace `System.Collections.Concurrent`.

Adding and removing items to BlockingCollection

`BlockingCollection<T>` is a thread safe collection class that provides blocking and bounding functionality. Bounding means that you can set the maximum capacity of a collection, which enables you to control the maximum size of the collection in the memory.

Multiple threads can add items to a collection concurrently, but if the collection reaches capacity, the producing threads will block until items are removed. Multiple consumers can remove items from the collection concurrently. If the collection becomes empty, consumption will block until more items are produced and added to the collection.

In this recipe, we will take a look at the basics of adding items to, and removing items from `BlockingCollection`.

We are going to create a `Console` application that initializes a range of integers and creates a parallel task to add the numbers to a blocking collection. Another parallel task will be created to remove items from the collection.

How to do it...

Let's start Visual Studio and see how to add and remove items with `BlockingCollection`.

1. Start a new project using the **C# Console Application** project template and assign `BlockingCollection` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. First, in the `Main` method of the `Program` class, let's create our range of input data and our blocking collection.

```
var data = Enumerable.Range(0, 100);
var numbers = new BlockingCollection<int>(100);
```

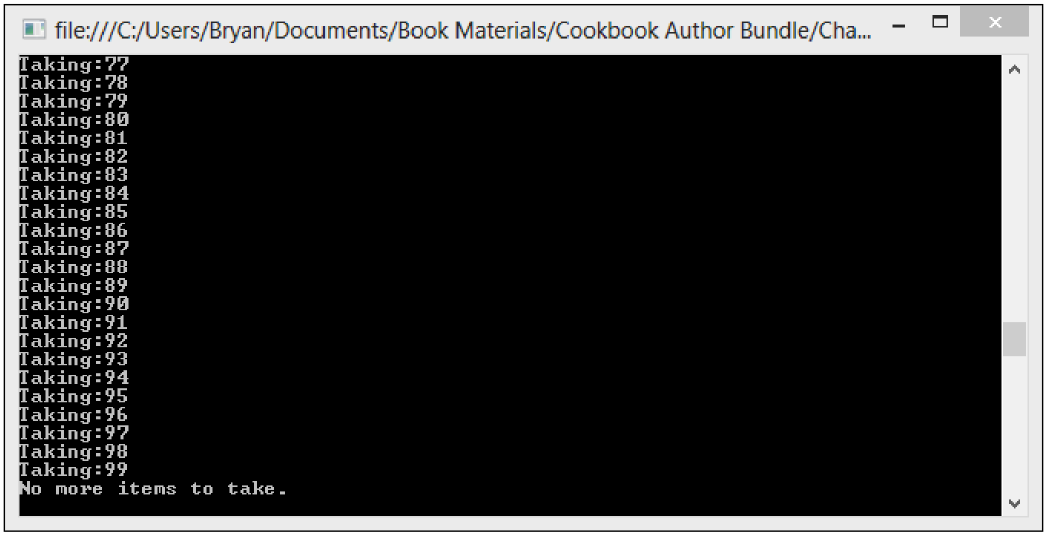
4. Now let's create simple producer `Task` which will use a `for` loop to iterate through the numbers of our source data and add them to the blocking collection. After we are finished with the loop, use the `CompleteAdding` method of `BlockingCollection` to indicate we are done producing data.

```
// A simple blocking producer
Task.Factory.StartNew( ()=>
{
    foreach (var item in data)
    {
        numbers.Add(item);
        Console.WriteLine("Adding:{0} Item Count={1}", item,
numbers.Count);
    }
    numbers.CompleteAdding();
});
```

5. Next, let's create a simple consumer `Task` that uses a `while` loop to take items from `BlockingCollection` and write the output to `Console`. Finish up by waiting for user input before exiting.

```
// A simple blocking consumer.
Task.Factory.StartNew(() =>
{
        int item = -1;
        while (!numbers.IsCompleted)
        {
            try
            {
                item = numbers.Take();
            }
            catch (InvalidOperationException)
            {
                Console.WriteLine("Nothing to take");
                break;
            }
            Console.WriteLine("Taking:{0} ", item);
            // wait for a bit
             Thread.SpinWait(1000);
    }

    Console.WriteLine("\rNo more items to take.");
});

Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

In this recipe, we see how to add items to and remove items from `BlockingCollection`. `BlockingCollection` is actually a wrapper for `IProducerConsumer<T>`, and provides the blocking and bounding capabilities for thread safe collections. `BlockingCollection` takes `IProducerConsumerCollection` in its constructor, or uses `ConcurrentQueue` by default.

Adding the source data to the collection was easy enough. We just had to loop through the source data and call the `Add` method of `BlockingCollection` to add the item. When we are finished adding items to the collection, we call the `CompleteAdding` method. After a collection has been marked as complete for adding, no more adding will be permitted, and threads removing items from the collection will not wait when the collection is empty.

```
foreach (var item in data)
{
    numbers.Add(item);
    Console.WriteLine("Adding:{0} Item Count={1}", item, numbers.
Count);
}
numbers.CompleteAdding();
```

Consumer `Task` uses the `IsCompleted` property of `BlockingCollection` to control a `while` loop. The `IsCompleted` property, as you would expect, indicates if `BlockingCollection` has been marked as complete for adding, and is empty. Inside the `while` loop, we just use the `Take` method to take an item from the collection and display it on the `Console` application.

```
Task.Factory.StartNew(() =>
{
    while (!numbers.IsCompleted)
    {
        try
        {
            item = numbers.Take();
        }
        catch (InvalidOperationException)
        {
            Console.WriteLine("Nothing to take");
            break;
        }
    ...
    }
Console.WriteLine("\rNo more items to take.");
});
```

Iterating a BlockingCollection with GetConsumingEnumerable

`BlockingCollection` provides us with an easier alternative for looping through a collection, and removing items without setting up a `while` loop, and checking the `IsCompleted` property. `BlockingCollection` gives us the ability to do a simple `foreach` loop with the `GetConsumingEnumerable` method.

In this recipe, we are going to create a `Console` application that initializes a range of source data and spins up a producer task to add the data to the collection. The consumer of the collection data will use the `GetConsumingEnumerable` method to get `IEnumerable<T>` for items in the collection.

How to do it...

Let's take a look at how to iterate over a `BlockingCollection` with `GetConsumingEnumerable`.

1. Start a new project using the **C# Console Application** project template and assign `Enumerate` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class.

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. First, in the `Main` method of the `Program` class, let's create our range of input data and our blocking collection.

```
var data = Enumerable.Range(0, 100);
var numbers = new BlockingCollection<int>(100);
```

4. Next let's create a simple producer `Task` which will use a `for` loop to iterate through the numbers of our source data and add them to the blocking collection.

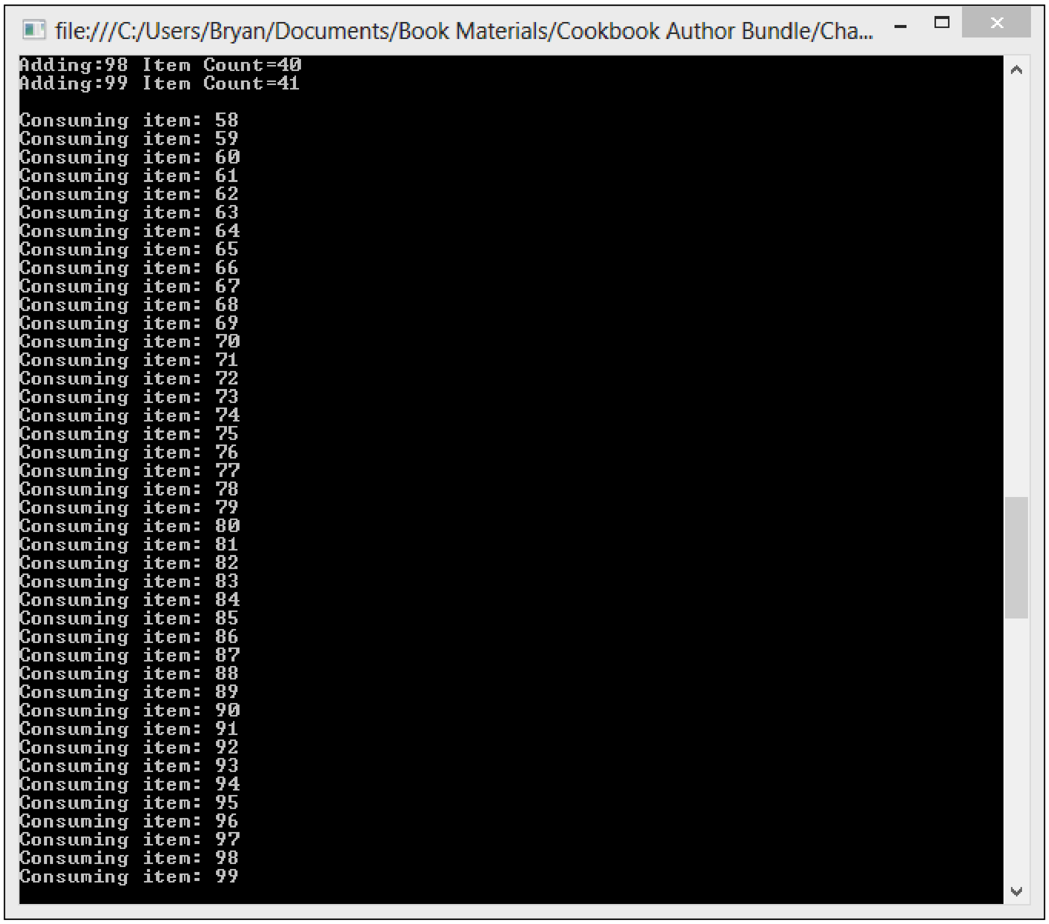
```
// A simple blocking producer
Task.Factory.StartNew( ()=>
{
    foreach (var item in data)
    {
        numbers.Add(item);
        Console.WriteLine("Adding:{0} Item Count={1}", item,
numbers.Count);
    }
    numbers.CompleteAdding();
});
```

5. Finally, let's create a consumer `Task` which will iterate through the collection with a `foreach` loop by calling the `GetConsumingEnumerable` method of blocking collection. Finish up by waiting for user input before exiting.

```
Task.Factory.StartNew(() =>
{
    foreach (var item in numbers.GetConsumingEnumerable())
    {
        Console.Write("\nConsuming item: {0}", item);
    }
```

```
});
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

Producer `Task` in this recipe is exactly the same as producer `Task` created in the first recipe. The only real change to take note of is that we no longer have to set up a `while` loop to take items from the collection, as we did in the first recipe.

```
Task.Factory.StartNew(() =>
{
```

```
        while (!numbers.IsCompleted)
        {
            try
            {
                item = numbers.Take();
            }
            catch (InvalidOperationException)
            {
                Console.WriteLine("Nothing to take");
                break;
            }
        ...
        }
    Console.WriteLine("\rNo more items to take.");
    });
```

By calling the `GetConsumingEnumerable` method of `BlockingCollection`, we can now use much cleaner `foreach` loop syntax.

```
Task.Factory.StartNew(() =>
{
    foreach (var item in result.GetConsumingEnumerable())
    {
        Console.Write("\nConsuming item: {0}", item);
    }
});
Console.ReadLine();
```

The `GetConsumingEnumerable` method takes a snapshot of the current state of the underlying collection and returns `IEnumerable<T>` for the collection items.

Performing LIFO operations with ConcurrentStack

`ConcurrentStack` is the thread safe counterpart of `Systems.Collections.Generic.Stack`, which is the standard **Last-In-First-Out** (**LIFO**) container in the .NET Framework. For algorithms that favor stack usage such as depth-first searches, a thread safe stack is a big benefit.

In this recipe we are going to take a look at the basic usage of `ConcurrentStack`. Our `Console` application for this recipe will initialize a range of data, which a simple producer `Task` will push onto the stack. Consumer `Task` will concurrently pop items from the stack and write them to `Console`.

How to do it...

Now, let's take a look at performing LIFO operations with `ConcurrentStack`.

1. Start a new project using the **C# Console Application** project template and assign `ConcurrentStack` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. First, in the `Main` method of the `Program` class, let's create our range of input data and our blocking collection.

```
var data = Enumerable.Range(0, 100);
ConcurrentStack<int> stack = new ConcurrentStack<int>();
```

4. Next, let's create a simple producer task which will use a `for` loop to iterate through the numbers of our source data and pop them onto the stack.

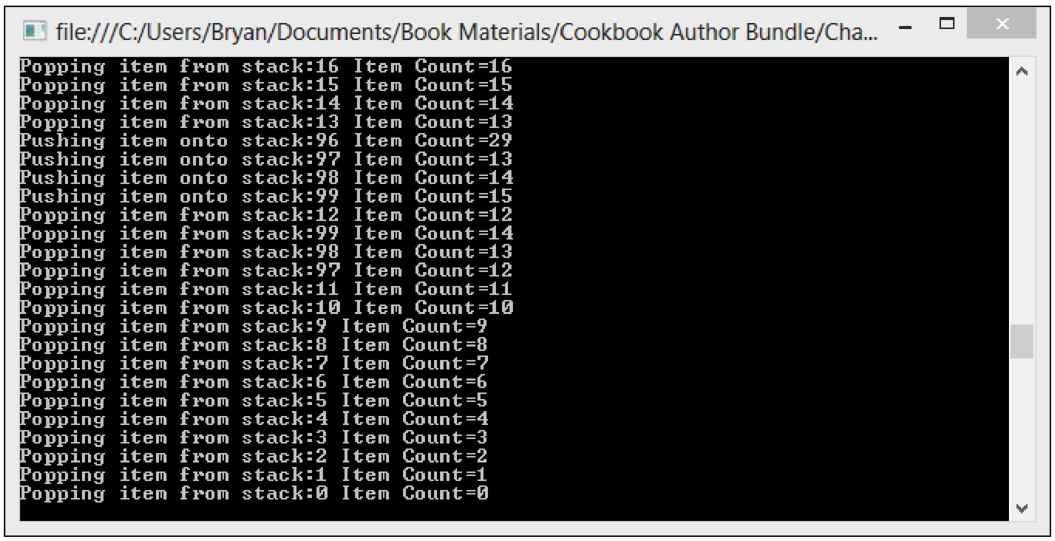
```
// producer
Task.Factory.StartNew(() =>
{
    foreach (var item in data)
    {
        stack.Push(item);
        Console.WriteLine("Pushing item onto stack:{0} Item
Count={1}",
      item, stack.Count);
    }
});
```

5. Now let's create a consumer `Task` which will use a `while` loop to pop items off the stack while the `IsEmpty` property of the stack is false. Finish by waiting for user input before exiting.

```
//consumer
Task.Factory.StartNew(() =>
{
    Thread.SpinWait(1000000);
    while (!stack.IsEmpty)
    {
        int result = 0;
```

```
            stack.TryPop(out result);
            Console.WriteLine("Popping item from stack:{0} Item
Count={1}",
          result, stack.Count);
        }
    });
    Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

`ConcurrentStack` achieves thread safe access by using the `System.Threading.Interlocked` operations. Interlocked operations provide a simple mechanism for synchronizing access to variables shared by multiple threads. Interlocked operations are also very fast.

For the most part, `ConcurrentStack` behaves like `System.Collections.Generic.Stack`. To push an item onto the stack, you just use the `Push` method.

```
foreach (var item in data)
{
    stack.Push(item);
    Console.WriteLine("Pushing item onto stack:{0} Item Count={1}",
    item, stack.Count);
}
```

However, the `Pop` method was removed in favor of `TryPop`. `TryPop` returns `true` if an item existed and was popped, otherwise it returns `false`. The `out` parameter contains the object removed if the pop was successful, otherwise it is indeterminate.

```
while (!stack.IsEmpty)
{
    int result = 0;
    stack.TryPop(out result);
    Console.WriteLine("Popping item from stack:{0} Item Count={1}",
    result, stack.Count);
}
```

Thread safe data lookups with ConcurrentDictionary

`ConcurrentDictionary` is the thread safe counterpart to the generic `dictionary` collection. Both are designed for quick lookups of data based on a key. However, `ConcurrentDictionary` allows us to interleave both reads and updates. `ConcurrentDictionary` achieves its thread safety with no common lock to improve efficiency. It actually uses a series of locks to provide concurrent updates, and has lockless reads.

In this recipe, we will create `ConcurrentDictionary` and initialize it with a small set of key value pairs. Our `dictionary` will be concurrently updated by one task and read by another.

How to do it...

Let's take a look at how to use `ConcurrentDictionary` for data lookups.

1. Start a new project using the **C# Console Application** project template and assign `ConcurrentDictionary` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

3. First, let's create our `ConcurrentDictionary` and initialize it with some data.

```
var dictionary = new ConcurrentDictionary<string, int>();
dictionary["A"] = 1;
dictionary["B"] = 2;
dictionary["C"] = 3;
dictionary["D"] = 4;
dictionary["E"] = 5;
dictionary["F"] = 6;
dictionary["G"] = 7;
dictionary["H"] = 8;
dictionary["I"] = 9;
dictionary["J"] = 10;
dictionary["K"] = 11;
dictionary["L"] = 12;
dictionary["M"] = 13;
dictionary["N"] = 14;
dictionary["O"] = 15;
```

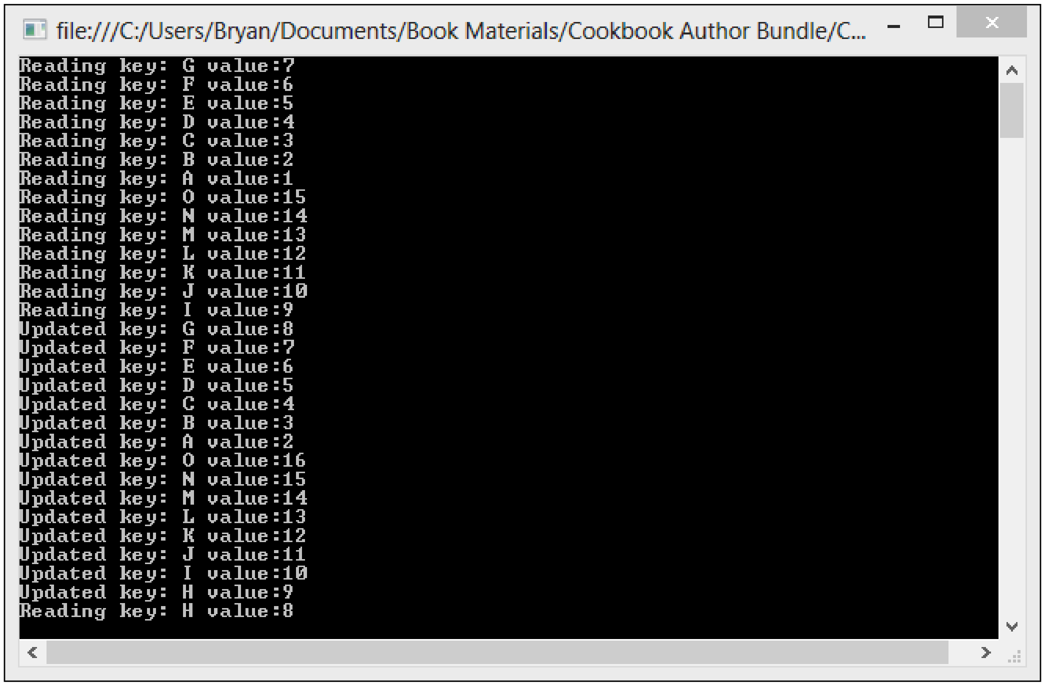
4. Now let's create `Task` to update `dictionary` on a separate thread.

```
// update dictionary on a separate thread
Task.Factory.StartNew(() =>
{
    foreach (var pair in dictionary)
    {
        var newValue = pair.Value + 1;
        dictionary.TryUpdate(pair.Key,newValue,pair.Value);
        Console.WriteLine("Updated key: {0} value:{1}", pair.Key,
newValue);
    }
});
```

5. Now let's create another `Task` which will be concurrently reading from `dictionary`.

```
Task.Factory.StartNew(() =>
{
    foreach (var pair in dictionary)
    {
        Console.WriteLine("Reading key: {0} value:{1}",pair.
Key,pair.Value);
    }
});

Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

`ConcurrentDictionary` behaves like `dictionary` counterpart with slight differences. We are updating `dictionary` using the `TryUpdate` method. This method was added to provide us with an atomic operation to check if the item exists, and if not, add it while still under an atomic lock.

```
foreach (var pair in dictionary)
    {
        var newValue = pair.Value + 1;
        dictionary.TryUpdate(pair.Key,newValue,pair.Value);
        Console.WriteLine("Updated key: {0} value:{1}", pair.Key,
newValue);
    }
```

We are reading `dictionary` directly from the `Key` and `Value` properties of each `KeyValuePair` in the collection.

Cancelling an operation in a concurrent collection

When working with `BlockingCollection`, most `Add` and `Take` operations are performed in a loop. The `TryAdd` and `TryTake` methods of `BlockingCollection` can accept a `CancellationToken` parameter so that we can respond to cancellation requests and break out of a loop.

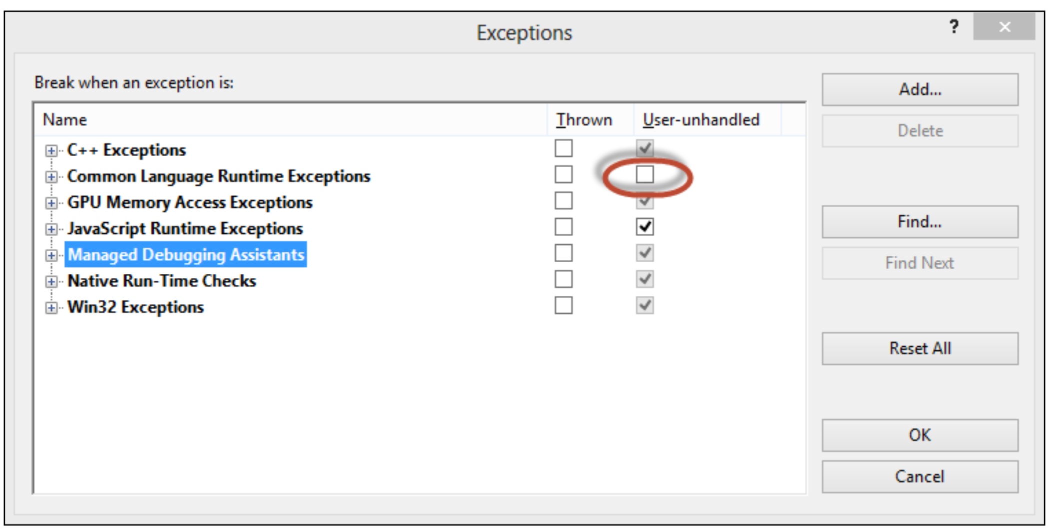
In this recipe, we are going to create a `Console` application that has producer `Task` and consumer `Task`. The producer will be adding items to `BlockingCollection` using `TryAdd`, and the consumer will be removing items using `Try`.

After the producer and consumer get started, we will call the `Cancel` method on a token source to see how we can use the `TryAdd` and `TryTake` overloads to handle cancellation of our operation.

Getting ready...

For this recipe, we need to turn off the Visual Studio 2012 Exception Assistant. The Exception Assistant appears whenever a runtime `Exception` is thrown, and intercepts the `Exception` before it gets to our handler.

1. To turn off the Exception Assistant, go to the **Debug** menu and select **Exceptions**.
2. Uncheck the **User-unhandled** checkbox next to **Common Language Runtime Exceptions**.



How to do it...

Now, let's see how to cancel a concurrent collection operation.

1. Start a new project using the **C# Console Application** project template and assign `CancelOperation` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
```

3. In the `Main` method of the `Program` class, let's create a source range of numbers, instantiate our `CancellationTokenSource` and obtain `CancellationToken`.

```
var data = Enumerable.Range(0, 100);
var numbers = new BlockingCollection<int>(100);
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
```

4. Next, just below the previous lines, create a producer `Task` and pass in `CancellationToken`. The producer should add items to `BlockingCollection` by calling `TryAdd` inside a `try/catch` block. The `catch` block should handle `OperationCancelledException`.

```
Task.Factory.StartNew(() =>
{
    foreach (var item in data)
    {
    try
    {
      numbers.TryAdd(item,5,token);
      Console.WriteLine("Adding:{0} Item Count={1}",
        item, numbers.Count);
    }
    catch(OperationCanceledException)
    {
      Console.WriteLine("Adding operation has been cancelled");
      numbers.CompleteAdding();
      break;
```

```
        }
      }
      numbers.CompleteAdding();
  },token);
```

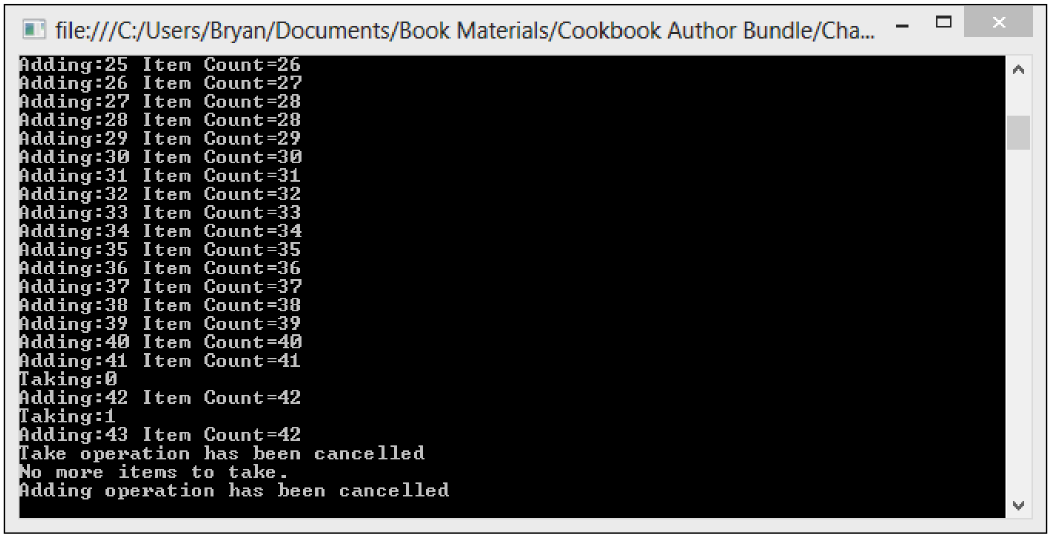
5. Now let's create a consumer `Task` and pass in `CancellationToken`. The consumer task should take items from `BlockingCollection` by calling `TryTake` inside a `try/catch` block. The `catch` block should handle `OperationCancelledException`.

```
// A simple blocking consumer.
Task.Factory.StartNew(() =>
{

  while (!numbers.IsCompleted)
  {
    try
    {
      numbers.TryTake(out item,5,token);
    }
    catch (OperationCanceledException)
    {
      Console.WriteLine("Take operation has been cancelled");
      break;
    }
    Console.WriteLine("Taking:{0} ", item);
    // wait for a bit
    Thread.SpinWait(10000);
  }
  Console.WriteLine("\rNo more items to take.");
},token);
```

6. Finally, let's have the main thread wait for a bit, then call the `Cancel` method of `CancellationTokenSource`. Wait for user input before exiting.

```
Thread.SpinWait(2000000);
tokenSource.Cancel();
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

Responding to cancellations when working with `BlockingCollection` is pretty consistent with other classes in the Task Parallel Library.

In the producer task, we use the overload of `TryAdd` that accepts an out parameter, a timeout parameter and `CancellationToken`. We also call `TryAdd` in a `try/catch` block, so we can respond to `OperationCancelledException`. When the operation is cancelled, we call `CompleteAdding` to indicate we will be adding more items and execute a `break` statement to break out of the loop.

```
foreach (var item in data)
{
  try
  {
    numbers.TryAdd(item,5,token);
    Console.WriteLine("Adding:{0} Item Count={1}",
              item, numbers.Count);
  }
  catch(OperationCanceledException)
  {
    Console.WriteLine("Adding operation has been cancelled");
    numbers.CompleteAdding();
    break;
  }
}
```

Things are very similar on the consumer side. We pass `CancellationToken` into `TryTake` and handle `OperationCancelledException` in our `catch` block. When the operation is cancelled, we issue a `break` statement to break out of the loop.

Working with multiple producers and consumers

It is possible to use single `BlockingCollection` as a buffer between multiple producers and consumers.

In this recipe, we are going to build a `Console` application that will create multiple producer tasks which perform an expensive math operation on a small range of numbers. We will also have two consumer tasks that loop through the `BlockingCollection` buffer and display the results.

How to do it...

Now, let's take a look at using a single `BlockingCollection` with multiple producers and consumers.

1. Start a new project using the **C# Console Application** project template and assign `MultiptleProducerConsumer` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

3. Let's start by creating a `static` method on the `Program` class which performs our expensive math operation.

```
private static double CalcSumRoot(int root)
{
  double result = 0;
  for (int i = 1; i < 10000000; i++)
  {
    result += Math.Exp(Math.Log(i) / root);
  }
  return result;
}
```

4. Now let's create another `static` method on the `Program` class that the consumers will use to display the results to the `Console` application. This method will call `GetConsumingEnumerable` on `BlockingCollection` and loop through the collection.

```
private static void DisplayResults(BlockingCollection<double>
results)
{
  foreach (var item in results.GetConsumingEnumerable())
  {
    Console.Write("\nConsuming item: {0}", item);
  }
}
```

5. Next, in the `Main` method of the `Program` class, let's define our `BlockingCollection` buffer and create `List<Task>`, so we can coordinate our multiple tasks, and create a couple of simple consumer `tasks` that will use the `DisplayResults` method to print out the results.

```
var results = new BlockingCollection<double>();
var tasks = new List<Task>();
var consume1 = Task.Factory.StartNew(() =>
DisplayResults(results));
var consume2 = Task.Factory.StartNew(() =>
DisplayResults(results));
```

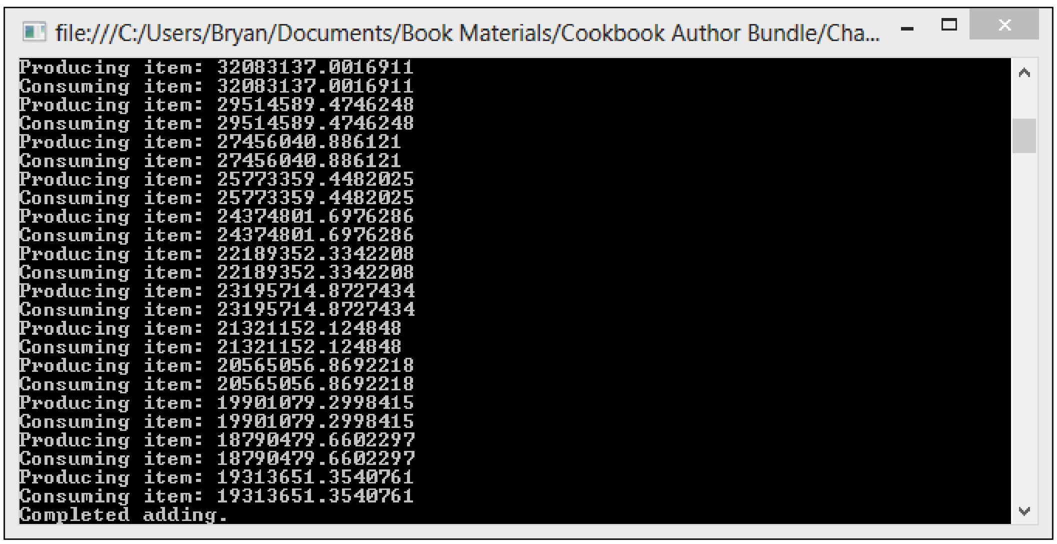
6. Now we need to create a `for` loop that loops from one to twenty-five, creating producer tasks that use the `CalcSumRoot` method to calculate the result, and then add the result to `BlockingCollection` by calling `TryAdd`. The loop must also add all of producer `tasks` to the `Task` list.

```
for (int item = 1; item < 25; item++)
{
  var value = item;
  var compute = Task.Factory.StartNew(() =>
  {
    var calcResult = CalcSumRoot(value);
    Console.Write("\nProducing item: {0}", calcResult);
    results.TryAdd(calcResult);
  });
  tasks.Add(compute);
}
```

7. Finally, let's create a continuation to run after all producer `tasks` that are complete. The continuation simply calls the `CompleteAdding` method of `BlockingCollection` to indicate that we are done adding items to the collection. Finish up by waiting for user input before exiting.

   ```
   Task.Factory.ContinueWhenAll(tasks.ToArray(),
   result =>
   {
     results.CompleteAdding();
     Console.Write("\nCompleted adding.");
   });

   Console.ReadLine();
   ```

8. In Visual Studio 2012, press *F5* to run the project. Notice the ordered results in the following screenshot:



How it works...

By default, `BlockingCollection` uses `ConcurrentQueue<T>` as the backing store. `ConcurrentQueue` takes care of thread synchronization and `BlockingCollection` does a non-busy wait while trying to take an item from the collection. That is, if the consumer calls `TryTake` when there are no items in the queue, it does a non-busy wait until any items are available.

In this recipe, we are spinning up producer `tasks` in a `for` loop. Each producer `task` is calling the `CalcSumRoot` method which is a fairly expensive math operation. Our consumers are simply displaying the output to the screen. As a result, our two consumer `tasks` are probably spending most of their time in a non busy wait state.

The producers and consumers are pretty simple, but we needed a way to call `CompleteAdding` after all producer `tasks` have finished. We handled this by adding all of our producer `Task` objects to `List<Task>`, and calling the `ContinueWhenAll` method of `Task.Factory`, so our continuation only runs when all of the producers complete. The only job of the continuation is to call the `CompleteAdding` method of `BlockingCollection`.

```
Task.Factory.ContinueWhenAll(tasks.ToArray(),
result =>
{
  results.CompleteAdding();
  Console.Write("\nCompleted adding.");
});

Console.ReadLine();
```

Creating object pool with ConcurrentStack

An object pool is a set of pre-initialized objects that your application can use, rather than creating and destroying all of the objects it needs. If the instantiation cost of an object type is high, your application might benefit from a pool of objects.

In this recipe, we are going to create an object pool based on `ConcurrentStack`. `ConcurrentStack` will handle concurrent access issues using fast interlocked operations, and will dispense our objects in a LIFO manner. We will also have an object pool client which creates three tasks. One creates objects and puts them in the pool, the other two tasks request objects from the pool on different threads.

How to do it...

Let's see how we can use `ConcurrentStack` to build a pool of pre-initialized objects.

1. Start a new project using the **C# Console Application** project template and assign `ObjectPool` as the **Solution name**.
2. Let's start by creating our object pool class. Right-click on the `ObjectPool` project in the **Solution Explorer** and click on **Add**, then choose **New Item**. Select **Visual C# Items**, and **Class.** Enter `ConcurrentObjectPool` as the name of the class.

3. Add the following `using` directives to the top of your `ConcurrentObjectPool` class:

   ```
   using System;
   using System.Collections.Concurrent;
   ```

4. We want our object pool to work with any type, so add a generic type parameter after the class name.

   ```
   public class ConcurrentObjectPool<T>
   {
   }
   ```

5. Our `ObjectPool` class is going to need a couple of private state fields. We need a `ConcurrentStack` field which will provide our backing store and a `Func<T>` field which will hold an object creation function the pool can use to generate objects when the pool is empty. Inside the class declaration, add the following fields:

   ```
   private ConcurrentStack<T> _objects;
   private Func<T> _objectInitializer;
   ```

6. Now we need a constructor for the `ConcurrentObjectPool` class. The constructor should take a `Func<T>` argument for the object generator and should instantiate a new `ConcurrentStack` object as the backing store.

   ```
   public ConcurrentObjectPool(Func<T> objectInitializer)
   {
     _objects = new ConcurrentStack<T>();
     _objectInitializer = objectInitializer;
   }
   ```

7. Now we need a `GetObject` method which will return a new object to the client. The `GetObject` method will try to pop an object off the stack. If it can't pop one off the stack, it will use `objectInitializer` to instantiate a new object.

   ```
   public T GetObject()
   {
     T item;
     if (_objects.TryPop(out item)) return item;
     return _objectInitializer();
   }
   ```

8. The last step for our object pool is a `PutObject` method that takes a generic item parameter and pushes it on the stack.

   ```
   public void PutObject(T item)
   {
     _objects.Push(item);
   }
   ```

9. Now we need to create the `Console` application that will use the object pool. Go back to `Program.cs` and add the following `using` directives at the top of the file:

```
using System;
using System.Text;
using System.Threading.Tasks;
```

10. The first step is to instantiate our object pool. In the `main` method of the program class, create a `ConcurrentObjectPool` object and pass in a function that creates a new `StringBuilder` object as the constructor parameter.

```
var pool = new ConcurrentObjectPool<StringBuilder>(()=>
  new StringBuilder("Pooled Object created by
objectInitializer"));
```

11. Now let's create a task that creates some objects and places them in `pool` using the `PutObject` method.

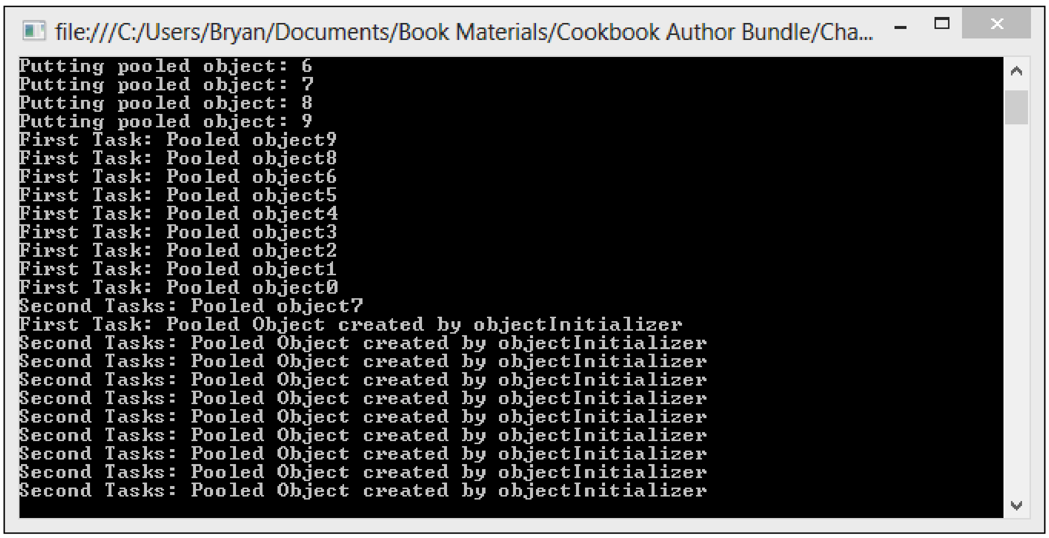
```
var task1 = Task.Factory.StartNew(() =>
{
  for (var index = 0; index < 10; index++)
  {
    StringBuilder newObject = new StringBuilder(string.
Concat("Pooled object",
          index.ToString()));
    Console.WriteLine("Putting pooled object: {0}", index.
ToString());
    pool.PutObject(newObject);
  }
});
```

12. Finally, let's create two continuation tasks that run after the first task is completed. Both tasks just request objects from the object `pool` using the `GetObject` method.

```
task1.ContinueWith((antecedent)=>
{
  for (var index = 0; index < 10; index++)
  {
    var pooledObject = pool.GetObject();
    Console.WriteLine("First Task: {0}", pooledObject.ToString());
  }
});

task1.ContinueWith((antecedent) =>
{
  for (var index = 0; index < 10; index++)
```

```
        {
          var pooledObject = pool.GetObject();
          Console.WriteLine("Second Tasks: {0}", pooledObject.
ToString());
        }
    });
```

13. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

There are other features we could have added to our object `pool`, such as controlling the concurrency level and/or using thread local segments to store our objects, but this simple implementation does the job for our purposes.

The constructor of the object `pool` takes a function argument that it can use to generate an object if the pool is empty, and stores the function in a private `objectInitializer` field. We are just pooling `StringBuilder` objects in this sample, so we passed in the following function:

```
()=>new StringBuilder("Pooled Object created by objectInitializer")
```

Our `GetObject` method, which the client uses to get objects from `pool`, just uses the `TryPop` method of `ConcurrentStack` to return an object. If `TryPop` fails to return anything because the stack is empty, we just return the result of the `objectInitializer` function.

```
public T GetObject()
{
  T item;
  if (_objects.TryPop(out item)) return item;
  return _objectInitializer();
}
```

The `PutObject` method probably doesn't require much explanation. It just uses the `Push` method of `ConcurrentStack` to push an object onto the stack.

Given that we chose to use `ConcurrentStack`, our object references are returned in a LIFO fashion. We could have chosen another type of backing store if this didn't work for us. For example, we could have chosen to use `ConcurrentQueue` as a backing store to have items returned in a **First-In-First-Out** (**FIFO**) fashion, or we could have used `ConcurrentBag` to provide unordered storage.

Adding blocking and bounding with IProducerConsumerCollection

By default, `BlockingCollection` uses `ConcurrentQueue` as its backing store. However, you can add blocking and bounding functionality to any custom or derived collection class by implementing the `IProducerConsumerCollection` interface in the class. You can then use an instance of the custom collection class as the backing store for `BlockingCollection`.

In this recipe, we are going to create a custom priority queue and use the custom queue as the backing store for `BlockingCollection`.

How to do it...

Let's examine how we can use `IProducerConsumerColletion` to add blocking and bounding functionality to a custom collection.

1. Start a new project using the **C# Console Application** project template and assign **CustomBlockingBounding** as the **Solution name**.
2. First, let's add a `Class` file for our custom queue. Right-click on the **CustomBlockingBounding** project and click on **Add Item**, and then click on **Add New Item** and then click on **Class**. Name the new class `PriorityQueue.cs`.
3. Add the following `using` directives to the top of your `PriorityQueue` class:

```
using System.Collections.Concurrent;
using System.Collections.Generic;
```

```
using System.Collections;
using System;
using System.Threading;
```

4. Below the `PriorityQueue` class, let's create an enumeration for our queue priority levels. We just want to use low, medium, and high as the possible priority levels.

```
public enum QueuePriorityLevel
{
  High = 0,
  Medium = 1,
  Low = 2
}
```

5. Our custom collection class will hold `KeyValuePairs` of the queue priority level and the data queued. Add priority level and queued data generic type parameters to `Class` and declare the `IProducerConsumerCollection` interface.

```
public class PriorityQueue<PriorityLevel, TValue>:
  IProducerConsumerCollection<KeyValuePair<QueuePriorityLevel,
TValue>>
{

}
```

6. Next, we need some private fields for the `PriorityQueue` class. We will need three `ConcurrentQueue<QueuePriorityLevel, TValue>` fields; one each for the low, medium, and high priority queues. We will need an array of `ConcurrentQueue` to hold all of the queues and an integer count variable.

```
private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_lowPriotityQueue = null;
private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_mediumPriotityQueue = null;
private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_highPriotityQueue = null;
private ConcurrentQueue<KeyValuePair<QueuePriorityLevel,
TValue>>[] _queues = null;
private int _count = 0;
```

7. Now let's add a default constructor to the `PriorityQueue` class that initializes all of our fields.

```
public PriorityQueue()
{
  _lowPriotityQueue = new ConcurrentQueue<KeyValuePair<QueuePriori
tyLevel,TValue>>();
```

```
  _mediumPriotityQueue = new ConcurrentQueue<KeyValuePair<QueuePri
orityLevel,TValue>>();
  _highPriotityQueue = new ConcurrentQueue<KeyValuePair<QueuePrior
ityLevel,TValue>>();
  _queues = new ConcurrentQueue<KeyValuePair<QueuePriorityLevel,
TValue>>[3]
  {
    _lowPriotityQueue,
    _mediumPriotityQueue,
    _highPriotityQueue
  };
}
```

8. Next, we need to provide an implementation for several of the `IProducerConsumerCollection` interface members. Let's start with the `CopyTo` method. This method makes a copy of our collection array to a destination array.

```
public void CopyTo(KeyValuePair<QueuePriorityLevel, TValue>[]
array, int index)
{
  if (array == null) throw new ArgumentNullException();

  KeyValuePair<QueuePriorityLevel, TValue>[] temp = this.
ToArray();
  for (int i = 0; i < array.Length && i < temp.Length; i++)
    array[i] = temp[i];
}
```

9. Now we need to provide an implementation for the `ToArray` method which returns an array of `KeyValuePairs`.

```
public KeyValuePair<QueuePriorityLevel, TValue>[] ToArray()
{
  KeyValuePair<QueuePriorityLevel, TValue>[] result;

  lock (_queues)
  {
    result = new KeyValuePair<QueuePriorityLevel, TValue>[this.
Count];
    int index = 0;
    foreach (var q in _queues)
    {
      if (q.Count > 0)
      {
        q.CopyTo(result, index);
```

```
            index += q.Count;
          }
        }
        return result;
      }
    }
```

10. Now we are getting to the key `IProducerConsumerCollection` method implementations. We need to provide an implementation for the `TryAdd` method which is going to determine our private `ConcurrentQueue` collections to add the new item to, and then add the item, and use `Interlocked.Increment` to increment the count.

```
public bool TryAdd(KeyValuePair<QueuePriorityLevel, TValue> item)
{
  int priority = (int) item.Key;
  _queues[priority].Enqueue(item);
  Interlocked.Increment(ref _count);
  return true;
}
```

11. The `TryTake` method implementation needs to loop through the backing `ConcurrentQueues` in priority order, and try to take the first available item from one of the queues, and decrement the count.

```
public bool TryTake(out KeyValuePair<QueuePriorityLevel, TValue>
item)
{
  bool success = false;

  for (int i = 0; i <= 2; i++)
  {
    lock (_queues)
    {
      success = _queues[i].TryDequeue(out item);
      if (success)
      {
        Interlocked.Decrement(ref _count);
        return true;
      }
    }
  }

  item = new KeyValuePair<QueuePriorityLevel, TValue>(0,
default(TValue));
  return false;
}
```

12. Next we need to implement the `GetEnumerator` methods required to implement `IEnumerable`.

```
public IEnumerator<KeyValuePair<QueuePriorityLevel, TValue>>
GetEnumerator()
{
  for (int i = 0; i <= 2; i++)
  {
    foreach (var item in _queues[i])
      yield return item;
  }
}

IEnumerator IEnumerable.GetEnumerator()
{
  return GetEnumerator();
}
```

13. We're almost done with the collection. The last thing we need to do is implement a simple getter for the `count` field. There is no need to provide an implementation for the other `IProducerConsumerCollection` members.

```
public int Count
{
  get { return _count; }
}
```

14. Ok, let's move on to our `Console` application which will use the custom `queue` class. Open `Program.cs` and add the following `using` directives to the top of the class:

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

15. In the `Main` method of your `Program` class, start with some local variable declarations for a `PriorityQueue` variable, a `BlockingCollection` variable that takes the `PriorityQueue` variable as a constructor argument, and a list of tasks which will hold references to our producer and consumer `tasks`.

```
var queue = new PriorityQueue<QueuePriorityLevel, int>();
var bc = new BlockingCollection<KeyValuePair<QueuePriorityLevel,
int>>(queue);
var tasks = new List<Task>();
```

16. Our `Console` application has a producer task which will add items to the priority queue with a random priority level. The consumer task will remove items from the queue in priority order and write the results to `Console`. Let's start with the `producer` task.

```
var producer = Task.Factory.StartNew(() =>
{
  Random r = new Random();
  int itemsToAdd = 50;
  int count = 0;
  for (int i = 0; i < itemsToAdd; i++ )
  {
    Thread.SpinWait(10000);
    // Generate random priority level
    QueuePriorityLevel[] values = (QueuePriorityLevel[])Enum.GetVa
lues(typeof(QueuePriorityLevel));
    var priority = values[new Random().Next(0, values.Length)];
    var item = new KeyValuePair<QueuePriorityLevel, int>(priority,
count++);
    bc.Add(item);
    Console.WriteLine("added priority {0}, data={1}", priority,
item.Value);
  }
  Console.WriteLine("Producer is finished.");
 }).ContinueWith( (antecedent)=>
   {
     bc.CompleteAdding();
   });
```

17. Right after the `producer` task, make a call to `Thread.SpinWait` to make the main thread wait for a bit before starting the `consumer` task.

```
Thread.SpinWait(100000);
```

18. Now let's add the `consumer` task which will pull items from the queue and display the results to the `Console` application.

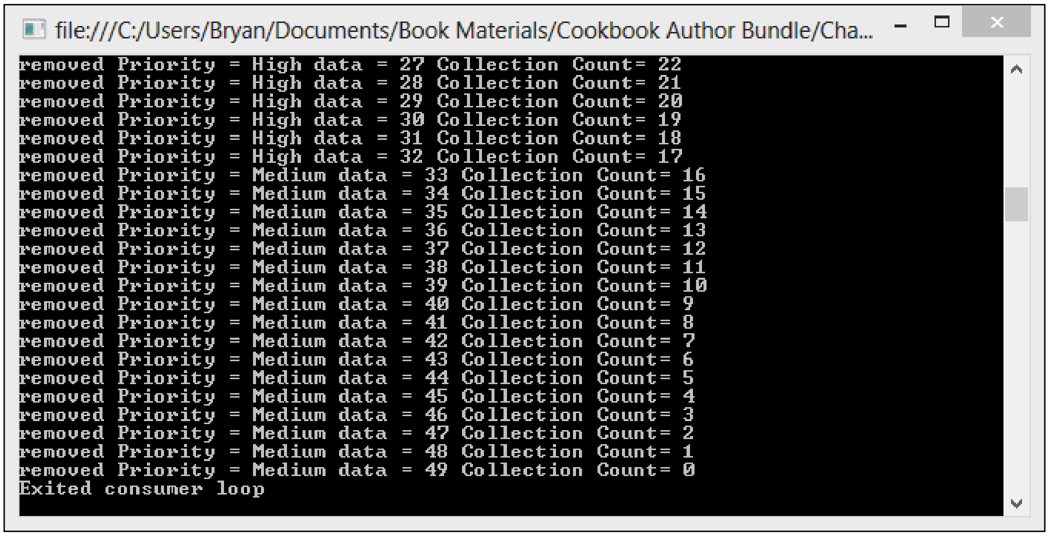
```
var consumer = Task.Factory.StartNew(() =>
{
  while (!bc.IsCompleted )
  {
     KeyValuePair<QueuePriorityLevel, int> item = new KeyValuePair
<QueuePriorityLevel, int>();
```

```
    bool success = false;
    success = bc.TryTake(out item);
    if (success)
    {
    Console.WriteLine("removed Priority = {0} data = {1}
Collection Count= {2}", item.Key, item.Value, bc.Count);
    }
    else
      Console.WriteLine("No items remaining. count = {0}",
bc.Count);
  }
  Console.WriteLine("Exited consumer loop");
});
```

19. Finish up by adding the `producer` and `consumer` tasks to the list of `tasks`. Wait on both tasks to complete by calling `Task.WaitAll` inside a `try/catch` block. In the `catch` block, handle any `AggregateException` that may be thrown. Lastly, wait for user input before exiting.

```
tasks.Add(producer);
tasks.Add(consumer);

try
{
  Task.WaitAll(tasks.ToArray());
}

catch (AggregateException ae)
{
  foreach (var v in ae.InnerExceptions)
    Console.WriteLine(v.Message);
}

Console.ReadLine();
```

20. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

There were a lot of codes in this recipe, but the key points of the implementation can be distilled to just a few `IProducerConsumerCollection` interface method implementations.

`IProducerConsumerCollection<T>` defines a handful of methods for manipulating thread safe collections for producer/consumer usage.

To create our custom collection class, we just implemented the `IProducerConsumerCollection` interface on our custom `PriorityQueue` class and used some `ConcurrentQueue` fields as our backing stores.

```
public class PriorityQueue<PriorityLevel, TValue>:
    IProducerConsumerCollection<KeyValuePair<QueuePriorityLevel,
TValue>>
{
  private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_lowPriotityQueue = null;
  private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_mediumPriotityQueue = null;
  private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>
_highPriotityQueue = null;
```

```
    private ConcurrentQueue<KeyValuePair<QueuePriorityLevel, TValue>>[]
  _queues = null;
    private int _count = 0;
    ...
  }
```

The actual implementation of the `IProducerConsumerCollection.TryAdd` method is pretty simple. We just determine the queue to place the item in by casting our `QueuePriorityLevel` enumeration to an integer, then enqueue the item. We then do `Interlocked.Increment` on our count field. `Interlocked.Increment` does a thread safe increment of the count field.

```
public bool TryAdd(KeyValuePair<QueuePriorityLevel, TValue> item)
{
  int priority = (int) item.Key;
  _queues[priority].Enqueue(item);
  Interlocked.Increment(ref _count);
  return true;
}
```

`TryTake` isn't much more complex. We just loop through our three private backing queues in order of priority and remove the first item we come to. `TryTake` returns a bool to indicate of it was successful in taking an item.

```
public bool TryTake(out KeyValuePair<QueuePriorityLevel, TValue> item)
{
  bool success = false;

  for (int i = 0; i <= 2; i++)
  {
    lock (_queues)
    {
      success = _queues[i].TryDequeue(out item);
      if (success)
      {
        Interlocked.Decrement(ref _count);
        return true;
      }
    }
  }
}
```

Using multiple concurrent collections to create a pipeline

A pipeline is like an assembly line in a factory. With the pipeline pattern, data is processed in a sequential order where the output from the first stage becomes the input for the second stage and so on. Pipelines use parallel tasks and concurrent queues to process a series of input values.

In this recipe, we are going to create a simple pipeline that creates a range of numbers, doubles the numbers in the range, and then writes the results to `Console`.

How to do it...

Now, let's see how to create a pipeline by using multiple concurrent collections.

1. Start a new project using the **C# Console Application** project template and assign `Pipeline` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

   ```
   using System;
   using System.Collections.Concurrent;
   using System.Threading.Tasks;
   ```

3. First, let's add a `static` method to the `Program` class to create the range. This method needs a `BlockingCollection` parameter. It will simply add items to `BlockingCollection` in a loop.

   ```
   static void CreateRange(BlockingCollection<int> result)
   {
     try
     {
       for (int i = 1; i < 10; i++)
       {
         result.Add(i);
         Console.WriteLine("Create Range {0}", i);
       }
     }
     finally
     {
       result.CompleteAdding();
     }
   }
   ```

4. Next, let's create a `static` method to square the range. This method will take two `BlockingCollection` parameters and will square each of the items in the source collection and place them in the result collection.

```
static void SquareTheRange(BlockingCollection<int> source,
BlockingCollection<int> result)
{
  try
  {
    foreach (var value in source.GetConsumingEnumerable())
    {
      result.Add((int)(value * value));
    }
  }
  finally
  {
    result.CompleteAdding();
  }
}
```

5. Now let's create a `static` method to display the results. This method will take a `BlockingCollection` parameter and will loop through its items and write the values to the `Console` application.

```
{
  foreach (var value in input.GetConsumingEnumerable())
  {
    Console.WriteLine("The result is {0}", value);
  }
}
```

6. In the `Main` method of the `Program` class, declare the two `BlockingCollection<int>` variables. These blocking collections will be the data buffers for the pipeline.

```
var bufferA = new BlockingCollection<int>(10);
var bufferB = new BlockingCollection<int>(10);
```

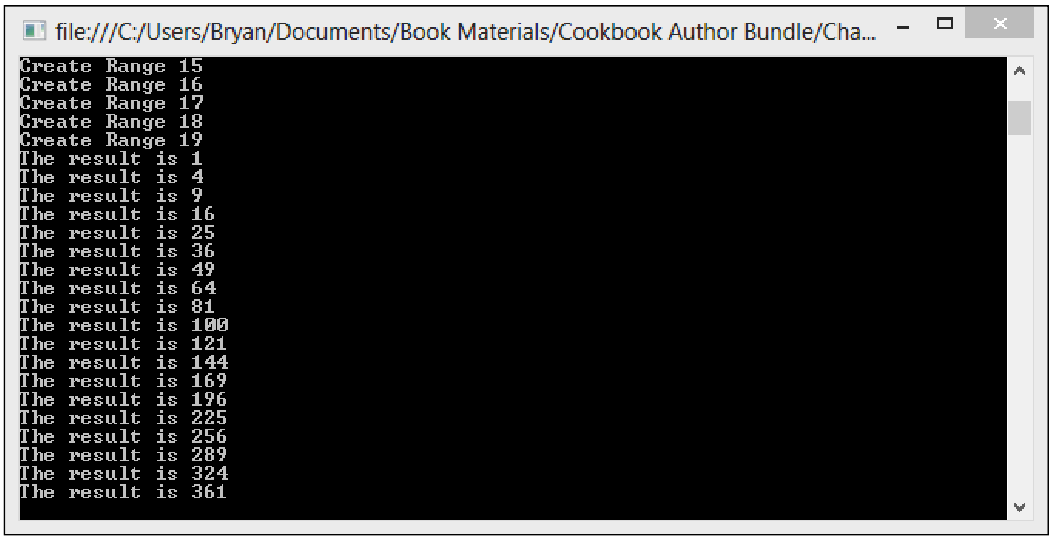
7. Create and start three tasks to call each of our three methods passing in the `BlockingCollection` buffers required for each method.

```
var createStage = Task.Factory.StartNew(() =>
  {
    CreateRange(bufferA);
  },TaskCreationOptions.LongRunning);
var squareStage = Task.Factory.StartNew(() =>
  {
    SquareTheRange(bufferA, bufferB);
  },TaskCreationOptions.LongRunning );

var displayStage = Task.Factory.StartNew(() =>
  {
    DisplayResults(bufferB);
  },TaskCreationOptions.LongRunning);
```

8. Finally, wait for all three tasks to complete by calling `Task.WaitAll`. Wait for user input before exiting.

```
Task.WaitAll(createStage, squareStage, displayStage);
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. You should see the output as shown in the following screenshot:



How it works...

In this recipe, we created a simple pipeline composed of three stages. Each stage reads from and/or writes to a particular buffer. If your machine has more available processor cores than there are stages in the pipeline, the stages can run in parallel. The concurrent queues used by `BlockingCollection` will buffer all shared inputs and outputs.

Each stage in the pipeline can add items to its output buffer as long as there is room. If the buffer is full, the pipeline stage waits for space to become available before adding an item. The stages can also wait on inputs from the previous stage.

The stages that produce data use `BlockingCollection.CompleteAdding` to signal that they are finished adding data. This tells the consumer that it can end its processing loop after all previously added data has been removed or processed.

6
Synchronization Primitives

In this chapter, we will cover the following recipes:

- Using monitor
- Using mutual exclusion lock
- Using `SpinLock` for synchronization
- Interlocked operations
- Synchronizing multiple tasks with a Barrier
- Using `ReaderWriterLockSlim`
- Using `WaitHandles` with Mutex
- Waiting for multiple threads with `CountdownEvent`
- Using `ManualResetEventSlim` to spin and wait
- Using `SemaphoreSlim` to limit access

Introduction

This chapter is about coordinating the work that is performed by parallel tasks.

When concurrent tasks read from and write to variables without an appropriate synchronization mechanism, a **race condition** has the potential to appear. Race conditions can produce inconsistent results in your program, and can be very difficult to detect and correct.

Let's take a second to understand what a race condition is. Consider a scenario that has two parallel tasks; task1 and task2. Each task tries to read and increment the value of a public variable. Task1 reads the original value of the variable, let's say 10, and increments the value to 11. At the same time task1 is reading the value of the variable but before it increments the value, task2 reads the same value of 10 and increments to 11. The final value of the variable ends up being 11 instead of the correct value of 12.

.NET framework 4.5 offers several new data structures for parallel programming that simplify complex synchronization problems. Knowledge of these synchronization primitives will enable you to implement more complex algorithms and solve many of the issues associated with multithreaded programming. It is important to learn the various alternatives so that you can choose the most appropriate one for scenarios that require communication and synchronization among multiple tasks.

Using monitor

A monitor, like the lock statement, is a mechanism for ensuring that only one thread at a time may be running in a critical section of code. A monitor has a lock, and only one thread at a time may acquire it. To run in a critical section of code, a thread must have acquired the monitor. While a thread owns the lock for an object, no other thread can acquire that lock.

For this recipe, we are going to create an application that uses a `ConsoleWriter` class with a `WriteNumbers` method to write some numbers out to the Console. Three parallel tasks will each be trying to write some numbers to the Console, and we will use monitor to control access to the critical section of code.

How to do it...

Have a look at the following steps:

1. Start a new project using the **C# Console Application** project template and assign `MonitorExample` as the **Solution name**.
2. Add a new class to your project and name the class `ConsoleWriter.cs`.
3. Add the following code snippet using the directives to the top of your `ConsoleWriter` class:

   ```
   using System;
   using System.Threading;
   ```

4. First, inside the declaration of your `ConsoleWriter` class, create a private member variable of type object which we will use as our lock object.

   ```
   public class ConsoleWriter
   {
   ```

```
  private object _locker = new object();
}
```

5. Now let's create a method of the `ConsoleWriter` class called `WriteNumbersUnprotected`. This is a simple method that executes a `for` loop. Each iteration of the loop will write the number of the loop index to the Console. As you might have guessed by the method name, we will not use monitor to lock the critical section of this method.

```
public void WriteNumbersUnprotected()
{
  for (int numbers = 0; numbers < 5; numbers++)
  {
    Thread.Sleep(100);
    Console.Write(numbers + ",");
  }
  Console.WriteLine();
}
```

6. Now let's create a method on the `ConsoleWriter` class named `WriteNumbers`. This method will have the same functionality as the previous method we created. However, this method will use monitor to ensure that only a single thread can enter the critical section of code.

```
public void WriteNumbers()
{
  Monitor.Enter(_locker);
  try
  {
    for (int number = 0; number <= 5; number++)
    {
      Thread.Sleep(100);
      Console.Write(number + ",");
    }
    Console.WriteLine();
  }
  finally
  {
    Monitor.Exit(_locker);
  }
}
```

7. Now let's go back to the `Program` class. Add the following code snippet using directives to the top of the `Program` class:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
```

8. In the `Main` method of the `Program` class, start by creating, and instantiating an instance of `ConsoleWriter`. Also create a list of tasks which we will use to hold a reference to our tasks.

```
static void Main(string[] args)
{
  var writer = new ConsoleWriter();
  var taskList = new List<Task>();
}
```

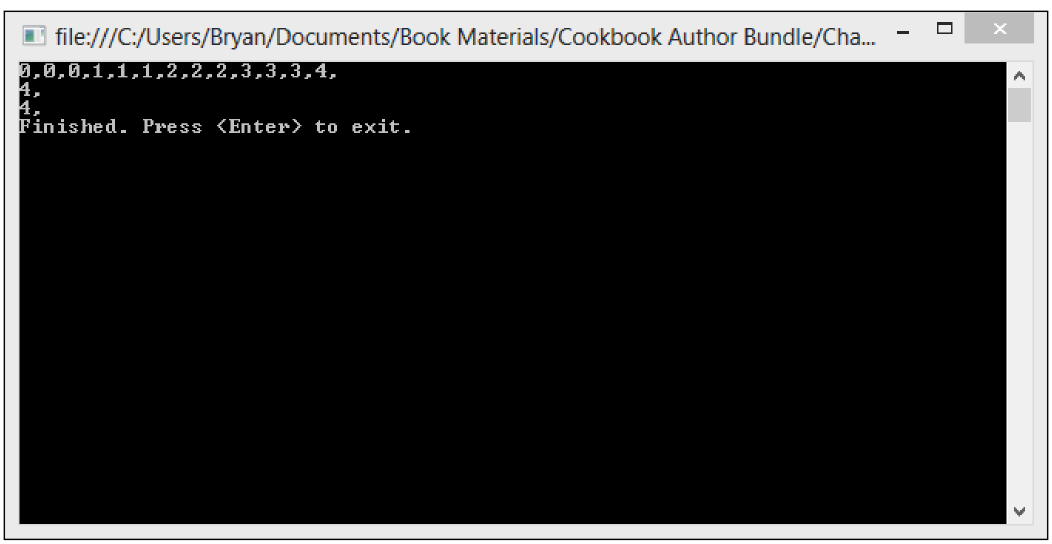
9. Now let's create a `for` loop that creates and starts three tasks. The task will each call the `WriteNumbersUnprotected` method of the shared `ConsoleWriter` object.

```
for (int i = 0; i < 3; i++)
{
  taskList.Add(Task.Factory.StartNew(()=>
    {
      writer.WriteNumbersUnprotected();
    }));
}
```

10. Finish up the `Main` method of the `Program` class by waiting on the tasks to complete and waiting on user input before exiting.

```
Task.WaitAll(taskList.ToArray());
Console.WriteLine("Finished. Press <Enter> to exit.");
Console.ReadLine();
```

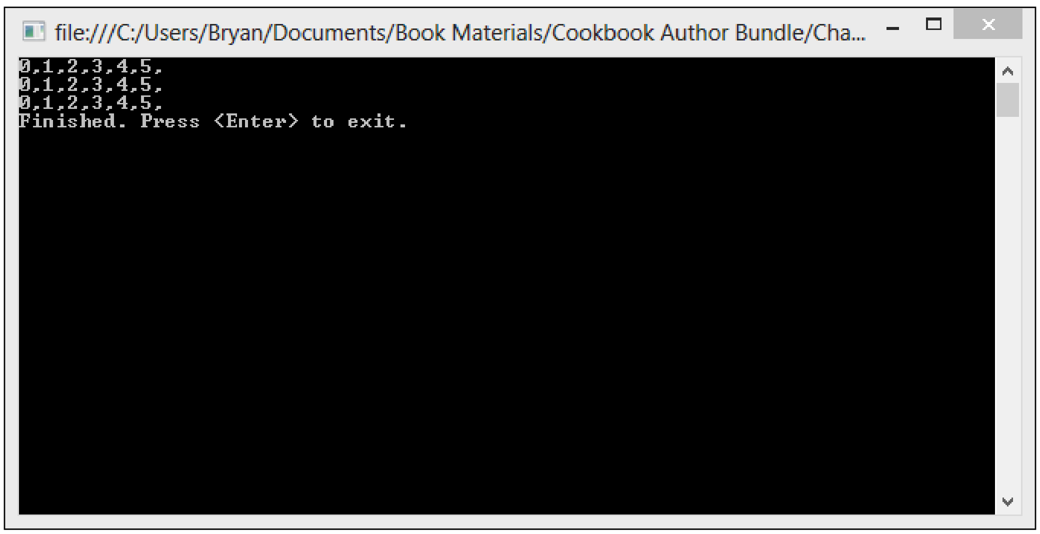
11. Now, in Visual Studio 2012, press *F5* to run the project. You will probably see some pretty ugly output because more than one thread is calling the method at a time. Have look at the following screenshot:



12. Let's fix this by going back to our tasks in the `Main` method of the `Program` class and change them now to call the `WriteNumbers` method which protects the critical section of code with monitor.

```
for (int i = 0; i < 3; i++)
{
  taskList.Add(Task.Factory.StartNew(()=>
    {
      writer.WriteNumbers();
    }));
}
```

13. Let's press *F5* again to run the project. This time you should see more orderly output because only one thread at a time can be in the critical section. This is shown in the following screenshot:



How it works...

Lock and monitor are very similar. In fact, the `lock` keyword is implemented using the `Monitor` class. When using `Monitor`, the developer has to be a bit more careful to explicitly remove the lock using `exit`. Lock calls `Enter` and `Exit` implicitly, but when using `Monitor` they have to be called by the developer. It is best practice to call `exit` in a `finally` block to ensure the lock will be released in the event of an `Exception`.

```
Monitor.Enter(_locker);
try
```

```
{
   // Critical Section
}
finally
{
  Monitor.Exit(_locker);
```

Using `lock` is generally preferred over using `Monitor` directly. This is because lock is more concise and lock ensures that the underlying monitor is released, even if the protected code throws an `Exception`.

However, the `lock` keyword isn't quite as fully-featured as the `Monitor` class. For example, `Monitor` has a `TryEnter` method that can wait for a `lock` for a specified period of time instead of waiting infinitely.

Using a mutual exclusion lock

Locking is essential in parallel programs. It restricts code from being executed by more than one thread at the same time. Exclusive locking is used to ensure that only one thread can enter a particular section of code at a time.

The simplest way to use synchronization in c# is with the `lock` keyword. The `lock` keyword works by marking a block of code as a critical section by obtaining a mutual exclusion lock for an object running a statement and then releasing the lock.

In this recipe, we are going to create a class that represents a bank account. An object of this class will be shared by a couple of parallel tasks that will be making a series of withdrawals for random amounts. The critical section of code in the `Withdraw` method that updates the balance of the shared account object will be protected by a `lock` statement.

How to do it...

Let's go to Visual Studio 2012 and take a look at the following steps on how to use mutual exclusion locks:

1. Start a new project using the **C# Console Application** project template and assign `LockExample` as the **Solution name**.
2. Add a new class to the project and name the class `Account.cs`.
3. Add the following code snippet using directives to the top of your `Account` class:

   ```
   using System.Text;
   using System.Threading.Tasks;
   ```

4. Add a private field of type `double` to the `Account` class to store the balance of the account and a private object that will be used for locking.

```
private double _balance;
private object _locker = new object();
```

5. Next let's add a constructor to the `Account` class. This constructor should accept a parameter of type `double` and should initialize the `balance` field.

```
public Account(double initialBalance)
{
  _balance = initialBalance;
}
```

6. Now let's create a `Withdraw` method for the account. If the account has a negative balance, the `Withdraw` method should throw an error. Otherwise, the `Withdraw` method should obtain a mutual exclusion lock on the `Account` object and deduct the requested amount from the balance.

```
public double Withdraw(double amount)
{

  if (_balance < 0)
  throw new Exception("Account has a negative balance.");
  }

  lock (_locker)
  {
    if (_balance >= amount)
    {
      Console.WriteLine("Starting balance :  " + _balance);
      Console.WriteLine("Withdraw  amount : -" + amount);
      _balance = _balance - amount;
      Console.WriteLine("Current balance :  " + _balance);
      return amount;
    }
    else
    {
      return 0;
    }
  }
}
```

7. Now let's go back to our `Program` class. Make sure to add the following code snippet using directives that are at the top of the `Program` class:

```
using System;
using System.Threading.Tasks;
```

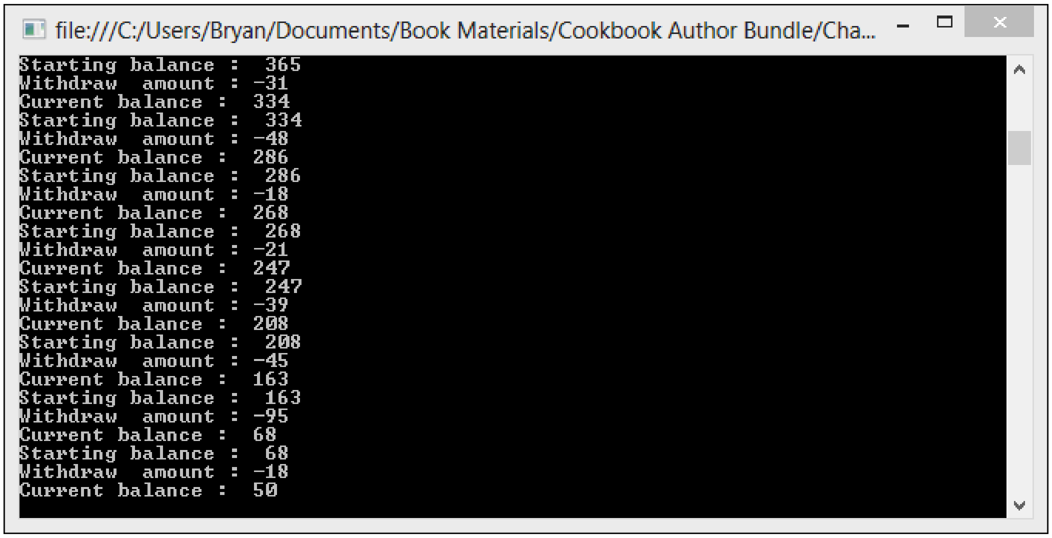
8. Create a static `DoTransactions` method for the `Program` class. The `DoTransactions` method should loop ten times doing a withdrawal of a random amount.

```
static void DoTransactions(Account account)
{
  Random r = new Random();
  for (int i = 0; i < 10; i++)
  {
    account.Withdraw((double)r.Next(1, 100));
  }
}
```

9. Finally, in the `Main` method of the `Program` class, let's create a shared account object and two tasks that will concurrently execute the withdrawals. Finish up by waiting for the user input before exiting.

```
static void Main(string[] args)
{
  Account account = new Account(1000);
  Task task1 = Task.Factory.StartNew(() =>
DoTransactions(account));
  Task task2 = Task.Factory.StartNew(() =>
DoTransactions(account));
  Console.ReadLine();
}
```

10. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

The `lock` keyword is a c# language shortcut for using the `System.Threading.Monitor` class. Basically, the `lock` keyword ensures that threads cannot enter a critical section of code while another thread is in the critical section; the following is the code contained in the scope of the `lock` statement:

> A critical section is simply a piece of code that accesses a shared resource that must not be concurrently accessed by more than one thread.

```
lock (this)
{
  //This is the critical section
}
```

If a thread tries to enter a locked section of code, it will block and wait until the locked object is released. The lock will be released when the locking thread exits the scope of the lock. The `lock` keyword calls `System.Threading.Monitor.Enter` at the start of the scope and `System.Threading.Monitor.Exit` at the end of the scope.

Notice that we created a private lockable object to lock on instead of locking on the instance of the `Account` class. This is the best practice. In general, you should avoid locking on a public type or on instances of objects that are beyond your code's control. If another programmer locks your class to synchronize their data, a deadlock can occur. A deadlock is a situation in which two or more competing threads are waiting for each other to finish their work and release a lock, and thus neither one ever does. Note also that locks can only be obtained on reference types.

Using SpinLock for synchronization

SpinLock is a special purpose lock that should only be used when lock contention is relatively rare and when lock-hold times are always very short. Unlike monitor and other lock types that work by using what is essentially a wait event; SpinLock sits in a loop and repeatedly checks until the lock becomes available. The SpinLock can be faster than a monitor lock because it reduces thread context switches. However, because the thread is spinning in a loop, a SpinLock can cause high CPU usage if locks are held for a long time.

In this recipe, we are going to revisit our bank account solution which will have a shared bank account object that will be updated by multiple tasks in parallel. Each of the tasks will have access to the shared account object and will manage concurrency using SpinLock.

How to do it...

Let's create a new Console Application and see how to use SpinLock for synchronization, by going through the following steps:

1. Start a new project using the **C# Console Application** project template and assign `SpinBasedLocking` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

   ```
   using System;
   using System.Collections.Generic;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. After the `Program` class, but inside the `SpinBasedLocking` namespace, create a very simple definition for an `Account` class. This class only needs to have a single `public integer` field for the balance.

   ```
   class Account
   {
     public int Balance { get; set; }
   }
   ```

4. Now, in the `Main` method of the `Program` class, let's start by creating the shared account object, a `SpinLock`, and a list to hold our tasks.

```
var account = new Account();
var spinLock = new SpinLock();
var taskList = new List<Task>();
```

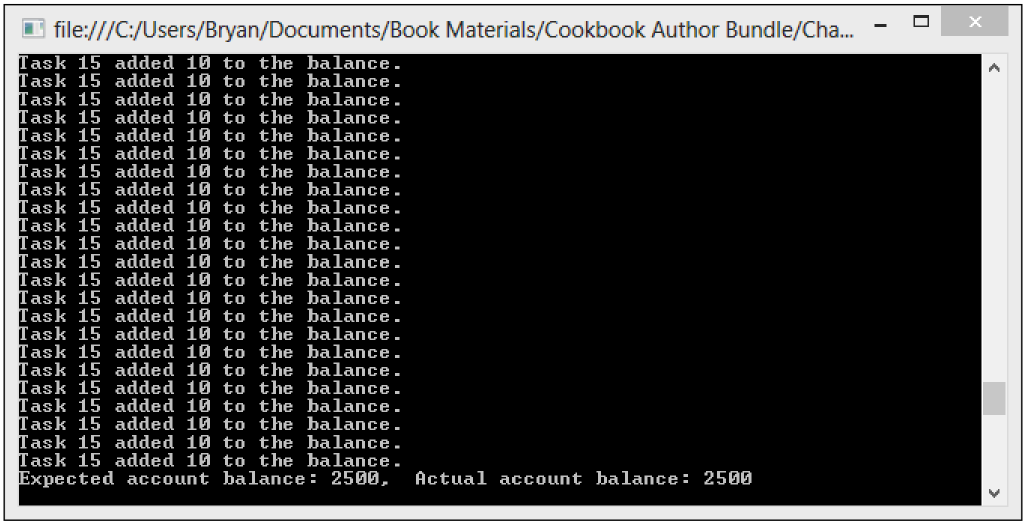
5. Next, let's add a `for` loop to the `Main` method that creates five tasks. Each task will loop several times updating the balance and using `SpinLock` to manage concurrent access. The `SpinLock` should be acquired in a `try` block and released in a `finally` block.

```
for (int i = 0; i < 5; i++)
{
  taskList.Add(Task.Factory.StartNew(() =>
  {
    for (int x = 0; x < 50; x++)
    {
      bool lockAquired = false;
      try
      {
        spinLock.Enter(ref lockAquired);
        Thread.Sleep(50);
        account.Balance = account.Balance + 10;
        Console.WriteLine("Task {0} added 10 to the balance.",
            Thread.CurrentThread.ManagedThreadId);
      }
      finally
      {
        if(lockAquired) spinLock.Exit();
      }
    }
  }));
}
```

6. Finish up the `Main` method by waiting for all of our tasks to complete and wait for the user input before exiting.

```
Task.WaitAll(taskList.ToArray());
Console.WriteLine("Expected account balance: 2500,  Actual account
balance:      {0}", account.Balance);
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output, similar to the following screenshot:



How it works...

A SpinLock can be useful to avoid blocking in our applications, but if we expect a large amount of blocking, we probably shouldn't use SpinLock, as excessive spinning could make the situation much worse. However, if the critical section performs a very minimal amount of work and the wait times for the lock are minimal, then spin locking could be a good choice.

The `enter` method of `SpinLock` takes a `Boolean` parameter that indicates if the lock was successfully taken. Even in the case of an `Exception`, we can examine the `Boolean` parameter to reliably determine if the lock was successfully taken.

```
bool lockAquired = false;
try
{
  spinLock.Enter(ref lockAquired);
  //Critical section
}
finally
{
  if(lockAquired) spinLock.Exit();
}
```

The `SpinLock` should be exited in a `finally` block to ensure that the locked is released. You should also use the `Boolean` parameter to check if the lock is actually held before exiting because calling `exit` on a lock that isn't held will produce an error.

Interlocked operations

Locking is fairly safe in most of the cases but there are cases when locking may not be the safe solution. Some of these cases are incrementing or decrementing the value of a variable, adding to, or subtracting from a variable, and exchanging two variables with each other. These operations seem like atomic operations, but actually are not.

For example, increment and decrement operations include three steps. The first is loading the value of from the variable to a register, the second is incrementing the value of variable, and the third is storing the incremented value back in the variable.

The problem is that a thread can be pre-empted after the first two steps and another thread can start execution before the incremented value of the variable is saved back in the variable from the register. In the meantime, a second thread can go ahead and execute all three steps. After that, the first thread executes the third step and overwrites the value of the counter variable. Now the operation that was executed by the second thread is lost.

So how do can we avoid this scenario? This is where interlocking comes in. The `Interlock` class provides members that can be used to increment/decrement values, and exchange data between two variables. The `Interlock` class provides atomic operations in variables that are shared among multiple concurrent threads.

In this recipe, we are going to create an application that has a bank account object that will be updated by multiple tasks in parallel. Each of the tasks will have access to the shared account object, which has only a public field for the balance. The tasks will use `Interlocked.Add` to update the account balance as an atomic operation.

How to do it...

Now, let's take a look at the following steps on how to use interlocked in a Console Application:

1. Start a new project using the **C# Console Application** project template and assign `InterlockedExample` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `InterlockedExample` namespace:

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;
```

3. After the `Program` class, but inside the `InterlockedExample` namespace, create a very simple definition for an `Account` class. This class only needs to have a single `public integer` field for the balance.

```
class Account
{
  public int Balance = 0;
}
```

4. Now, in the `Main` method of the `Program` class, let's start by creating the shared account object and a list of `Task` to hold our tasks.

```
static void Main(string[] args)
{
  var account = new Account();
  var taskList = new List<Task>();
}
```

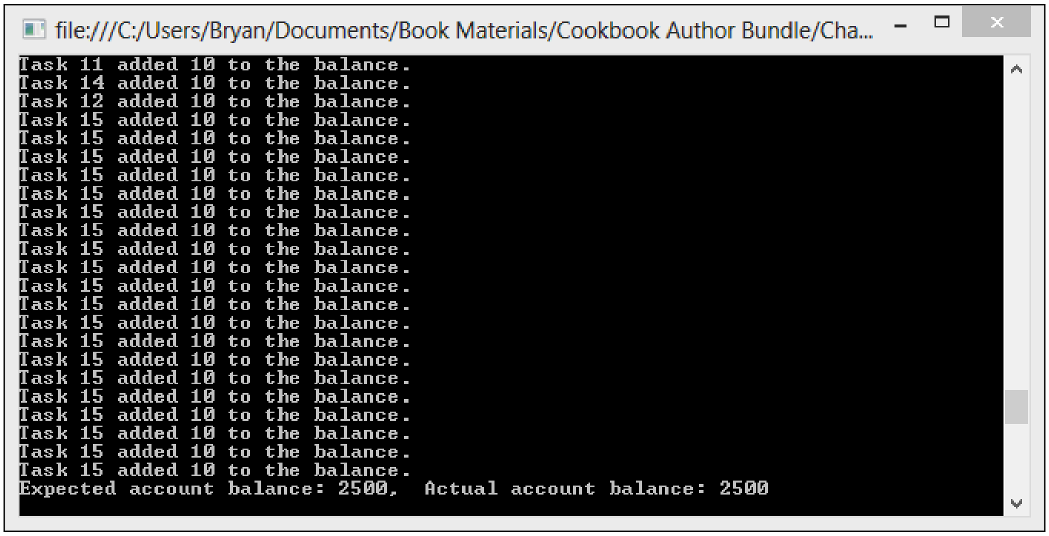
5. Next, let's add a `for` loop to the `Main` method that creates five tasks. Each task will loop several times and use `Interlocked.Add` to increase the balance of the `Account` object as an atomic operation.

```
for (int i = 0; i < 5; i++)
{
  taskList.Add(Task.Factory.StartNew(() =>
  {
    for (int x = 0; x < 50; x++)
    {
      Thread.Sleep(50);
      Interlocked.Add(ref account.Balance, 10);
      Console.WriteLine("Task {0} added 10 to the balance.",
        Thread.CurrentThread.ManagedThreadId);
    }
  }));
}
```

6. Finish up the `Main` method by waiting for all of our tasks to complete and wait for the user input before exiting.

```
Task.WaitAll(taskList.ToArray());
Console.WriteLine("Expected account balance: 2500,
    Actual account balance: {0}", account.Balance);
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see an output similar to the following screenshot:



How it works...

Interlocked can be used instead of a locking mechanism to provide simpler and fast operations on shared variables.

The usage of interlocked is very simple. You just use its `static` method to automatically, add to, subtract from, increment, decrement, or exchange values in a variable.

```
Interlocked.Add(ref account.Balance, 10);
```

These `static` methods change the math operations to be atomic. This means no other operations can be performed on the value during the call, and the operation won't be affected by context switching between threads.

Synchronizing multiple tasks with a Barrier

When you need some tasks to perform a series of parallel phases, and each phase needs to start, after all other tasks complete the previous phase, you can synchronize and coordinate this work using a barrier. In short, a barrier prevents individual tasks from continuing until all tasks reach the barrier.

Each task in the group is referred to as a participant, and signals that it has reached the barrier in each phase and is waiting for all the other participants to signal their arrival at the barrier before continuing. Optionally, you can also specify a time out to avoid the deadlock that will occur if one task fails to reach the barrier.

In this recipe, we are going to create a Console Application that has four participant tasks that execute a method with a `for` loop. Each iteration of the loop is a phase controlled by the barrier. The tasks will signal when they have reached the barrier and wait for all of the other tasks before continuing.

How to do it...

Let's create a new Console Application and take a look at the following steps on how to synchronize tasks with Barrier:

1. Start a new project using the **C# Console Application** project template and assign `Barrier` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. First, in the `Program` class, let's use a `static` method called `OperationWithBarrier` that accepts a parameter of a `Barrier` object.

```
static void OperationWithBarrier(Barrier barrier)
{

}
```

4. Now, in the body of the `OperationWithBarrier` method, let's create a `for` loop that loops three times. In each loop, get the `threadId` of the executing thread and then signal that the thread has reached the barrier.

```
for (int i = 0; i < 3; ++i)
{
  var threadId = Thread.CurrentThread.ManagedThreadId;
  Console.WriteLine("Thread {0} has reached wait.", threadId);
  barrier.SignalAndWait(100);
  Console.WriteLine("Thread {0} after wait wait.", threadId);
}
```

5. Back in the `Main` method of the `Program` class, let's create a `Barrier` object that has four participants and a post-phase action that writes to the Console when each barrier phase is reached.

```
var barrier = new Barrier(4, (b) =>
  Console.WriteLine("Barrier phase {0} reached.",
b.CurrentPhaseNumber));
```

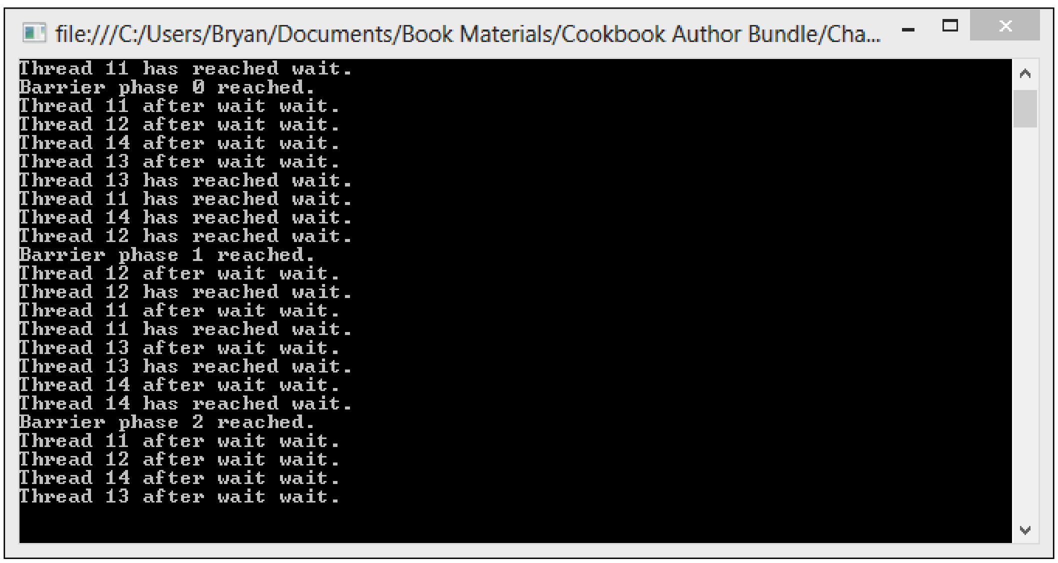
6. Now let's start four new tasks, each of which runs `OperationWithBarrier` passing in the `Barrier` object we just created at the parameter.

```
var task1 = Task.Factory.StartNew(() =>
OperationWithBarrier(barrier));
var task2 = Task.Factory.StartNew(() =>
OperationWithBarrier(barrier));
var task3 = Task.Factory.StartNew(() =>
OperationWithBarrier(barrier));
var task4 = Task.Factory.StartNew(() =>
OperationWithBarrier(barrier));
```

7. Finally, let's wait for all of our tasks to complete and wait for user input before exiting.

```
Task.WaitAll(task1, task2, task3, task4);
Console.ReadLine();
```

8. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

When you create a `Barrier`, specify the number of participants; in our case four. The `Barrier` constructor also has an overload that allows you to specify a post phase action of type `Action<Barrier>`. This action will fire after all the four tasks signal have reached the barrier.

```
Barrier barrier = new Barrier(4, (b) =>
  Console.WriteLine("Barrier phase {0} reached.",
b.CurrentPhaseNumber));
```

Each of the four tasks signals reach the barrier and wait for the other tasks by calling the `Barrier.SignalAndWait` method.

```
barrier.SignalAndWait(100);
```

A deadlock will occur if one participant's task fails to reach the barrier because the tasks that reach the barrier will wait indefinitely for the fourth call to `Barrier.SignalAndWait`. To avoid these deadlocks, we used one of the overloads of the `SignalAndWait` method that specifies a time out. After the time out the remaining tasks are free to continue to the next phase.

Using ReaderWriterLockSlim

The `ReaderWriterLockSlim` class is used to protect a resource that is read by multiple threads and written to by one thread at a time.

`ReaderWriterLockSlim` allows a thread to enter one of the following three modes:

- **Read mode**: Allows multiple threads to enter the lock as long as there is no thread currently holding a write lock or waiting to acquire a write lock. If there are any threads that are holding or waiting for a write lock, the threads waiting to enter in read mode are blocked.
- **Upgradeable mode**: Intended for cases where a thread usually performs reads and might also occasionally perform writes.
- **Write mode**: Only one thread can be in write mode at a time. A thread waiting to enter the lock in write mode will block if there is a thread currently holding a lock in write mode or waiting to enter write mode. If there are threads in read mode, the thread that is upgrading to write mode will block.

In this recipe, we are going to build a Console Application that creates a writer task to write numeric values to an array. The application will also start up three reader tasks that read the values that were written to the array and append the values to a string using `StringBuilder`.

How to do it...

Now, let's see how to use `ReaderWriterLockSlim` by having a look at the following steps:

1. Start a new project using the **C# Console Application** project template and assign `ReaderWriter` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

3. Let's start by creating a `static` method on the `Program` class which our writer task will call to write values to an array. The `write` method will ask to enter the lock in write mode and will loop a few times writing the square of the loop index to the array.

```
static void Write()
{
  int id = Thread.CurrentThread.ManagedThreadId;
  for (int i = 0; i < MaxNumberValues; ++i)
  {
    _lock.EnterWriteLock();
    Console.WriteLine("Entered WriteLock on thread {0}", id);
    _array[i] = i*i;
    Console.WriteLine("Added {0} to array on thread {1}",
        _array[i], id);

    Console.WriteLine("Exiting WriteLock on the thread {0}",
        id);
    _lock.ExitWriteLock();
    Thread.Sleep(1000);
  }
}
```

4. Now let's create another `static` method on the `Program` class called `Read` that our reader tasks will use to read the values from the array and append the values to a string using `StringBuilder`. The `Write` method will request a reader lock and loop through the values of the array, writing the values to the output string.

```
static void Read()
{
  int idNumber = Thread.CurrentThread.ManagedThreadId;
  for (int i = 0; i < MaxNumberValues; ++i)
```

```
    {
      _lock.EnterReadLock();
      Console.WriteLine("Entered ReadLock on the thread {0}",
                idNumber);
      StringBuilder sbObj = new StringBuilder();
      for (int j = 0; j < i; j++)
      {
        if (sbObj.Length > 0) sbObj.Append(", ");
        sbObj.Append(_array[j]);
      }
      Console.WriteLine("Array: {0} on the thread {1}", sbObj,
                idNumber);
      Console.WriteLine("Exiting the ReadLock on thread {0}",
                idNumber);
      _lock.ExitReadLock();
      Thread.Sleep(2000);
    }
  }
```

5. Next, at the top of the `Program` class, let's create a constant for the maximum number of values and a couple of static fields for the array and the lock.

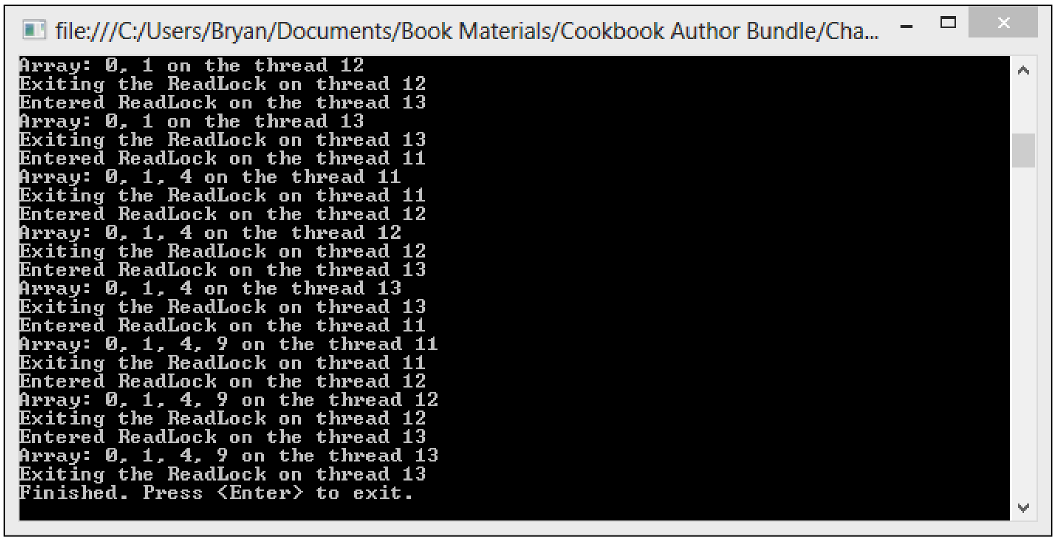
```
const int MaxNumberValues = 5;
static int[] _array = new int[MaxNumberValues];
static ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();
```

6. Now we need to create the `Main` method of our `program` class. The `Main` method will have a list of tasks that we can use to wait. We will need to create a single writer task that calls the `write` method and will do a loop to create three reader tasks which will call the `reader` method. Finish up by waiting for the user input before exiting.

```
static void Main(string[] args)
{
  var taskList = new List<Task>();
  taskList.Add(Task.Factory.StartNew(() => Write()));
  Thread.Sleep(1000);
  for (int i = 0; i < 3; i++)
  {
    taskList.Add(Task.Factory.StartNew(()=>Read()));
  }
  Task.WaitAll(taskList.ToArray());
```

```
    Console.WriteLine("Finished. Press <Enter> to exit.");
    Console.ReadKey();
  }
```

7. In Visual Studio 2012, press *F5* to run the project. Notice the ordered results in the following screenshot:



How it works...

`ReaderWriterLockSlim` allows multiple threads to be in read mode; allows one thread to be in write mode with an exclusive ownership of the lock, and allows one thread that has read access to be in upgradeable read mode.

A thread can enter the lock in three modes: read mode, write mode, and upgradeable read mode. In our Console Application, the `Write` method requests to enter the lock in write mode by calling the `EnterWriteLock` method and the `Read` method enters the read mode by calling the `EnterReadLock` method.

Only one thread can be in write mode at any time. When a thread is in write mode, no other thread can enter the lock in any mode. So, any of our reader tasks will block if a writer lock is currently held. Once our writer task releases the write lock by calling the `ExitWriteLock` method, multiple reader tasks will be able to obtain a read lock and enter the critical section.

Using WaitHandles with Mutex

A **Mutex** is like a lock, but it can work across multiple processes. A Mutex is a synchronization primitive that can also be used for inter-process synchronization. When two or more threads need to access a shared resource at the same time, the system needs a synchronization mechanism to ensure that only one thread at a time uses the resource. Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one thread. If a thread acquires a Mutex, the second thread that wants to acquire that Mutex is suspended until the first thread releases the Mutex.

In this recipe, we are going to return to our bank account example and build a Console Application that creates several tasks to update the balance on a shared bank account object. The tasks will use a Mutex to provide access to the balance for a single task at a time.

How to do it...

Let's create a new Console Application and see how to use Mutex by having a look at the following steps:

1. Start a new project using the **C# Console Application** project template and assign `MutexExample` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

   ```
   using System;
   using System.Collections.Generic;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. After the `Program` class, but inside the `MutexExample` namespace, create a very simple definition for an `Account` class. This class only needs to have a single `public integer` field for the balance.

   ```
   class Account
   {
     public int Balance { get; set; }
   }
   ```

4. Now, in the `Main` method of the `Program` class, let's start by creating the shared account object, a `Mutex`, and a list of `Task` to hold our tasks.

   ```
   var account = new Account();
   var mutex = new Mutex();
   var taskList = new List<Task>();
   ```

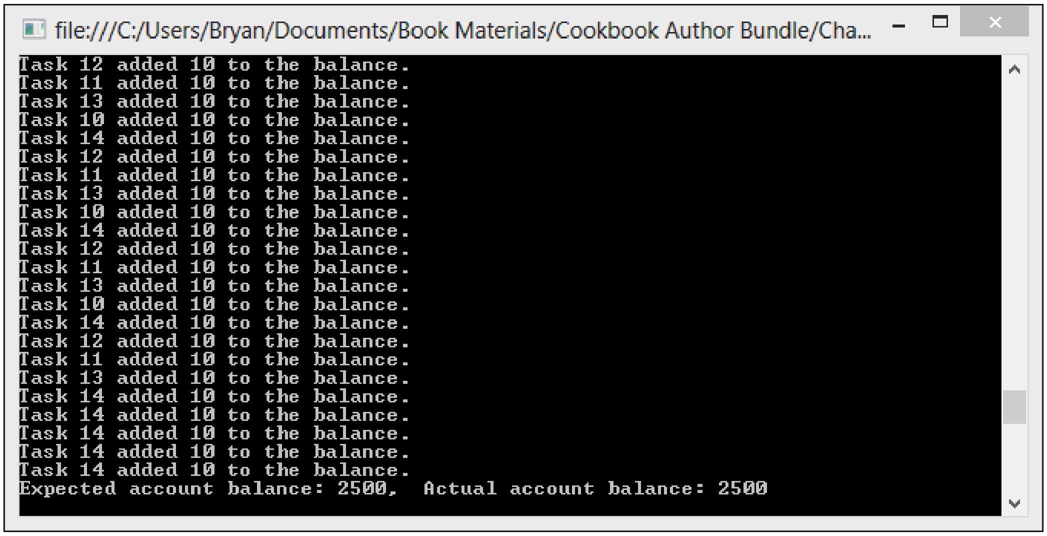
5. Next, let's add a `for` loop to the `Main` method that creates five tasks. Each task will loop several times updating the balance and using `Mutex` to manage concurrent access. The `Mutex` should be acquired in a `try` block and released in a `finally` block.

```
for (int i = 0; i < 5; i++)
{
  taskList.Add(Task.Factory.StartNew(() =>
  {
    for (int x = 0; x < 50; x++)
    {
      bool lockAquired = false;
      try
      {
        lockAquired = mutex.WaitOne();
        Thread.Sleep(50);
        account.Balance = account.Balance + 10;
        Console.WriteLine("Task {0} added 10 to the balance.",
          Thread.CurrentThread.ManagedThreadId);
      }
      finally
      {
        if (lockAquired) mutex.ReleaseMutex();
      }
    }
  }));
}
```

6. Finish up the `Main` method by waiting for all of our tasks to complete and wait for the user input before exiting.

```
Task.WaitAll(taskList.ToArray());
Console.WriteLine("Expected account balance: 2500,
      Actual account balance: {0}", account.Balance);
Console.ReadLine();
```

7. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

Basically, a Mutex is a mechanism that acts as a flag to prevent two threads from performing one or more actions simultaneously.

With a `Mutex` class, you call the `WaitHandle.WaitOne` method to lock. The `WaitOne` method takes a `Boolean` parameter that indicates if the lock was successfully taken. Even in the case of an `Exception`, you can examine the `Boolean` parameter to reliably determine if the lock was successfully taken.

```
bool lockAquired = false;
try
{
  lockAquired = mutex.WaitOne();
  //Critical section
}
finally
{
  if(lockAquired) mutex.ReleaseMutex();
}
```

Closing or disposing a `Mutex` automatically releases it. Just as with the lock statement, a Mutex can be released only from the same thread that obtained it. The `Mutex` should be released in a `finally` block to ensure that the locked is released. You should also use the `Boolean` parameter to check if the `Mutex` is actually held before exiting because calling `ReleaseMutex` on a `Mutex` that isn't held will produce an error.

Waiting for multiple threads with CountdownEvent

A common asynchronous pattern is the pattern known as fork/join parallelism. This typically manifests by starting a number of pieces of work and later joining with that work.

A `CountdownEvent` is initialized with a count. Threads can block waiting on the event until the count reaches `0`, at which point the `CountdownEvent` will be set and the threads can proceed.

In this recipe, we will create a Console Application that performs some simulated work in a loop. We will initialize a `CountdownEvent` to a small number of tasks, and then start simulating the work with the specified number of tasks. Each task will decrement the `CountDownEvent`. When the `CountDownEvent` reaches `0` and is signaled, we will reset the `CountDownEvent` with a higher count and start over until we reach the maximum number of tasks.

How to do it...

Now, let's take a look at using `CoundownEvent` to wait for multiple threads. Have a look at the following steps:

1. Start a new project using the **C# Console Application** project template and assign `ForkAndJoin` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

   ```
   using System;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. At the beginning of your `Program` class, start by creating a static variable for the `CountdownEvent` and a couple of constants for the number of tasks we want to start with and the number of tasks we want to finish with.

   ```
   private static CountdownEvent _countdownEvent;
   private const int BEGIN_TASKS = 2;
   private const int END_TASKS = 6;
   ```

4. At the bottom of the `Program` class, after the `Main` method, create a new `static` method called `SimulateWork`. This method will take an `integer` parameter which represents the number of tasks to create. The method will then loop to create the number of tasks specified. The tasks will just sleep for a bit and write a message to the Console. When the tasks are finished executing, to call the `Signal` method of the `CountdownEvent` to decrement the count.

```
private static void SimulateTasks(int taskCount)
{
  for (int i = 0; i < taskCount; i++)
  {
    Task.Factory.StartNew((num) =>
      {
        try
        {
          var taskNumber = (int)num;
          Thread.Sleep(2500);
          Console.WriteLine("Task {0} simultated.",
              taskNumber);
        }
        finally
        {
          _countdownEvent.Signal();
        }
      },i);
  }
}
```

5. In the `Main` method of your `Program` class, start with instantiating the `CountdownEvent` object. Pass in the `Begin_Tasks` constant to the `CountdownEvent` constructor so that the event will be signaled after two tasks.

```
_countdownEvent = new CountdownEvent(BEGIN_TASKS);
```

6. Next, in the `Main` method, create a task that executes a `for` loop. Each iteration of the loop should reset the `CountdownEvent` to the number of tasks we want to wait for. Then the task will call the `SimulateWork` method and wait for the tasks to finish by calling the `Wait` method of `CountdownEvent`.

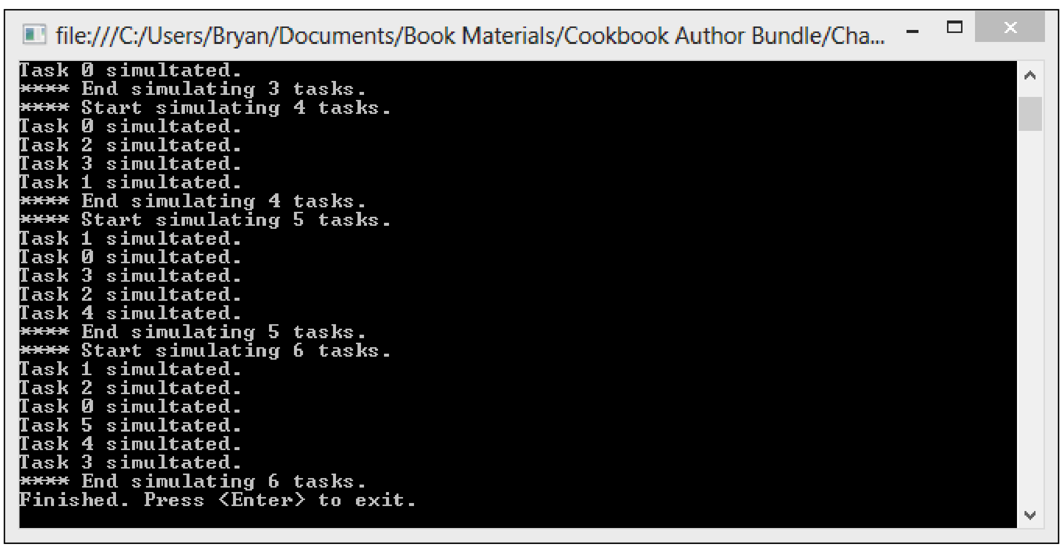
```
var task1 = Task.Factory.StartNew(() =>
{
  for (int i = BEGIN_TASKS; i <= END_TASKS; i++)
  {
    Console.WriteLine("**** Start simulating {0} tasks.", i);
    _countdownEvent.Reset(i);
```

```
      SimulateTasks(i);
      _countdownEvent.Wait();
      Console.WriteLine("**** End simulating {0} tasks.", i);
    }
});
```

7. Finish up the `Main` method by waiting for the previous task to complete in a `try` block and disposing of the `CountdownEvent` in a `finally` block. Wait for the user input before exiting.

```
try
{
  task1.Wait();
  Console.WriteLine("Finished. Press <Enter> to exit.");
}
finally
{
  _countdownEvent.Dispose();
}
Console.ReadLine();
```

8. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

The main feature of `CoutndownEvent`, as you have already seen, is that it can be used to signal when several tasks have been completed.

The constructor for `CountDownEvent` accepts an `integer` parameter to specify the initial count of signals that we want to wait for before triggering the event. In our case, we passed in a constant value that is equal to two.

```
_countdownEvent = new CountdownEvent(BEGIN_TASKS);
```

The number of events we are waiting for can be reset by calling the `Reset` method as we have done in our `for` loop. Each iteration of the `for` loop increases the number of events we are waiting for, up to the maximum number which we specified in another constant.

```
for (int i = BEGIN_TASKS; i <= END_TASKS; i++)
{
  Console.WriteLine("**** Start simulating {0} tasks.", i);
  _countdownEvent.Reset(i);
  SimulateTasks(i);
  _countdownEvent.Wait();
  Console.WriteLine("**** End simulating {0} tasks.", i);
}
```

After calling the `SimulateWork` method with the desired number of tasks to spin up, we wait for the tasks to complete by calling the `WaitMethod` on the `CountdownEvent`.

Finally, in the `SimulateWork` method, each task indicates that it has completed and decrements the count of the signals we are waiting for by calling the `Signal` method of `CountDownEvent`.

```
Task.Factory.StartNew((num) =>
{
  try
  {
    var taskNumber = (int)num;
    Thread.Sleep(2500);
    Console.WriteLine("Task {0} simultated.", taskNumber);
  }
  finally
  {
    _countdownEvent.Signal();
  }
},i);
```

Using ManualResetEventSlim to spin and wait

`ManualResetEventSlim`, a light-weight synchronization primitive that was introduced in .NET Framework 4.0, allows threads to communicate with each other by signaling.

When a task begins an activity that it must complete before other tasks proceed, it calls `Reset` to put `ManualResetEventSlim` in the non-signaled state. This thread can be thought of as controlling the reset event. Tasks that call the `Wait` method of `ManualResetEventSlim` will block, awaiting the signal. When the controlling thread completes the activity, it calls `Set` to signal that the waiting threads can proceed. All waiting threads are then released.

In this recipe, the main application thread will create a `ManualResetEventSlim` to coordinate five tasks that it spins up. The tasks will call the `Wait` method of the reset event and wait for the event to be signaled. After sleeping for a bit, the main thread will wake up and call the `Set` method of the reset event to release all of the other tasks to proceed.

How to do it…

Now, let's take a look at how to use `ManualResetEventSlim` by going through the following steps:

1. Start a new project using the **C# Console Application** project template and assign `SpinAndWait` as the **Solution name**.
2. Add the following code snippet using directives to the top of your `Program` class:

   ```
   using System;
   using System.Threading;
   using System.Threading.Tasks;
   ```

3. First, let's create a `static` method on the `Program` class that will use a `for` loop to create and start five tasks. Each task will sleep for two seconds, write a message to the Console, and wait for the main thread to set the `ManualResetEventSlim`.

   ```
   private static void StartTasks()
   {
     for (int i = 0; i < 5; i++)
     {
       Task.Factory.StartNew(()=>
       {
         Thread.Sleep(2000);
   ```

```
            Console.WriteLine("Task {0} waiting for event...",
              Thread.CurrentThread.ManagedThreadId);
            resetEvent.Wait();
            Console.WriteLine("Task {0} event signalled",
              Thread.CurrentThread.ManagedThreadId);
          } );
        }
      }
```

4. Now, in the `Main` method, create a `ManualResetEventSlim` object, and call the `StartTasks` method.

```
resetEvent = new ManualResetEventSlim(false);
StartTasks();
```

5. Now, put the main thread to sleep for a second and then call the `Set` method of the reset event object to the tasks that are waiting.

```
Thread.Sleep(1000);
Console.WriteLine("Main thread setting event");
resetEvent.Set();
```

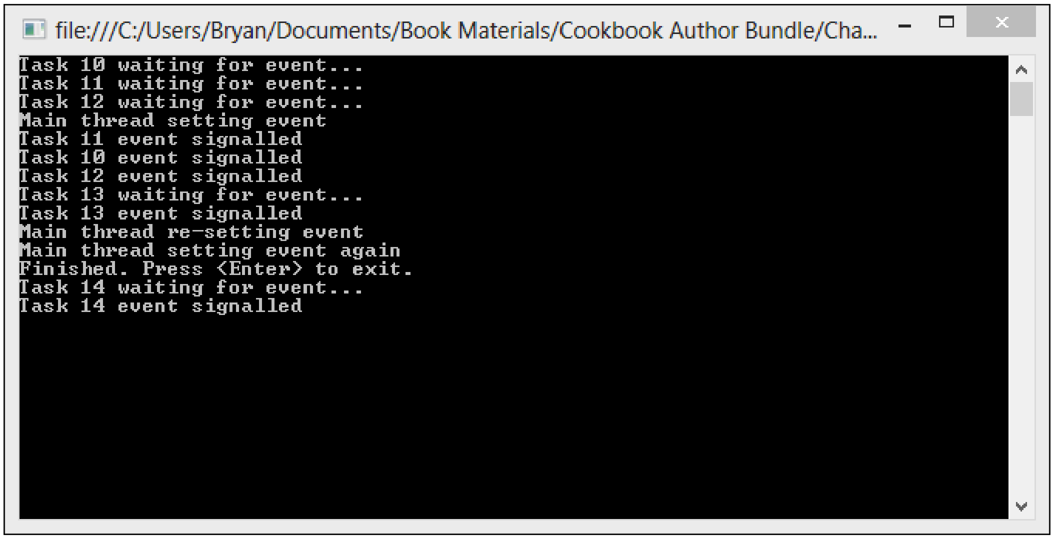
6. Next, sleep the main thread for `500` ms and then call the `Reset` method of the reset event object to stop any more tasks from continuing.

```
Thread.Sleep(500);
Console.WriteLine("Main thread re-setting event");
resetEvent.Reset();
```

7. Finally, sleep the main thread for another second, the call the `Set` method of the reset event to release the waiting tasks. Wait for the user input before exiting.

```
Thread.Sleep(1000);
Console.WriteLine("Main thread setting event again");
resetEvent.Set();

Console.WriteLine("Finished. Press <Enter> to exit.");
Console.ReadLine();
```

8. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

`ManualResetEventSlim` functions like a gate. When you call the `Set` method, you open the up the gate, allowing any tasks that have called the `Wait` method to start running. Calling `Reset` closes the gate and any task that calls `Wait` will block, waiting for the event to be set. When the gate is next opened, they will all be released again at once.

The code is pretty simple. We create five tasks and each of the tasks call the `Wait` method of the reset event object and block, waiting for the event to be set.

```
Task.Factory.StartNew(()=>
{
  ...
  resetEvent.Wait();
  ...
} );
```

The main thread then goes to sleep for a second, releasing the currently blocking tasks to run. Then we sleep for a bit more, reset, and set the event again, releasing the remaining tasks.

`ManualResetEventSlim` is optimized for short waiting times and has the ability to opt into spinning for a set number of iterations. It also has a more efficiently managed implementation and allows a `Wait` to be cancelled via a `CancellationToken`.

Using SemaphoreSlim to limit access

A semaphore works by keeping a counter. Each time a thread obtains the semaphore, the counter is reduced and each time the thread returns the semaphore, it is increased.

`SemaphoreSlim` is a lightweight semophore that limits the number of threads that can access a resource or resources concurrently. A task that calls the `Wait` method of a `SemaphoreSlim` object will block until the semaphore counter is below the number of requests the semaphore can grant, which is specified in the constructor.

In this recipe, the main application thread will use a `for` loop to create five tasks. Each of the tasks call a method that waits in a `SemaphoreSlim` object before allowing access to a simulated shared resource.

How to do it...

Let's finish up the chapter by seeing how to use `SemaphoreSlim` to limit access to a shared resource. Have a look at the following steps:

1. Start a new project using the **C# Console Application** project template and assign `SemaphoreSlimExample` as the **Solution name**.
2. Add the following code snippet using directives to the top of your program class:

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

3. First, let's create a `static` method on the `Program` class that each of our threads will call in order to access a simulated shared resource. This method will call the `Wait` method of a `SemaphoreSlim` object, which will only grant access to three tasks at a time.

```
static void Enter(object id)
{
  Console.WriteLine("Task {0} is trying to enter.",id);
  _semaphoreSlim.Wait();
  Console.WriteLine("Task {0} has entered.", id);
  Thread.Sleep(2000); //Shared resource
  Console.WriteLine("Task {0} is leaving.", id);
  _semaphoreSlim.Release();
}
```

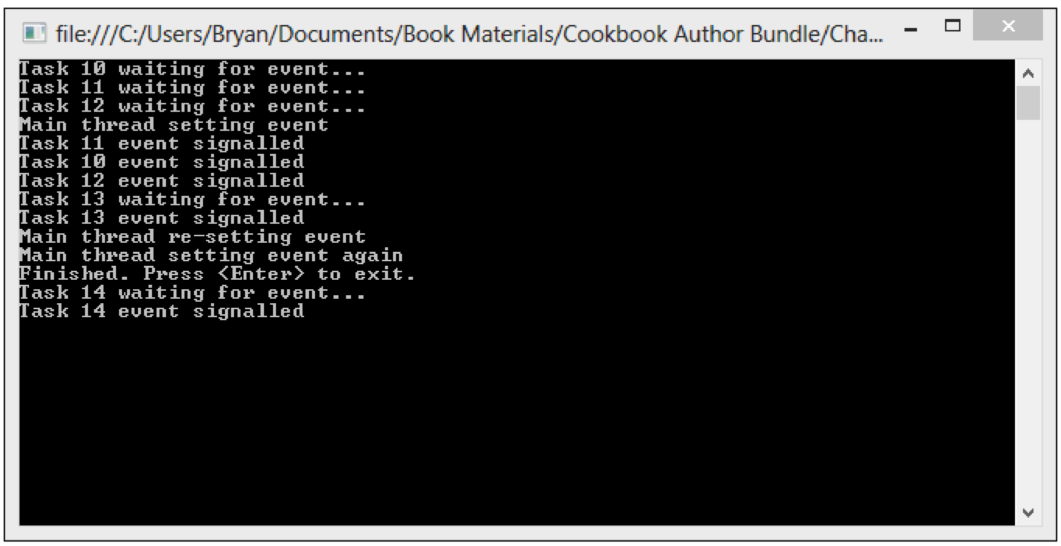
4. Now, previously to the `Main` method, create a static `SemaphoreSlim` object field on the `Program` class that will grant three access requests.

```
static SemaphoreSlim _semaphoreSlim = new SemaphoreSlim(3);
```

5. The `Main` method just needs to create and start five tasks in a `for` loop. The tasks only need to call the `Enter` method. Now, wait for the user input before exiting.

```
for (int i = 1; i <= 5; i++)
{
  Task.Factory.StartNew((num) =>
    {
      Enter(num);
    }, i);
}
Console.ReadLine();
```

6. In Visual Studio 2012, press *F5* to run the project. You should see output similar to the following screenshot:



How it works...

A semaphore is an enforcement of access limitation to a shared resource. Once it's full, no more tasks can enter the semaphore until one or more tasks complete and get terminated. A queue builds up outside for other tasks. Then, for each task that leaves the semaphore, another enters from the queue.

Using `SemaphoreSlim` is an easy two-step process. First you need to create a `SemaphoreSlim` object that all of your threads have visibility to. Use the constructor to specify the number of requests the semaphore can grant concurrently.

```
static SemaphoreSlim _semaphoreSlim = new SemaphoreSlim(3);
```

Before accessing a shared resource, call the `Wait` method on the semaphore method. Execution will continue into the shared resource for the specified number of tasks. All other tasks will block until one of the current tasks exits. Exiting tasks release the semaphore and decrement the request count by calling the `Release` method.

```
static void Enter(object id)
{
     _semaphoreSlim.Wait();
    //Shared resource
    _semaphoreSlim.Release();
}
```

7
Profiling and Debugging

In this chapter, we will cover the following recipes:

- Using the Threads and Call Stack windows
- Using the Parallel Stacks window
- Watching values in a thread with Parallel Watch window
- Detecting deadlocks with the parallel tasks window
- Measuring CPU utilization with Concurrency Visualizer
- Using Concurrency Visualizer Threads view
- Using Concurrency Visualizer Cores view

Introduction

Parallel programming can create complex problems. Maybe you didn't get the performance gain you expected from parallelizing your application. It could even be running slower that a sequential version of the same algorithm. Maybe you are getting consistently or occasionally incorrect results.

The problems that can occur in a parallel program are numerous and can be hard to detect. Perhaps oversubscription is causing poor performance because of the high number of context switches. Maybe you have inadvertently created a lock convoy, which is a condition that occurs when multiple threads of equal priority contend repeatedly for the same lock, and can lead to significant lock contention and serialization of the program even though multiple threads are in use.

In this chapter, we are going to take a look at the Visual Studio 2012 debugging features for multi-threaded applications, and how to use those features to solve concurrency related issues.

Using the Threads and Call Stack windows

When we want a thread-centric view of our application, the Threads window is the place to start. We can use the Threads window to see the location of all of our threads, see the thread call stack, and more. We can use the Call Stack window to view the stack frames of our application, or the function, or procedure calls that are currently on the stack.

In this recipe, we are going to see how to use the Threads and Call Stack windows in Visual Studio 2012 to view the call stack information for the threads in our application.

Getting ready…

Before we start looking at the debugging features of Visual Studio 2012, we need an application to debug. Let's create a `Console` application that spins up a few tasks so we can take a look at their call stack information.

1. Start a new project using the **C# Console Application** project template and assign `LockExample` as the **Solution name.**
2. Add the following `using` directives to the top of your `Program` class.

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
```

3. Let's start by creating a few `static` methods on the `Program` class. Add a method named `Method1` that loops three times, creating tasks. The tasks just need to call `Method2` with an `integer` parameter.

```
static void Method1()
{
  Console.WriteLine("In Method1.");
  for (int i = 0; i < 3; i++)
  {
    Task.Factory.StartNew(index => Method2((int)index), i);
  }
}
```

4. Now create a method named `Method2`. `Method2` just adds a random number to the parameter, writes the parameter to `Console`, and calls `Method4`.

```
static void Method2(int number)
{
  Random rnd = new Random();
  var sum = number + rnd.Next(1,10);
  Console.WriteLine("In Method2. Value:{0}", sum);
  Method4(sum);
}
```

5. Next create `Method3`, which just starts a third task that calls `Method2`.

```
static void Method3()
{
  Console.WriteLine("In Method3.");
  for (int i = 0; i < 3; i++)
  {
    Task.Factory.StartNew(() =>
    {
      Task.Factory.StartNew(index => Method2((int)index), i);
    });
    Thread.Sleep(10);
  }

}
```

6. Lastly, create `Method4` which contains our breakpoint.

```
static void Method4(int number)
{
  Console.WriteLine("In Method4.", number);
  Debugger.Break();
}
```

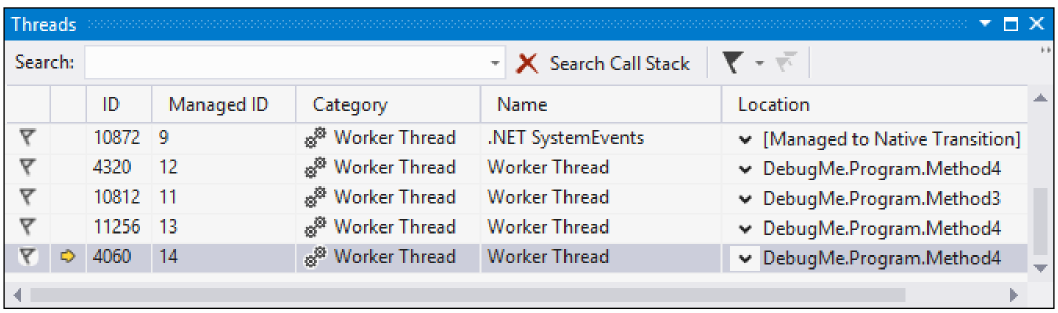
7. In the `Main` method, create a task that calls `Method1` and `Task` that calls `Method3`. Wait for the user input before exiting.

```
static void Main()
{
  var task1 = Task.Factory.StartNew(() => Method1());
  var task2 = Task.Factory.StartNew(() => Method3());
  Console.ReadLine();
}
```

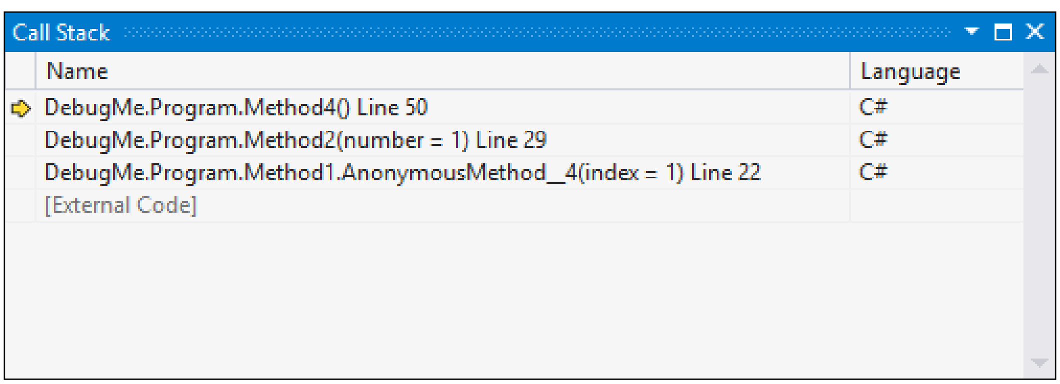
How to do it...

Let's start a debugging session and take a look at the window.

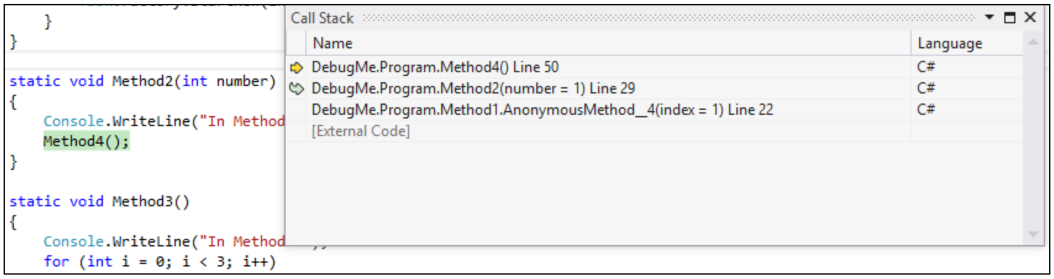
1. In Visual Studio 2012, press *F5* to run the project.
2. When the debugger hits the `Debug.Break` statement, go to the **Debug** menu of Visual Studio 2012, and click on **Windows**, and click on **Call Stack** to view the **Threads** window.
3. The active thread is the thread that is currently selected in the **Threads** window, indicated by the arrow icon. By default, the active thread is the one that hits the breakpoint. Switch the active thread via the **Threads** window by double-clicking on a different thread in the view.



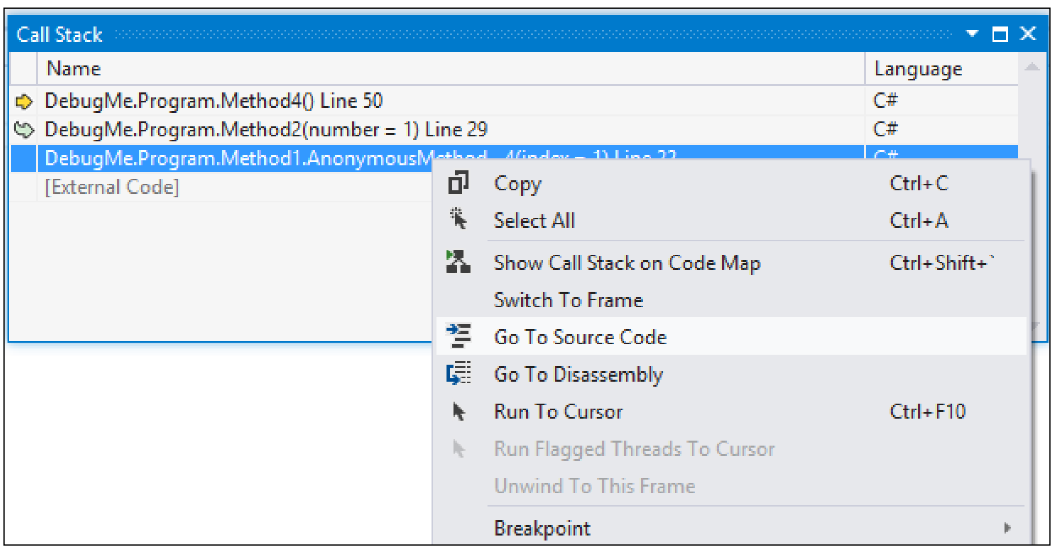
4. Go back to the **Debug** menu, click on **Windows,** and click on **Threads** to show the **Call Stack** window.
5. The **Call Stack** window indicates the top of stack of the active thread with a arrow icon. This is known as the active stack frame. When switching threads, the active stack frame changes. When execution resumes, the execution continues from the active stack frame onwards.



6. The current stack frame is the stack frame that drives the rest of the debugger tools and windows. Change the current stack frame by double-clicking on a different entry in the **Call Stack** window. When changing the current stack frame to be something other than the active stack frame, it shows a tapered arrow.



7. You can navigate to the source code for any entry in stack frame. In the **Call Stack** window, right-click on the function whose source code you want to see and click on **Go To Source Code** from the shortcut menu.



Using the Parallel Stacks window

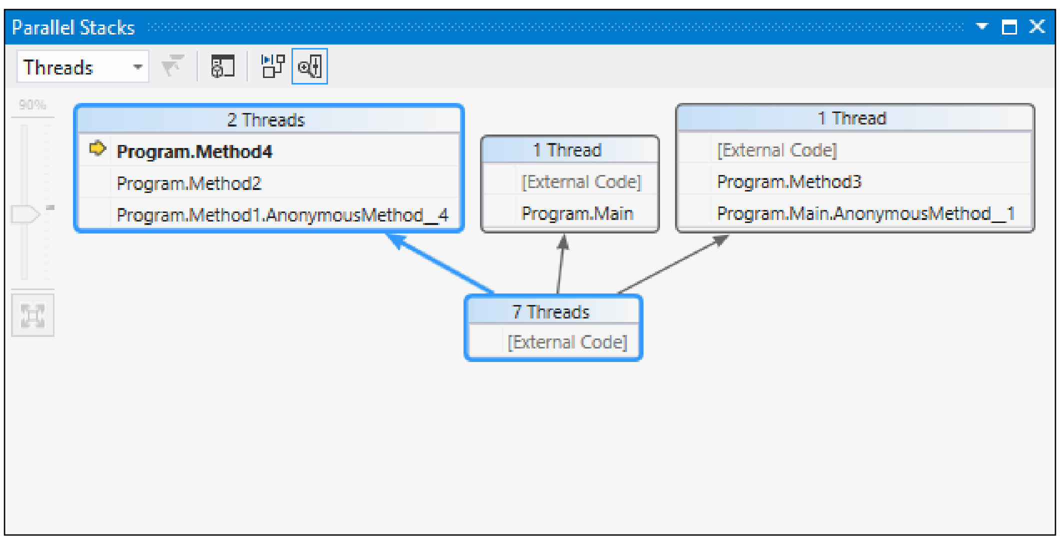
As the degree of parallelism in our applications continues to grow, we need the ability to view and navigate multiple threads call stacks from a single view. A developer debugging a multi-threaded application needs the ability to view call stacks of multiple threads at the same time, in order to see an overall picture of the application's status.

In this recipe, we will see how to use the Parallel Stacks window in Visual Studio 2012 to get a graphical view of the call stacks of all tasks in our application.

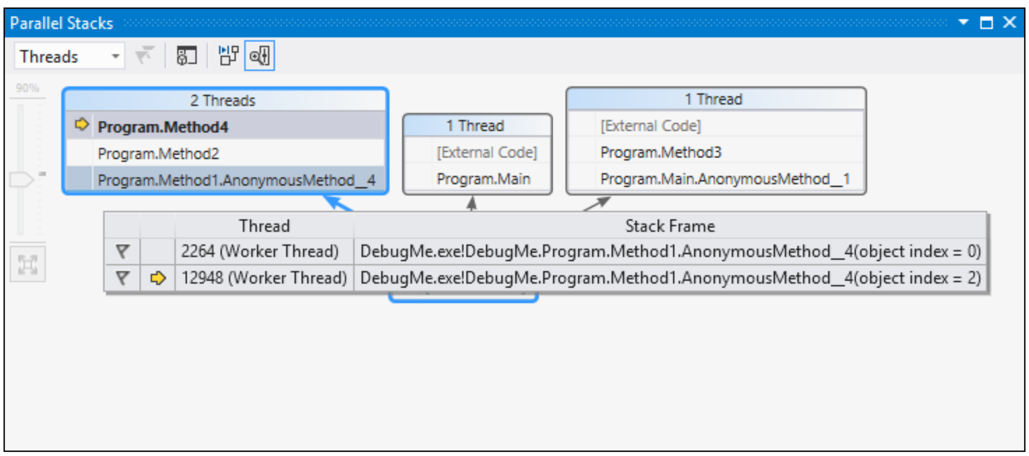
How to do it...

Now, let's go back to Visual Studio 2012 and take a look at the Parallel Stacks window.

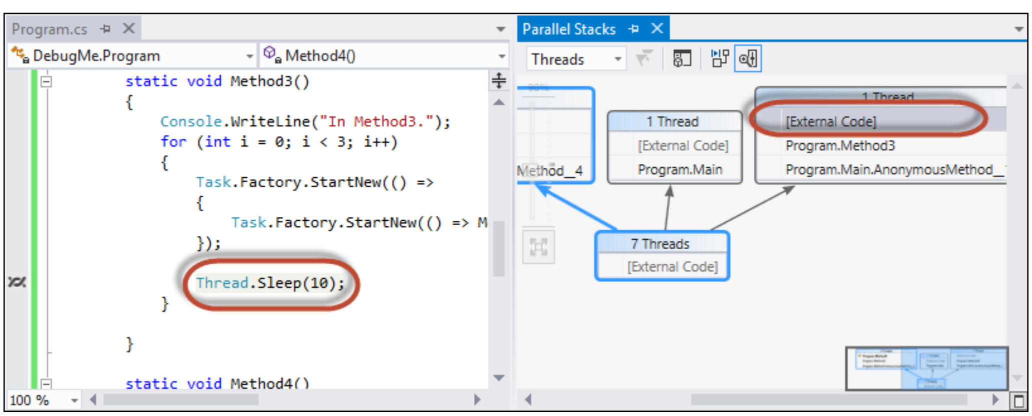
1. In Visual Studio 2012, press *F5* to run the project.
2. When the debugger hits the `Debug.Break` statement, go to the **Debug** menu, click on **Windows**, and click on **Parallel Stacks** to display the **Parallel Stacks** window.



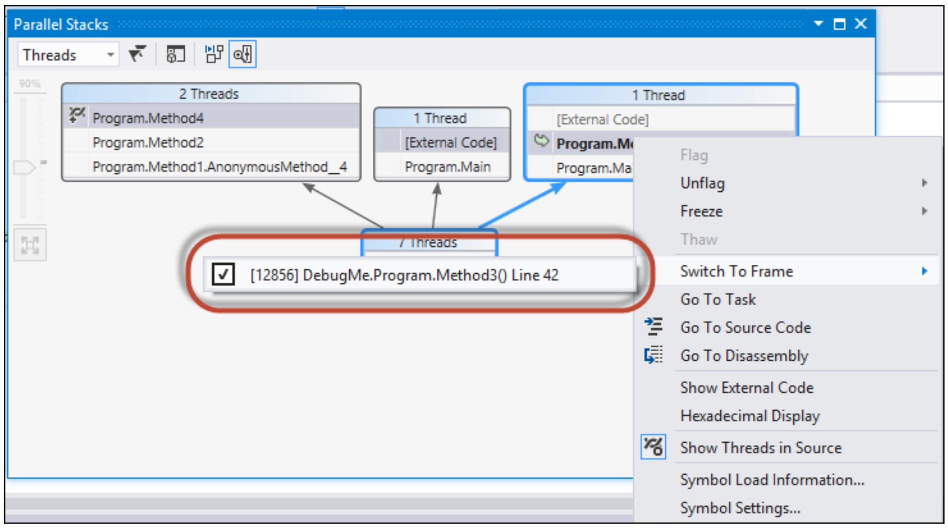
3. Your **Parallel Stacks** view may differ slightly from the image, but you can see the call stacks of all of our tasks in a single graph view. The Parallel Stacks window in the preceding screenshot shows that we have one thread that went from an anonymous method in `Main` to `Method3`, as was called out to **External Code**. One thread is in `Main`, and had gone out to the **External Code**. Two other threads started, have gone through an anonymous method in `Method1`, through `Method2`, to `Method4`. This is also the active stack frame and this is the current thread, as indicated by the flag on the active thread. Visual Studio 2012 groups threads that have the same call stack information together into the same box.
4. Hover your mouse over the boxes and notice the tool tips that show the stack frame information, including method name and parameter values for each thread grouped into the box.



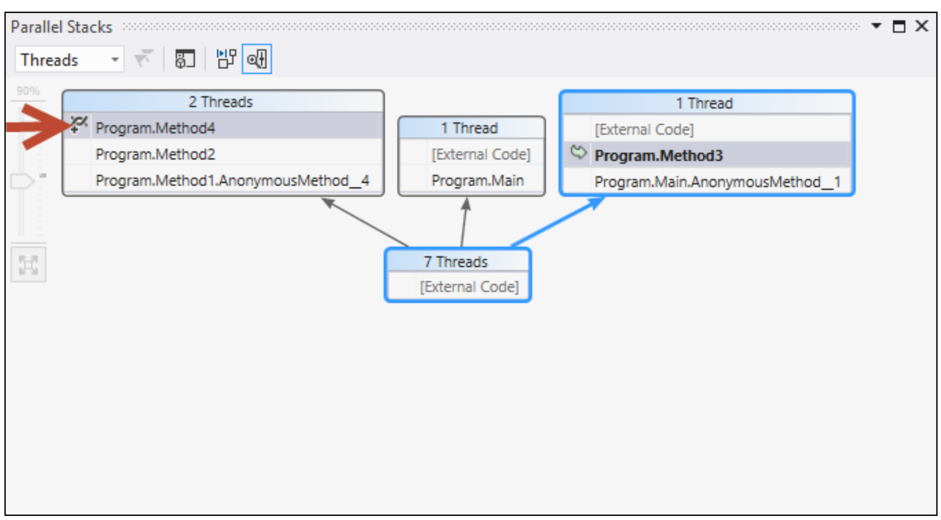
5. You can double-click on any item in the stack frame of the thread to navigate to the code.



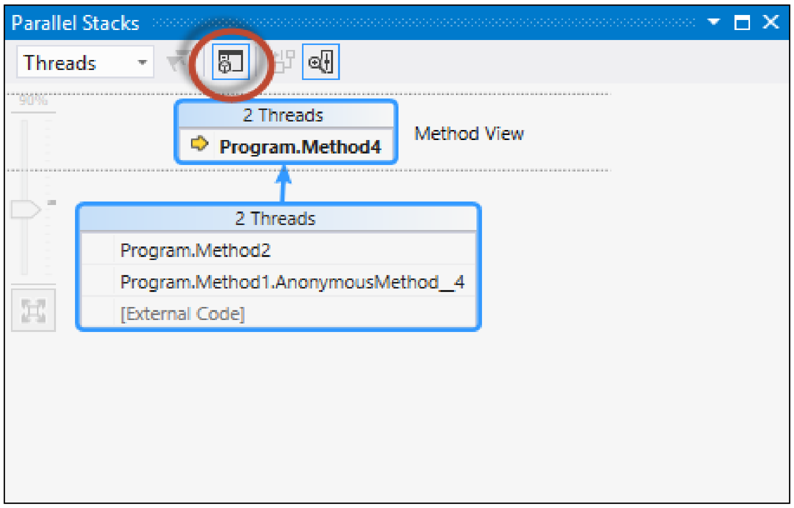
6. To switch to another thread, right-click on the stack frame of the desired thread and click on **Switch To Frame**. Notice that highlight has changed to the selected stack frame, and there is a green arrow in the box indicating that this is the current stack frame that the debugger is focusing on, as opposed to the active stack frame which is indicated by an arrow.



7. You can switch back to the active stack frame by double-clicking on it in the **Parallel Stacks** windows. Notice that it has a thread icon rather than an arrow while the different stack frame has the focus of the debugger. Once you double-click on it, the arrow returns, indicating that this is the active and current thread.



8. You can see the **Threads** that have called a method by clicking on the current stack frame, and then click on the **Method View** button on the **Parallel Stacks** window menu. After clicking on the button, the view will change to show which methods the threads are calling.



Watching values in a thread with Parallel Watch window

Traditionally, debuggers have been designed to work in the context of a single thread at a time. In order to work with a different thread, you needed to first switch the thread context. Visual Studio 2012 has a feature known as Parallel Watch window that allows you to display the values of a variable or expression on multiple threads.

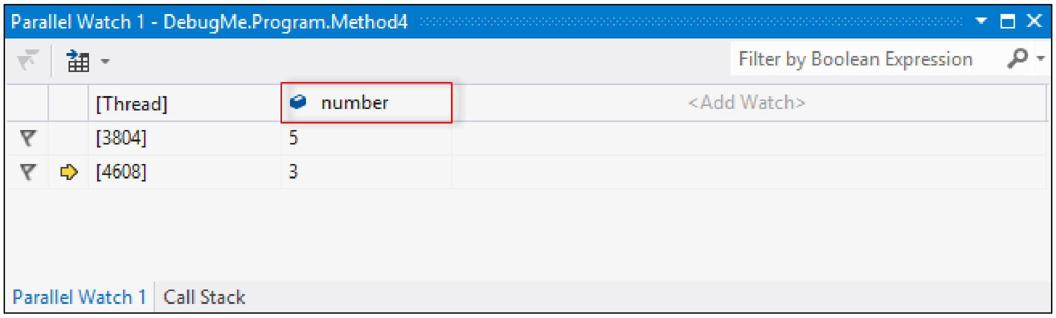
In this recipe, we are going to see how to view the value of a variable across multiple threads using the Parallel Watch window.

How to do it...

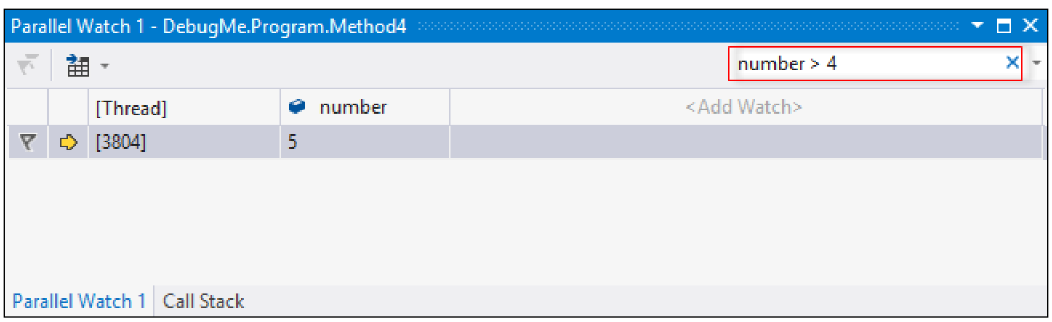
Let's see how the Parallel Watch window can help us view variable values across multiple threads.

1. In Visual Studio 2012, press *F5* to run the project.
2. When the debugger hits the `Debug.Break` statement, go to the **Debug** menu of Visual Studio 2012, and click on **Windows**. Then click on **Parallel Watch** and **Parallel Watch 1** to view the window.

3. By default, the **Parallel Watch** window brings up all the threads currently executing in the process. In order to add new watches, we need to click on **<Add Watch>** column which allows us to enter an expression. Click on **<Add Watch>** and enter numbers as the expression to watch. As soon as the watch is added we can now see the expression evaluated across all the different threads in the **Watch** window.



4. Enter a Boolean expression in the **Filter by Boolean Expression** box. The debugger evaluates the expression for each thread context. Only rows where the value is true are displayed.



Detecting deadlocks with the Parallel Tasks window

A very useful feature of the Visual Studio 2012 debugger is the ability to detect deadlocks in your tasks. A deadlock occurs when two or more tasks permanently block each other by each task having a lock on a resource which the other tasks are trying to lock, or by waiting for each other to finish.

The easiest way to find a deadlock in your application is to use the Parallel Tasks window of Visual Studio 2012. The Parallel Tasks window is very similar to the Threads window, except that it shows information about each `Task` or `task_handle` object instead of each thread, along with the status of the task.

In this recipe, we are going to create a `Console` application that will create several tasks in a loop. The tasks will deadlock because each task will be waiting for the next task to finish.

Getting ready...

Before we use the Parallel Task window to see the deadlocks in our code, we need to create an application that has some deadlocks for us to see.

1. Start a new project using the **C# Console Application** project template and assign `DetectDeadlock` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
```

3. In the `Main` method of the `Program` class, let's start by creating a variable for the number of tasks to create `CountDownEvent`, and an array of `tasks`.

```
static void Main()
{
  int taskCount = 5;
  var countdownEvent = new CountdownEvent(taskCount);
  var tasks = new Task[taskCount];
}
```

4. Now, just below the previous code, let's create the blocked `tasks` in a `for` loop. Each task should wait for the next task to finish.

```
for(int i = 0; i < taskCount; i++)
{
   tasks[i] = Task.Factory.StartNew((state) =>
     {
       countdownEvent.Signal();
       tasks[(((int)state)+1)taskCount].Wait();
     },i);
}
```

5. Next, let's create and start a couple of tasks that won't block.

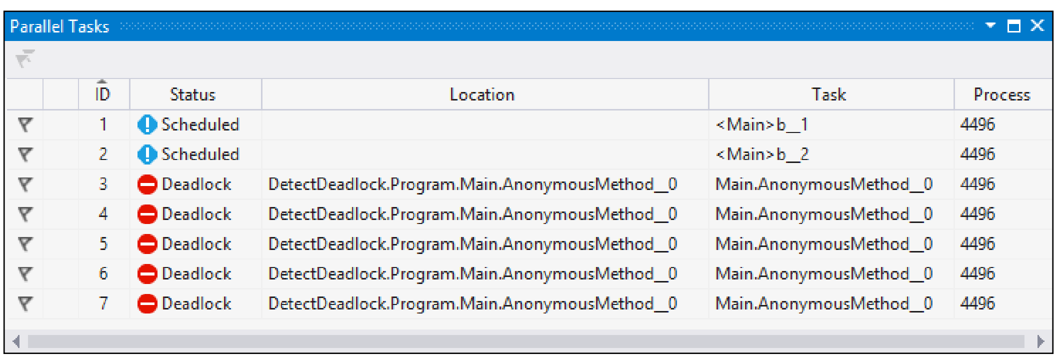
```
var task1 = new Task(() =>
{
  Thread.Sleep(500);
});

var task2 = new Task(() =>
{
  Thread.Sleep(500);
});

task1.Start();
task2.Start();
```

6. Finish up the `Main` method by waiting for `CountDownEvent` and setting a breakpoint for `Debugger`.

```
countdownEvent.Wait();
Debugger.Break();
```

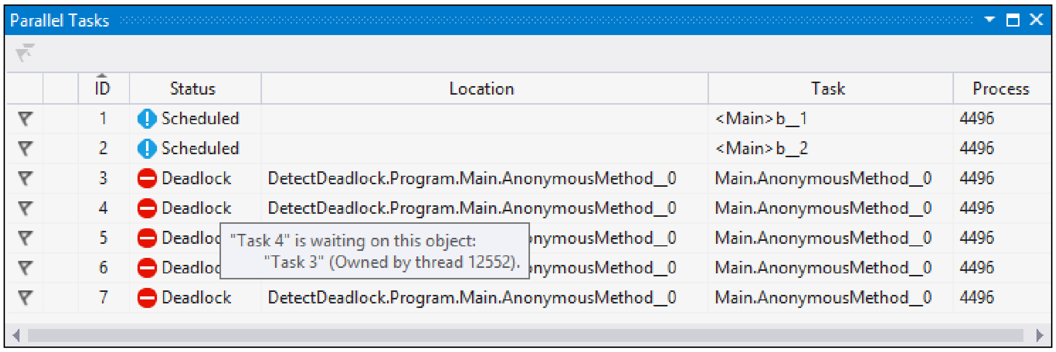
How to do it...

1. In Visual Studio 2012, press *F5* to run the project.
2. When the debugger hits the `Debug.Break` statement, go to the **Debug** menu of Visual Studio 2012 and click on **Windows**. Then click on **Parallel Tasks** to view the window.

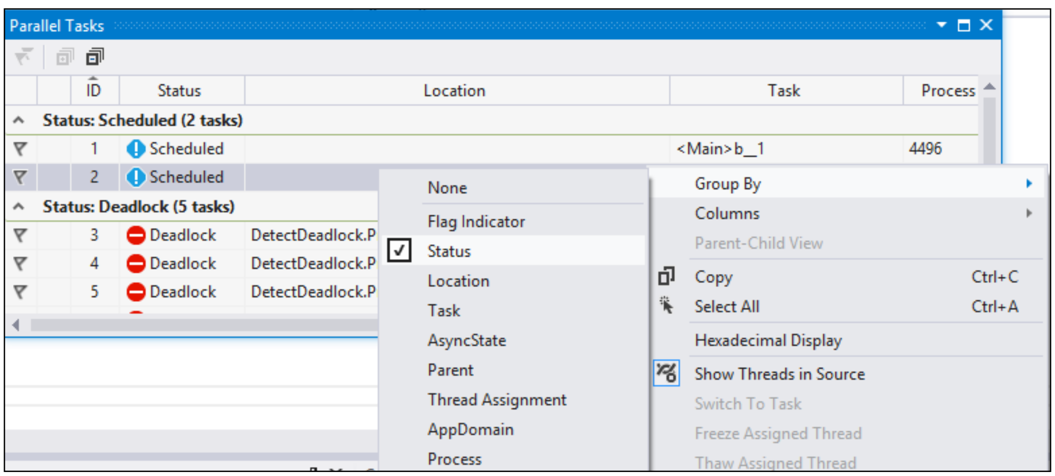

Parallel Tasks

ID	Status	Location	Task	Process
1	Scheduled		<Main>b__1	4496
2	Scheduled		<Main>b__2	4496
3	Deadlock	DetectDeadlock.Program.Main.AnonymousMethod__0	Main.AnonymousMethod__0	4496
4	Deadlock	DetectDeadlock.Program.Main.AnonymousMethod__0	Main.AnonymousMethod__0	4496
5	Deadlock	DetectDeadlock.Program.Main.AnonymousMethod__0	Main.AnonymousMethod__0	4496
6	Deadlock	DetectDeadlock.Program.Main.AnonymousMethod__0	Main.AnonymousMethod__0	4496
7	Deadlock	DetectDeadlock.Program.Main.AnonymousMethod__0	Main.AnonymousMethod__0	4496

3. In the **Parallel Tasks** window you can see all of the tasks that have been identified to be deadlocked. Hover your mouse over any of the blocked tasks to see what the task is waiting for.



4. For applications with a lot of tasks, it can be useful to group the tasks by their status. Right-click anywhere in the **Parallel Tasks** window, click on **Group By** and then click on **Status**.



Measure CPU utilization with Concurrency Visualizer

Parallel applications are not only prone to common sources of inefficiency that are found in sequential applications, but they can also suffer from uniquely parallel performance issues such as load imbalance, excessive synchronization overhead, or thread migration.

Understanding such performance issues can be a difficult and time-consuming process. However, Visual Studio 2012 includes a profiling tool, the Concurrency Visualizer, which can significantly reduce the burden of parallel performance analysis.

In this recipe we will be looking at the CPU Utilization view of the Concurrency Visualizer.

Getting ready...

Before we look at the Concurrency Visualizer, we need to create a `Console` application that is going to exercise the processor on your development machine a bit. This `Console` application will be a slight variation of the `MultipleProducerConsumer` application we created in *Chapter 5*, *Concurrent Collections*. The application will use a `for` loop to create some producer `tasks` that the producers use to perform a mathematic operation on some numbers, and add the results to `BlockingCollection`. `BlockingCollection`, which is a class that provides blocking and bounding capabilities for thread safe collections that implement `IProducerConsumerCollection<T>`. There will also be four consumer `tasks` that retrieve the results from `BlockingCollection` and write them to `Console`.

1. Start a new project using the **C# Console Application** project template and assign `MultipleProducerConsumer` as the **Solution name**.
2. Add the following `using` directives to the top of your `Program` class:

   ```
   using System;
   using System.Collections.Concurrent;
   using System.Collections.Generic;
   using System.Threading.Tasks;
   ```

3. Let's add a `static` method to the `Program` class which the producer tasks will call to perform the calculation.

   ```
   private static double CalcSumRoot(int root)
   {
     double result = 0;
     for (int i = 1; i < 10000000; i++)
     {
       result += Math.Exp(Math.Log(i) / root);
   ```

```
  }
  return result;
}
```

4. Now, just below the previous method, let's create another `static` method that consumer `tasks` will use to write the results to `Console`.

```
private static void DisplayResults(BlockingCollection<double>
results)
{
  foreach (var item in results.GetConsumingEnumerable())
  {
    Console.Write("\nConsuming item: {0}", item);
  }
}
```

5. In the `Main` method, let's start by creating `BlockingCollection`, to be the buffer between the producers and consumers, list of tasks, and the definition for four consumer `tasks`.

```
var results = new BlockingCollection<double>();
var tasks = new List<Task>();
var consume1 = Task.Factory.StartNew(() =>
DisplayResults(results));
var consume2 = Task.Factory.StartNew(() =>
DisplayResults(results));
var consume3 = Task.Factory.StartNew(() =>
DisplayResults(results));
var consume4 = Task.Factory.StartNew(() =>
DisplayResults(results));
```

6. Now let's create a `for` loop that spins up some producer `tasks`, performs the calculations, and adds the results to `BlockingCollection`.

```
for (int item = 1; item < 100; item++)
{
  var value = item;
  var compute = Task.Factory.StartNew(() =>
  {
    var calcResult = CalcSumRoot(value);
    Console.Write("\nProducing item: {0}", calcResult);
    results.TryAdd(calcResult);
  });
  tasks.Add(compute);
}
```

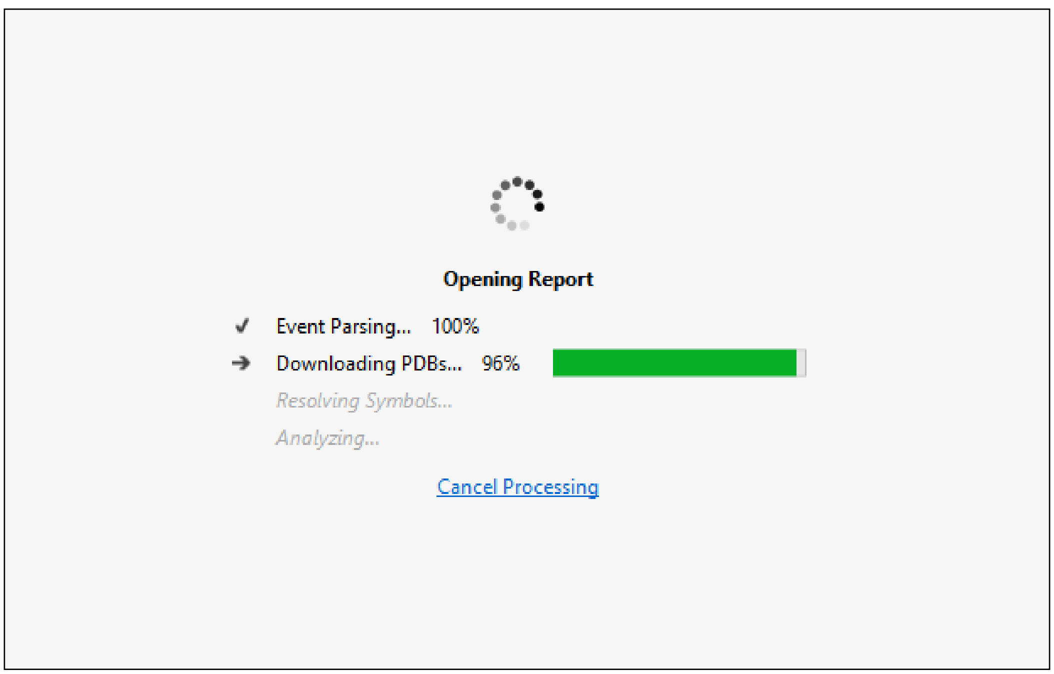
7. Finally, let's create a continuation that calls `CompleteAdding` on `BlockingCollection` when all producer `tasks` finish. Wait for the user input before exiting.

```
Task.Factory.ContinueWhenAll(tasks.ToArray(), result =>
{
 results.CompleteAdding();
 Console.Write("\nCompleted adding.");
});

Console.ReadLine();
```

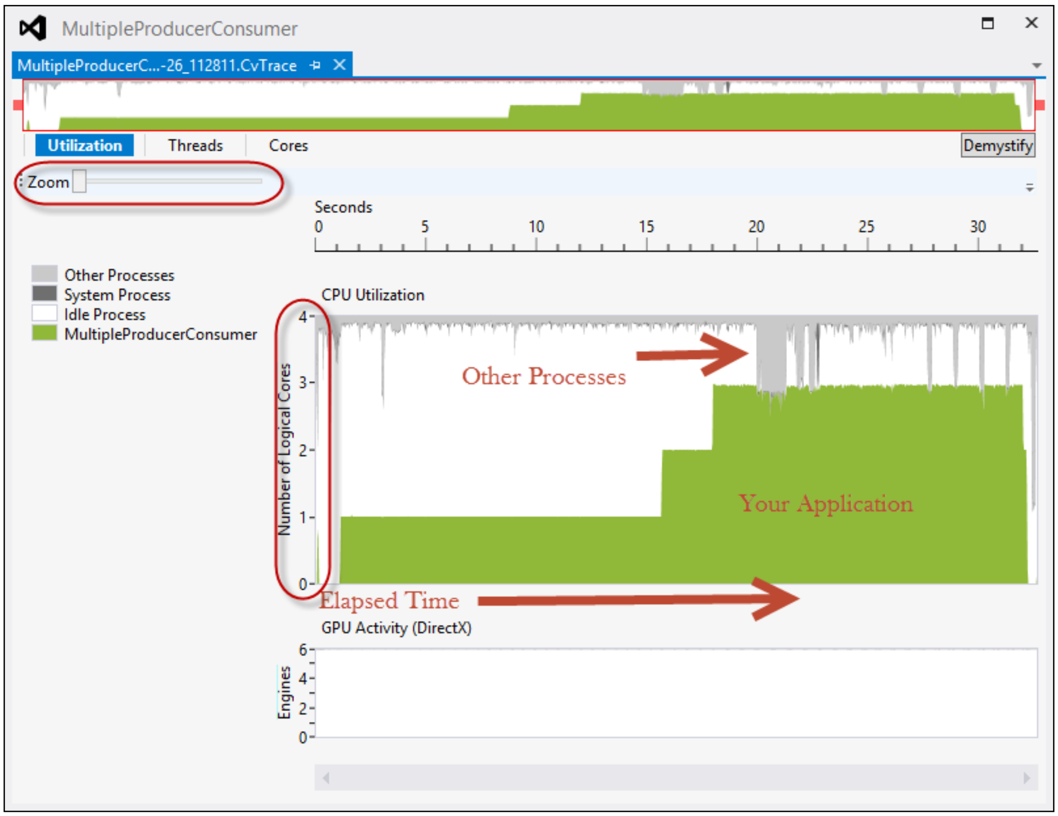
How to do it...

Now, let's see how we can use Concurrency Visualizer to report on the performance and efficiency of parallel code.

1. On Visual Studio 2012 menu, click on **Analyze**, and then click on **Concurrency Visualizer**, and **Start with Current Project**. You will see the application running while Visual Studio 2012 collects data and builds a report in the background.
2. When the application finishes running, close the application. Visual Studio 2012 will then finalize and open the performance report.



3. Once the report is completed and loaded, you will see the **Utilization** view of the Concurrency Visualizer. The X axis shows the **Elapsed Time** since the trace started. The Y axis shows the number of logical processor cores in your system. The green area shows the **Number of Logical Cores** that the application is using at any given point in the analysis run. The rest of the cores are either idle, or are being used by **Other Processes** which are shown by the gray lines coming from the top of the graph. There is a **Zoom** slider at the top which can be used to narrow the time scale of the graph.



4. When tuning your parallel application, this view allows you to confirm the degree of parallelism. You can get hints of common parallel performance problems by reviewing the graph. Load imbalances among the processor's cores appear as stair-step patterns in the graph. Contention for synchronization objects appear on the graph, such as serial execution when parallel is expected.

Using Concurrency Visualizer Threads view

Threads view is probably the most useful and frequently used view in the Concurrency Visualizer. By using this view, you can identify whether the threads are executing or blocking because of synchronization or some other reason. Threads view assigns a category to each context switch when a thread has stopped executing.

In this recipe, we are going to use the Concurrency Visualizer to show all of the context switch events for each application thread.

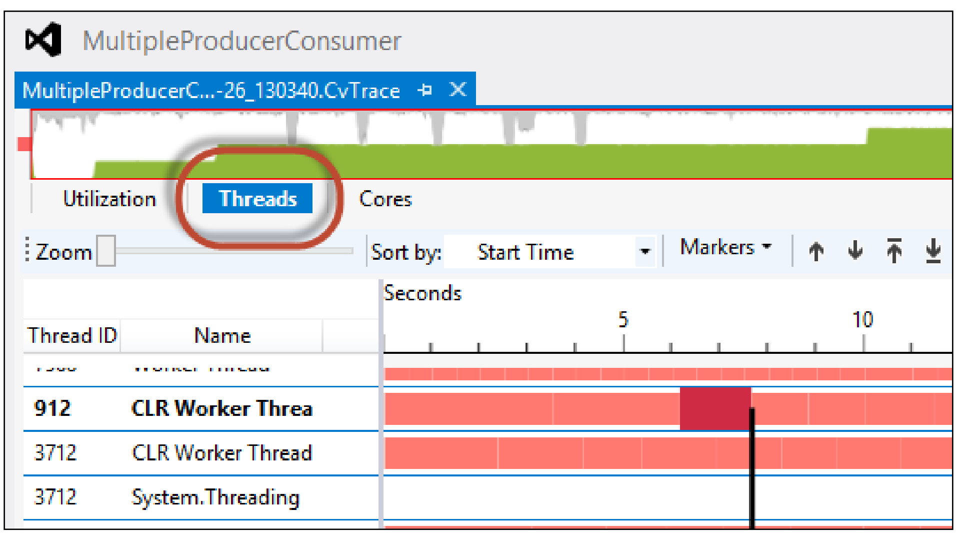
Getting ready...

For this recipe, we will use the same sample application as in the previous recipe. If you have closed the `MultipleProducerConsumer` solution, please reopen it, and go to the Concurrency Visualizer through the Visual Studio 2012 menu, and click on **Analyze**. Then, click on **Concurrency Visualizer**, and **Open Trace.** Alternately, you can rerun the Concurrency Visualizers as in the previous recipe.

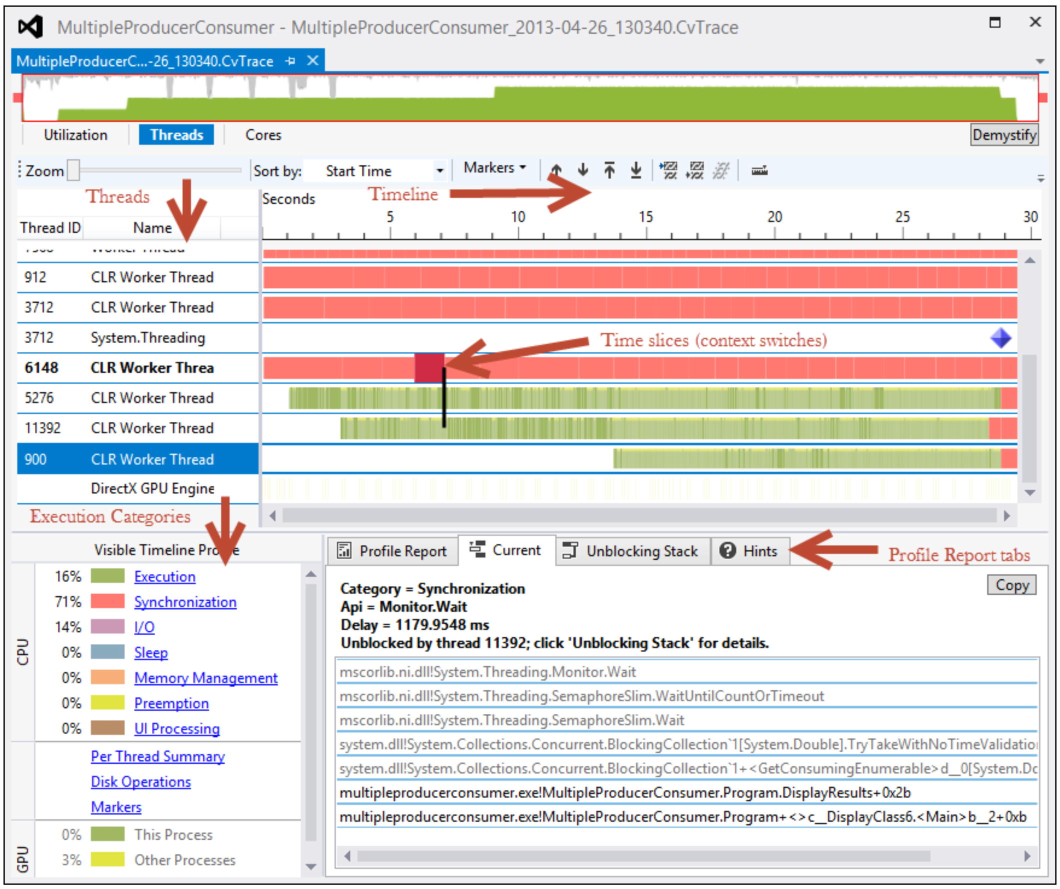
How to do it...

Let's take a look at what we can do with the Concurrency Visualizer Threads view.

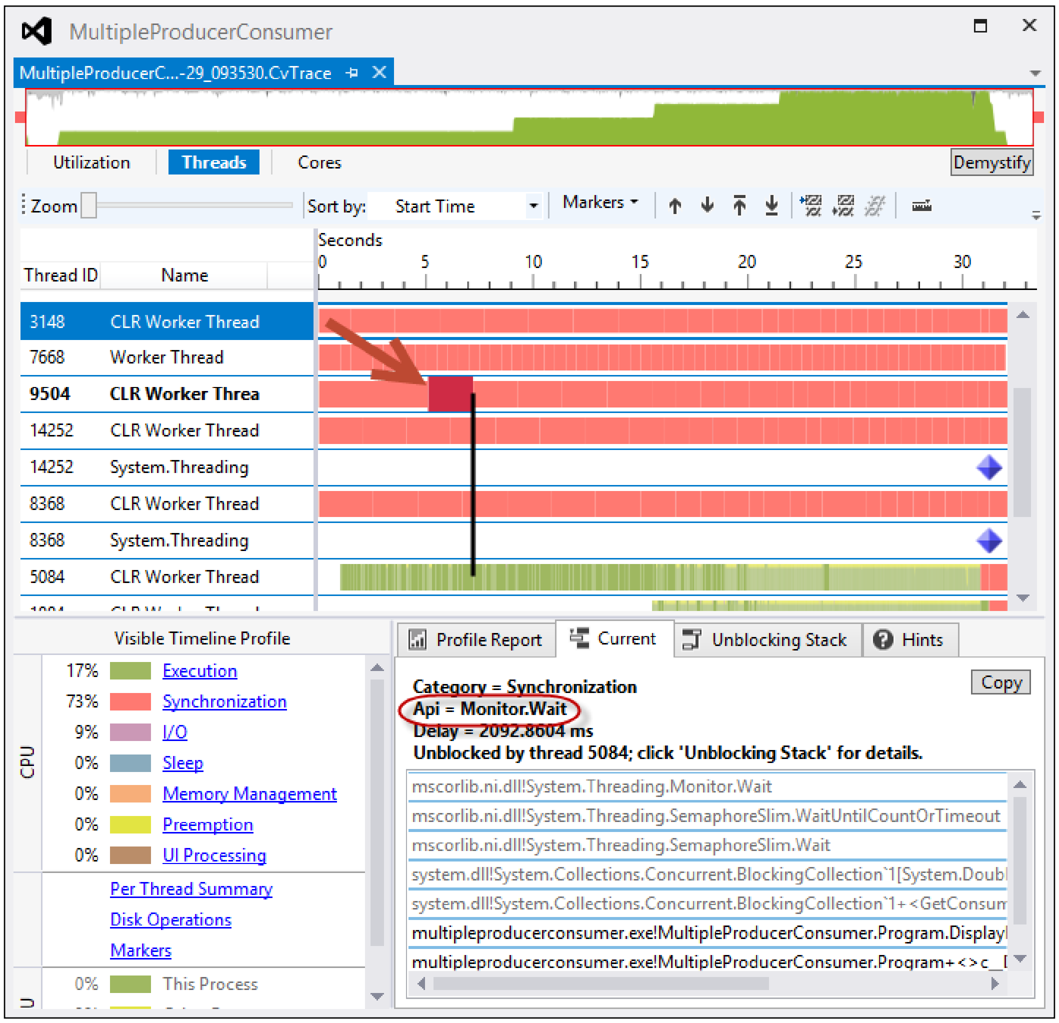
1. Open the **Concurrency Visualizer** for the **MultipleProducerConsumer** solution and click on the **Threads** view.



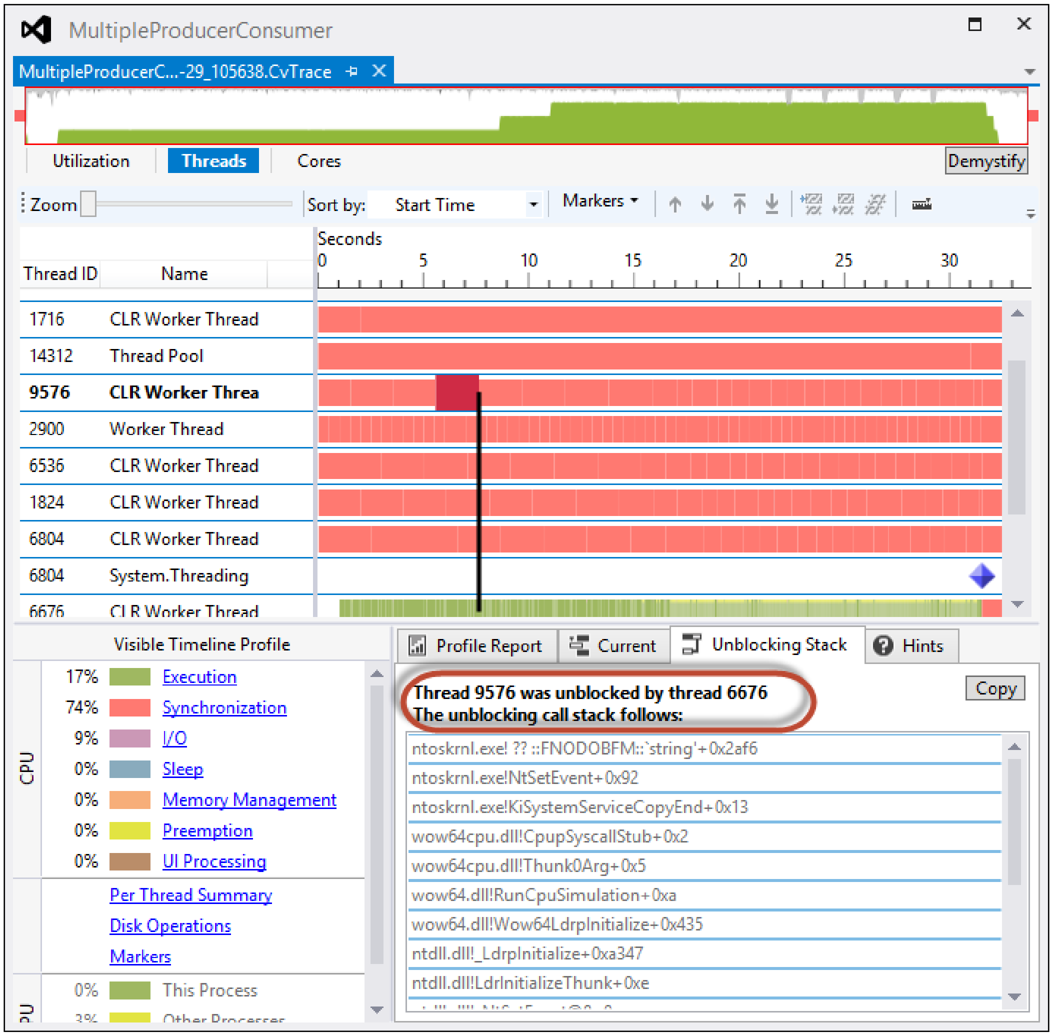
2. When the **Threads** view opens, you will notice that the X axis is again a **Timeline**. However, the Y axis is divided into horizontal channels. If the physical disks on your machine have any activity during the running of the application (which in this case they don't), the top channels will depict your physical disks. In our case, the channels are all threads in our application. You will see the `Main` thread, a debugger helper thread, and all of your worker threads. Below the list of **Threads**, you will see the **Execution Categories** that are assigned by Concurrency Visualizer. In the following screenshot, you can see that the application spent **16%** of the **Timeline** in **Execution** and **71%** of the **Timeline** in **Synchronization**.



3. Click on one of the CLR worker threads in the top channel. The **Timeline** next to the channel for the **CLR Worker Thread** will be divided into the **Time slices** for the thread. Click on the **Current** tab in the **Profile Report** tab. You will see the call stack for the thread at the time of the context switch, the reason for the context switch, and **Category** assigned by the report. You can see on the **Current** tab that this thread was context switched because it arrived at a **Monitor.Wait** call.



4. One of the most valuable features of the **Threads** view is the ability to determine thread dependencies. Select a synchronization segment for a worker thread (a pink segment). On the **Current** tab you will see the thread that unblocked the current thread. Click on the **Unblocking Stack** tab and you will see the call stack of the thread that unblocked the current thread.



Using Concurrency Visualizer Cores view

Frequent context switching can seriously degrade application performance, especially when threads migrate across cores when they resume execution. The reason for this performance impact is that running threads load instructions and data they need into the cache hierarchy, and when a thread resumes execution on a different core, there can be latency while working data is reloaded from memory or other caches.

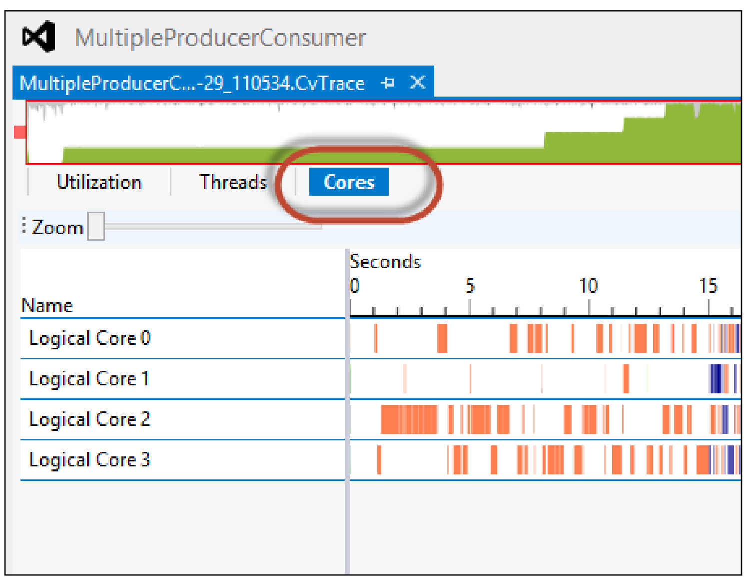
The Cores view of the Concurrency Visualizer is a tool that aids in identifying excessive context switches. In this recipe, we will return to the `MultipleProducerConsumer` solution to see how we can examine the context switching that occurs in the application.

Getting ready...

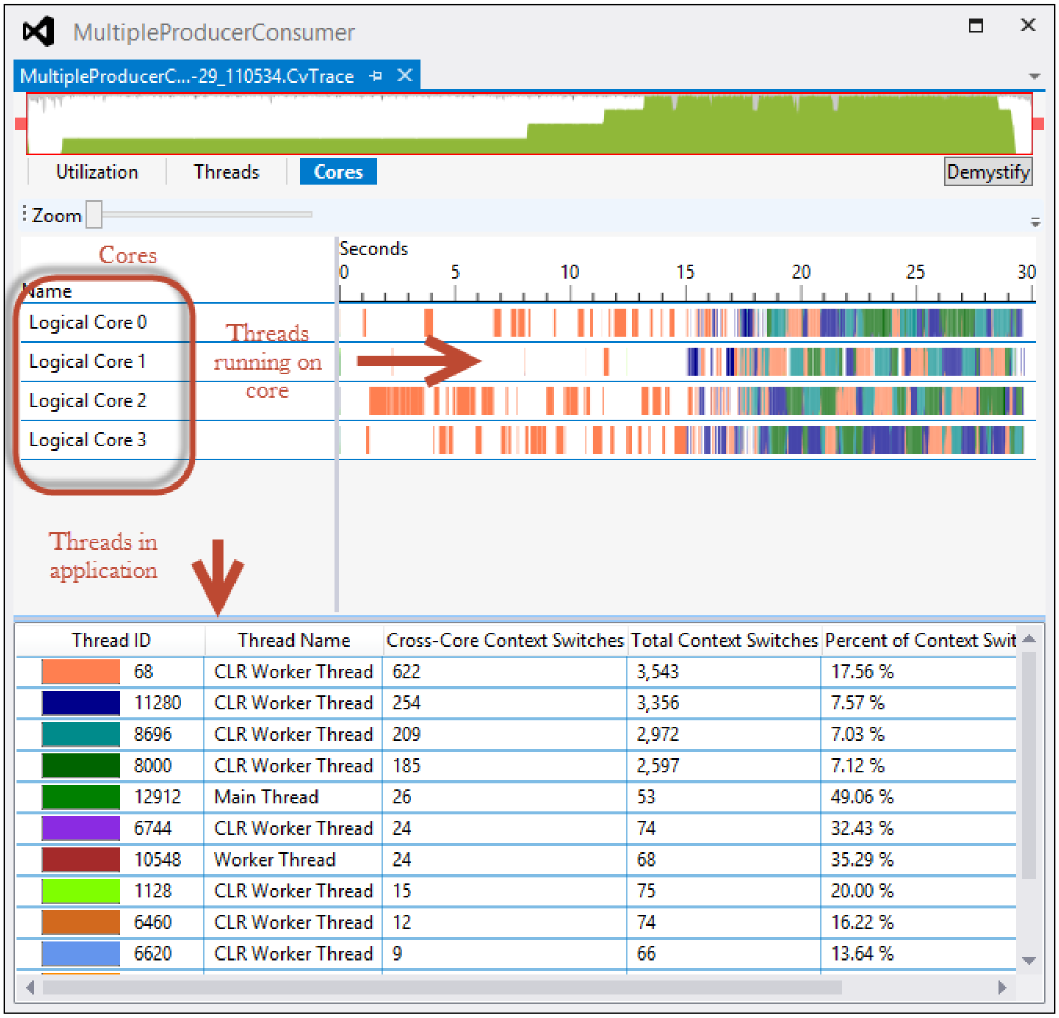
For this recipe, we will use the same sample application as in the previous recipe. If you have closed the `MultipleProducerConsumer` solution, please reopen it and go to the Concurrency Visualizer through the Visual Studio 2012 menu. Click on **Analyze**, and then click on **Concurrency Visualizer**, and **Open Trace**. Alternately, you can rerun the Concurrency Visualizers as in the previous recipe.

How to do it...

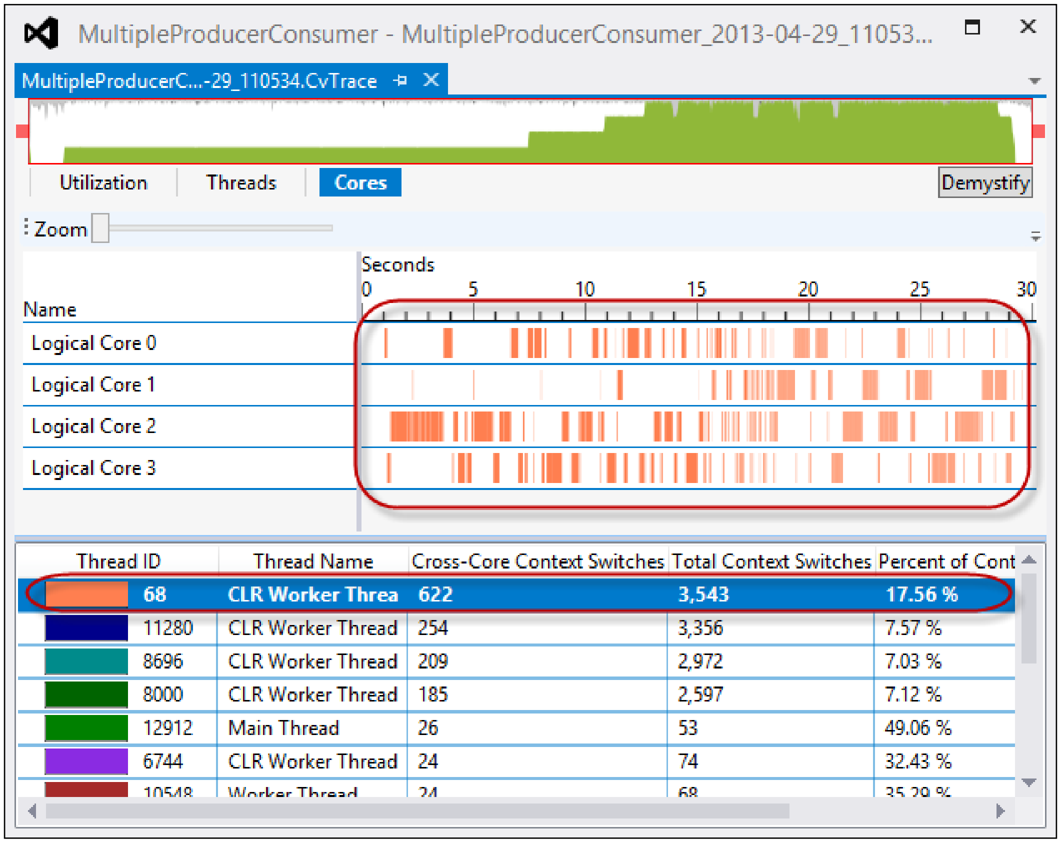
1. Open the Concurrency Visualizer for the `MultipleProducerConsumer` solution and click on the **Cores** view.



2. Like the other views, the **Cores** view displays the timeline on the X axis. The logical cores of the system are shown on the Y axis. Each thread in the application is shown in a different color, and thread execution segments are displayed on the core channels.



3. The statistics shown in the **Cores** view help the developer to identify **Threads** that have excessive context switches and incur core migrations. The list of threads at the bottom of the **Cores** view is sorted by the number of **Cross-Core Context Switches**. Click on the thread with the highest number of core switches (the top thread in the list). Notice how the thread execution is spread across the available **Cores** in your system.



8
Async

In this chapter, we will cover the following recipes:

- Creating an `async` method
- Handling `Exceptions` in asynchronous code
- Cancelling an asynchronous operation
- Cancelling `async` operation after timeout period
- Processing multiple `async` tasks as they complete
- Improving performance of `async` solution with `Task.WhenAll`
- Using `async` for file access
- Checking the progress of an asynchronous task

Introduction

We've all seen client applications that do not respond to mouse events or update the display for noticeable periods of time. This delay is likely the result of code holding on to the single UI thread for far too long. Maybe it is waiting for network I/O or maybe it is performing an intensive computation. Meanwhile, the user is left sitting there waiting, as our application grinds to a halt. The answer to this problem is asynchrony.

How is the concept of asynchrony different from parallelism? Parallelism, with which you are quite familiar by this point in the book, is mainly about application performance. Parallelism enables developers to perform CPU intensive work on multiple threads at once, taking advantage of modern multi-core computer architectures. Asynchrony on the other hand, is a superset of concurrency. It includes concurrency as well as other asynchronous calls which are more I/O bound than CPU bound. Let's say you are saving a large file to your hard drive or you want to download some data from the server. These kinds of I/O bound tasks are ideal for asynchrony. Asynchrony is a pattern which yields control instantly, and waits for a callback or other notification to occur before continuing.

So, just do things using an asynchronous pattern and your UI responsiveness problems are solved, right? Well, yes, but there is one small problem. Asynchronous code is difficult, at least historically speaking. However, asynchrony is taking a huge leap forward in terms of usability. Microsoft has delivered this by building on the `Task` functionality in .NET 4.5, as well as the addition of two new keywords to the .NET Framework: `async` and `await`.

In this chapter, we will walk through several recipes that show how to maintain a responsive UI or scalable services by using the new **Task-based Asynchronous Pattern** (**TAP**) of the .NET Framework 4.5.

Creating an async method

The TAP is a new pattern for asynchronous programming in .NET Framework 4.5. It is based on a task, but in this case a task doesn't represent work which will be performed on another thread. In this case, a task is used to represent arbitrary asynchronous operations.

Let's start learning how `async` and `await` work by creating a **Windows Presentation Foundation** (**WPF**) application that accesses the web using `HttpClient`. This kind of network access is ideal for seeing TAP in action. The application will get the contents of a classic book from the web, and will provide a count of the number of words in the book.

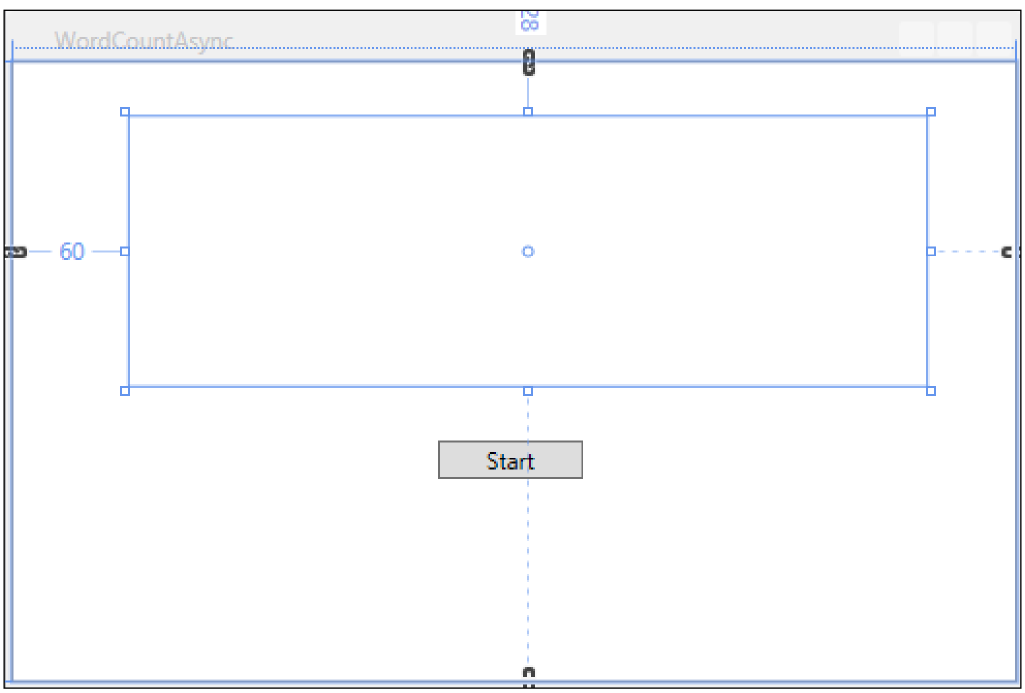
How to do it...

Let's go to Visual Studio 2012 and see how to use the `async` and `await` keywords to maintain a responsive UI by doing the web communications asynchronously.

1. Start a new project using the **WPF Application** project template and assign `WordCountAsync` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create a simple user interface containing `Button` and `TextBlock`:

```
<Window x:Class="WordCountAsync.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
```

```
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WordCountAsync" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton"
    Content="Start"
    HorizontalAlignment="Left"
    Margin="219,195,0,0"
    VerticalAlignment="Top"
    Width="75"
    RenderTransformOrigin="-0.2,0.45"
    Click="StartButton_Click"/>
        <TextBlock x:Name="TextResults"
      HorizontalAlignment="Left"
      Margin="60,28,0,0"
      TextWrapping="Wrap"
      VerticalAlignment="Top"
      Height="139"
      Width="411"/>

    </Grid>
</Window>
```



3. Next, open up `MainWindow.xaml.cs`. Go to the **Project** and add a reference to `System.Net.Http`.

4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
```

6. Add a button click event for the `StartButton` and add the `async` modifier to the method signature to indicate that this will be a `async` method. Please note that `async` methods that return `void` are normally only used for event handlers, and should be avoided.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{

}
```

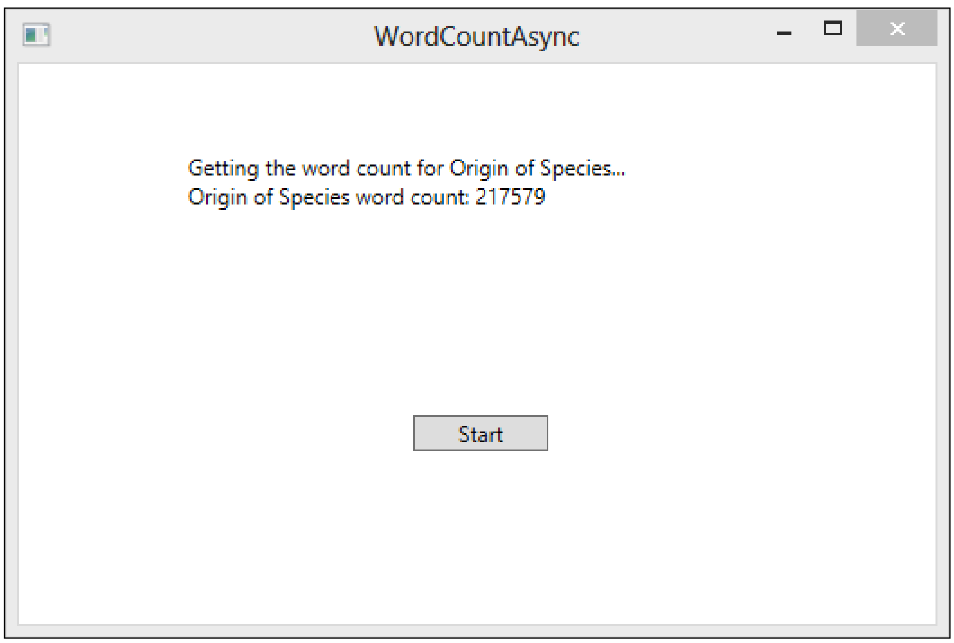
7. Next, let's create a `async` method called `GetWordCountAsync` that returns `Task<int>`. This method will create `HttpClient` and call its `GetStringAsync` method to download the book contents as a string. It will then use the `Split` method to split the string into a `wordArray`. We can return the count of the `wordArray` as our `return` value.

```
public async Task<int> GetWordCountAsync()
{
  TextResults.Text += "Getting the word count for Origin of
Species...\n";
  var client = new HttpClient();
  var bookContents = await client.GetStringAsync(@"http://www.
gutenberg.org/files/2009/2009.txt");
  var wordArray = bookContents.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);
  return wordArray.Count();
}
```

8. Finally, let's complete the implementation of our button click event. The `Click` event handler will just call `GetWordCountAsync` with the `await` keyword and display the results to `TextBlock`.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{
  var result = await GetWordCountAsync();
  TextResults.Text += String.Format("Origin of Species word count:
{0}",result);
}
```

9. In Visual Studio 2012, press *F5* to run the project. Click on the **Start** button, and your application should appear as shown in the following screenshot:



How it works...

In the TAP, asynchronous methods are marked with an `async` modifier. The `async` modifier on a method does not mean that the method will be scheduled to run asynchronously on a worker thread. It means that the method contains control flow that involves waiting for the result of an asynchronous operation, and will be rewritten by the compiler to ensure that the asynchronous operation can resume this method at the right spot.

Let me try to put this a little more simply. When you add the async modifier to a method, it indicates that the method will wait on an asynchronous code to complete. This is done with the await keyword. The compiler actually takes the code that follows the await keyword in an async method and turns it into a continuation that will run after the result of the async operation is available. In the meantime, the method is suspended, and control returns to the method's caller.

If you add the async modifier to a method, and then don't await anything, it won't cause an error. The method will simply run synchronously.

An async method can have one of the three return types: void, Task, or Task<TResult>. As mentioned before, a task in this context doesn't mean that this is something that will execute on a separate thread. In this case, task is just a container for the asynchronous work, and in the case of Task<TResult>, it is a promise that a result value of type TResult will show up after the asynchronous operation completes.

In our application, we use the async keyword to mark the button click event handler as asynchronous, and then we wait for the GetWordCountAsync method to complete by using the wait keyword.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
  StartButton.Enabled = false;
  var result = await GetWordCountAsync();
  TextResults.Text += String.Format("Origin of Species word count:
{0}",
..................  result);
  StartButton.Enabled = true;
}
```

The code that follows the await keyword, in this case, the same line of code that updates TextBlock, is turned by the compiler into a continuation that will run after the integer result is available.

If the Click event is fired again while this asynchronous task is in progress, another asynchronous task is created and awaited. To prevent this, it is a common practice to disable the button that is clicked.

It is a convention to name an asynchronous method with an Async postfix, as we have done with GetWordCountAsync.

Handling Exceptions in asynchronous code

So how would you add `Exception` handling to code that is executed asynchronously? In previous asynchronous patterns, this was very difficult to achieve. In C# 5.0 it is much more straightforward because you just have to wrap the asynchronous function call with a standard `try/catch` block.

On the surface this sounds easy, and it is, but there is more going on behind the scene that will be explained right after we build our next example application.

For this recipe, we will return to our classic books word count scenario, and we will be handling an `Exception` thrown by `HttpClient` when it tries to get the book contents using an incorrect URL.

How to do it...

Let's build another WPF application and take a look at how to handle `Exceptions` when something goes wrong in one of our asynchronous methods.

1. Start a new project using the **WPF Application** project template and assign `AsyncExceptions` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create a simple user interface containing `Button` and a `TextBlock`:

```
<Window x:Class="WordCountAsync.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WordCountAsync" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton"
    Content="Start"
    HorizontalAlignment="Left"
    Margin="219,195,0,0"
    VerticalAlignment="Top"
    Width="75"
    RenderTransformOrigin="-0.2,0.45"
    Click="StartButton_Click"/>
        <TextBlock x:Name="TextResults"
      HorizontalAlignment="Left"
```

```
        Margin="60,28,0,0"
        TextWrapping="Wrap"
        VerticalAlignment="Top"
        Height="139"
        Width="411"/>

    </Grid>
</Window>
```

3. Next, open up `MainWindow.xaml.cs`. Go to the **Project Explorer**, right-click on **References**, click on **Framework** from the menu on the left side of the **Reference Manager**, and then add a reference to `System.Net.Http`.
4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
```

6. Now let's create our `GetWordCountAsync` method. This method will be very similar to the last recipe, but it will be trying to access the book on an incorrect URL. The asynchronous code will be wrapped in a `try/catch` block to handle `Exception`. We will also use a `finally` block to dispose of `HttpClient`.

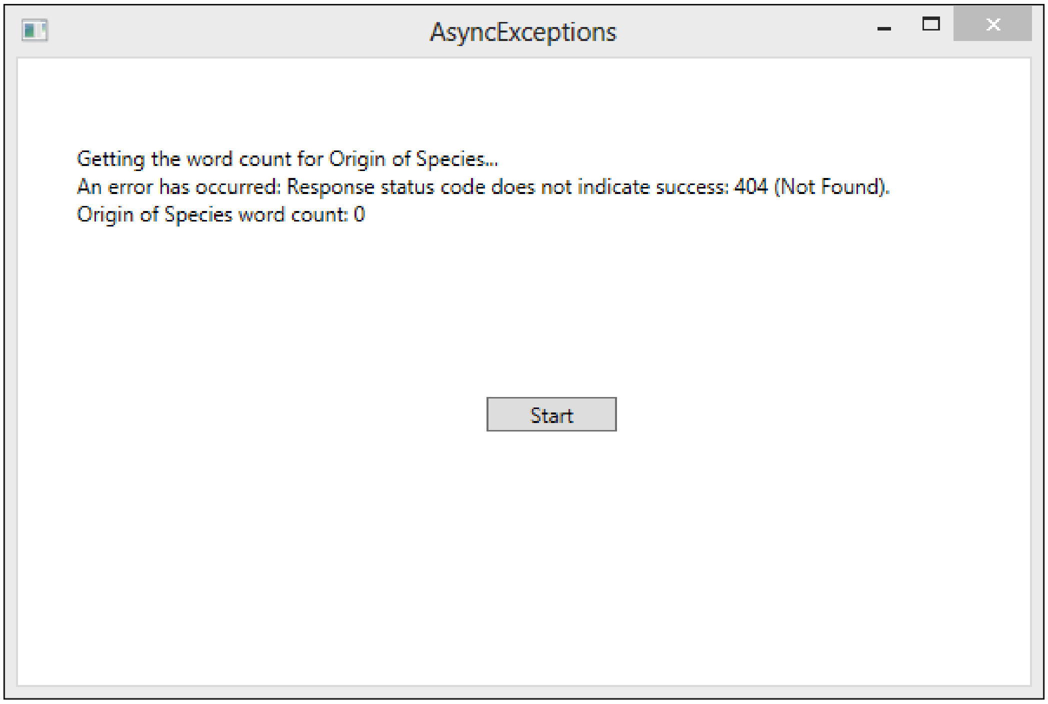
```
public async Task<int> GetWordCountAsync()
{
  ResultsTextBlock.Text += "Getting the word count for Origin of
Species...\n";
  var client = new HttpClient();
  try
  {
    var bookContents = await client.GetStringAsync(@"http://www.
gutenberg.org/files/2009/No_Book_Here.txt");
    var wordArray = bookContents.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);
    return wordArray.Count();
```

```
    }
    catch (Exception ex)
    {
      ResultsTextBlock.Text += String.Format("An error has occurred:
{0} \n", ex.Message);
      return 0;
    }
    finally
    {
      client.Dispose();
    }

  }
```

7. Finally, let create the `Click` event handler for our `StartButton`. This is pretty much the same as the last recipe, just wrapped in a `try/catch` block. Don't forget to add the `async` modifier to the method signature.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{
  try
  {
    var result = await GetWordCountAsync();
    ResultsTextBlock.Text += String.Format("Origin of Species word
count: {0}",
        result);
  }
  catch(Exception ex)
  {
    ResultsTextBlock.Text += String.Format("An error has occurred:
{0} \n",
            ex.Message);
  }
}
```

8. Now, in Visual Studio 2012, press *F5* to run the project. Click on the **Start** button. Your application should appear as shown in the following screenshot:



How it works...

Wrapping your asynchronous code in a `try/catch` block is pretty easy. In fact, it hides some of the complex work Visual Studio 2012 to doing for us.

To understand this, you need to think about the context in which your code is running.

When the TAP is used in Windows Forms or WPF applications, there's already a context that the code is running in, such as the message loop UI thread. When `async` calls are made in those applications, the awaited code goes off to do its work asynchronously and the `async` method exits back to its caller. In other words, the program execution returns to the message loop UI thread.

The `Console` applications don't have the concept of a context. When the code hits an awaited call inside the `try` block, it will exit back to its caller, which in this case is `Main`. If there is no more code after the awaited call, the application ends without the `async` method ever finishing.

To alleviate this issue, Microsoft included `async` compatible context with the TAP that is used for `Console` apps or unit test apps to prevent this inconsistent behavior. This new context is called `GeneralThreadAffineContext`.

Do you really need to understand these context issues to handle async `Exceptions`? No, not really. That's part of the beauty of the Task-based Asynchronous Pattern.

Cancelling an asynchronous operation

In .NET 4.5, asynchronous operations can be cancelled in the same way that parallel tasks can be cancelled, by passing in `CancellationToken` and calling the `Cancel` method on `CancellationTokenSource`.

In this recipe, we are going to create a WPF application that gets the contents of a classic book over the web and performs a word count. This time though we are going to set up a **Cancel** button that we can use to cancel the `async` operation if we don't want to wait for it to finish.

How to do it...

Let's create a WPF application to show how we can add cancellation to our asynchronous methods.

1. Start a new project using the **WPF Application** project template and assign `AsyncCancellation` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create our user interface. In this case, the UI contains `TextBlock`, `StartButton`, and `CancelButton`.

```
<Window x:Class="AsyncCancellation.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="AsyncCancellation" Height="400" Width="599">
    <Grid Width="600" Height="400">
        <Button x:Name="StartButton"
        Content="Start"
        HorizontalAlignment="Left"
        Margin="142,183,0,0"
        VerticalAlignment="Top"
        Width="75"
        RenderTransformOrigin="-0.2,0.45"
```

```
            Click="StartButton_Click"/>
            <Button x:Name="CancelButton"
            Content="Cancel"
            HorizontalAlignment="Left"
            Margin="379,185,0,0"
            VerticalAlignment="Top"
            Width="75"
            Click="CancelButton_Click"/>
            <TextBlock x:Name="TextResult"
            HorizontalAlignment="Left"
            Margin="27,24,0,0"
            TextWrapping="Wrap"
            VerticalAlignment="Top"
            Height="135"
            Width="540"/>
        </Grid>
</Window>
```

3. Next, open up `MainWindow.xaml.cs`, click on the **Project Explorer**, and add a reference to `System.Net.Http`.
4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
```

6. Next, let's create the `GetWordCountAsync` method. This method is very similar to the method explained before. It needs to be marked as asynchronous with the `async` modifier and it returns `Task<int>`. This time however, the method takes a `CancellationToken` parameter. We also need to use the `GetAsync` method of `HttpClient` instead of the `GetStringAsync` method, because the former supports cancellation, whereas the latter does not. We will add a small delay in the method so we have time to cancel the operation before the download completes.

```
public async Task<int> GetWordCountAsync(CancellationToken ct)
{
  TextResult.Text += "Getting the word count for Origin of
Species...\n";
```

```
  var client = new HttpClient();
  await Task.Delay(500);
  try
  {
    HttpResponseMessage response = await client.GetAsync(@"http://
www.gutenberg.org/files/2009/2009.txt", ct);
    var words = await response.Content.ReadAsStringAsync();
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
  }
  finally
  {
    client.Dispose();
  }
}
```

7. Now, let's create the `Click` event handler for our `CancelButton`. This method just needs to check if `CancellationTokenSource` is `null`, and if not, it calls the `Cancel` method.

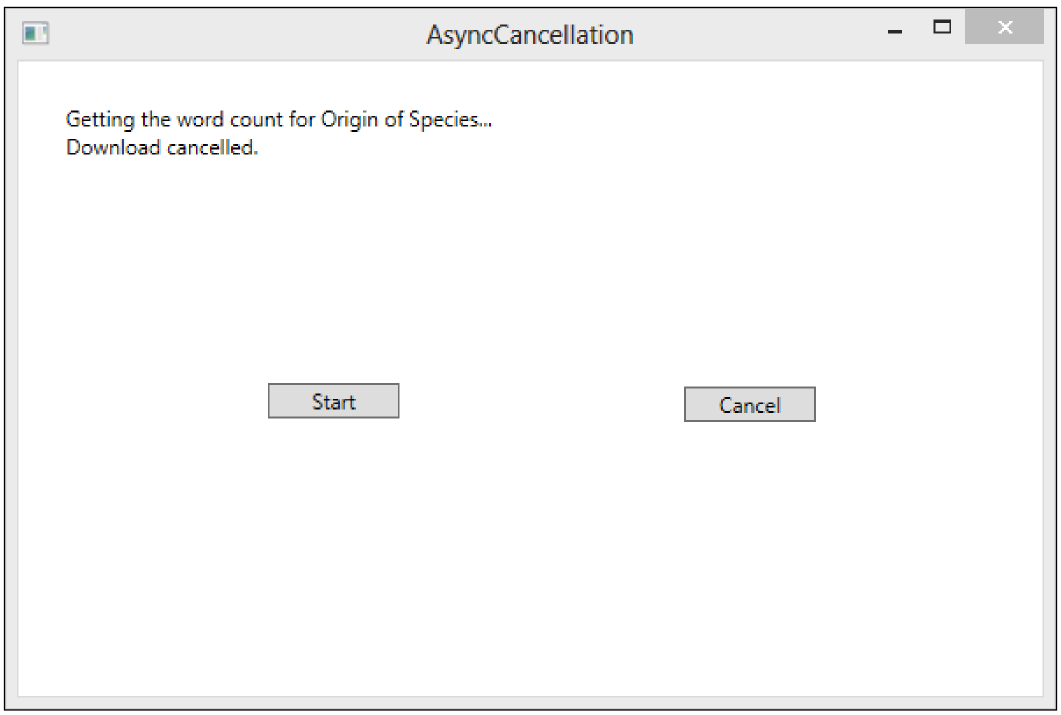
```
private void CancelButton_Click(object sender, RoutedEventArgs e)
{
  if (cts != null)
  {
    cts.Cancel();
  }
}
```

8. Ok, let's finish up by adding a `Click` event handler for `StartButton`. This method is the same as explained before, except we also have a `catch` block that specifically handles `OperationCancelledException`. Don't forget to mark the method with the `async` modifier.

```
public async Task<int> GetWordCountAsync(CancellationToken ct)
{
  TextResult.Text += "Getting the word count for Origin of
Species...\n";
  var client = new HttpClient();
  await Task.Delay(500);
  try
  {
    HttpResponseMessage response = await client.GetAsync(@"http://
www.gutenberg.org/files/2009/2009.txt", ct);
```

```
        var words = await response.Content.ReadAsStringAsync();
        var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
        return wordArray.Count();
      }
      finally
      {
        client.Dispose();
      }
    }
```

9. In Visual Studio 2012, press *F5* to run the project Click on the **Start** button, then the **Cancel** button. Your application should appear as shown in the following screenshot:



How it works...

Cancellation is an aspect of user interaction that you need to consider to build a professional `async` application. In this example, we implemented cancellation by using a **Cancel** button, which is one of the most common ways to surface cancellation functionality in a GUI application.

In this recipe, cancellation follows a very common flow.

1. The caller (start button click event handler) creates a `CancellationTokenSource` object.

   ```
   private async void StartButton_Click(object sender,
   RoutedEventArgs e)
   {
     cts = new CancellationTokenSource();
     ...
   }
   ```

2. The caller calls a cancelable method, and passes `CancellationToken` from `CancellationTokenSource` (`CancellationTokenSource.Token`).

   ```
   public async Task<int> GetWordCountAsync(CancellationToken ct)
   {
     ...
     HttpResponseMessage response = await client.GetAsync(@"http://
   www.gutenberg.org/files/2009/2009.txt", ct);
     ...
   }
   ```

3. The cancel button click event handler requests cancellation using the `CancellationTokenSource` object (`CancellationTokenSource.Cancel()`).

   ```
   private void CancelButton_Click(object sender, RoutedEventArgs e)
   {
     if (cts != null)
     {
       cts.Cancel();
     }
   }
   ```

4. The task acknowledges the cancellation by throwing `OperationCancelledException`, which we handle in a `catch` block in the start button click event handler.

Cancelling async operation after timeout period

Another common scenario for cancelling asynchronous tasks is to set a timeout period by using the `CancellationTokenSource.CancelAfter` method. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the `CancelAfter` expression.

In this recipe, we are going to create a WPF application that gets the contents of a classic book over the web and performs a word count. This time though, we are going to set a timeout period after which the task gets cancelled.

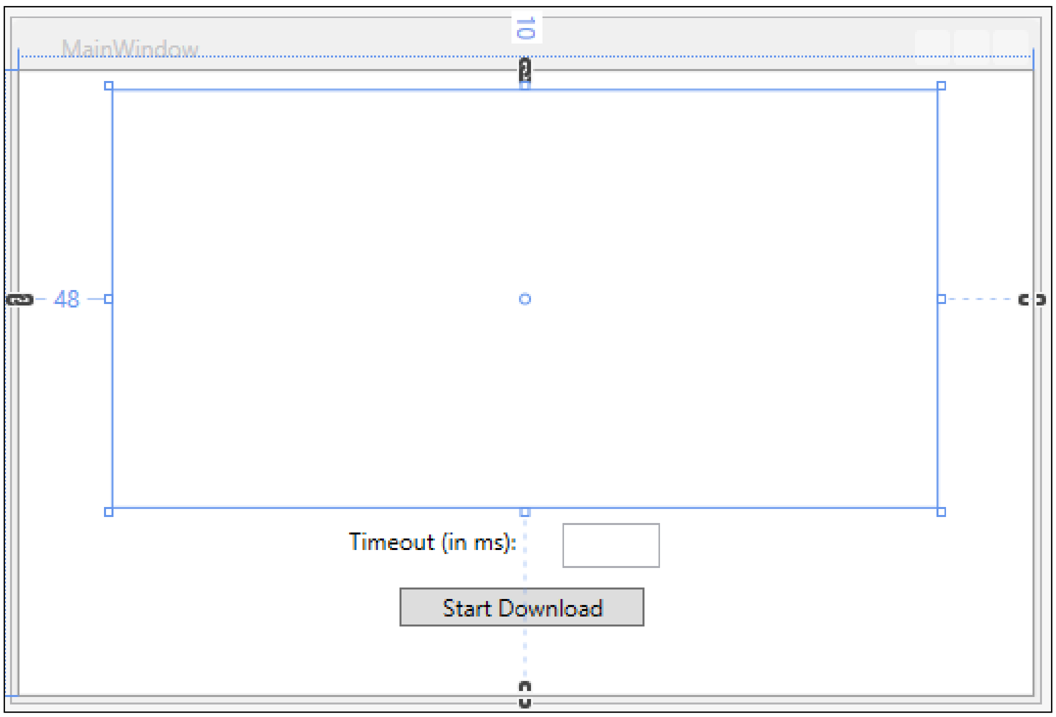
How to do it...

Now, let's see how we can create an asynchronous task that cancels after a specified timeout period.

1. Start a new project using the **WPF Application** project template and assign `CancelAfterTimeout` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and add the following XAML to create our user interface:

```
<Window x:Class="CancelAfterTimeout.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton"
        Content="Start Download"
        HorizontalAlignment="Left"
        Margin="194,264,0,0"
        VerticalAlignment="Top"
        Width="125"
        RenderTransformOrigin="-0.2,0.45"
        Click="StartButton_Click"/>
        <TextBlock x:Name="TextResult"
        HorizontalAlignment="Left"
        Margin="48,10,0,0" TextWrapping="Wrap"
        VerticalAlignment="Top"
        Height="213"
        Width="420"/>
```

```
            <Label Content="Timeout (in ms):"
             HorizontalAlignment="Left"
             Margin="163,227,0,0"
             VerticalAlignment="Top"/>
            <TextBox x:Name="TextTimeout"
            HorizontalAlignment="Left"
            Height="23"
            Margin="277,231,0,0"
            TextWrapping="Wrap"
            VerticalAlignment="Top"
            Width="50"/>
        </Grid>
    </Window>
```



3. Next, open up `MainWindow.xaml.cs`. Go to the **Project Explorer** and add a reference to `System.Net.Http`.

4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Windows;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
```

6. Next, let's create the `GetWordCountAsync` method. This method is exactly the same as the last recipe. It needs to be marked as asynchronous with the `async` modifier and it returns `Task<int>`. The method takes a `CancellationToken` parameter. We will add a small delay in the method so we have time to cancel the operation before the download completes.

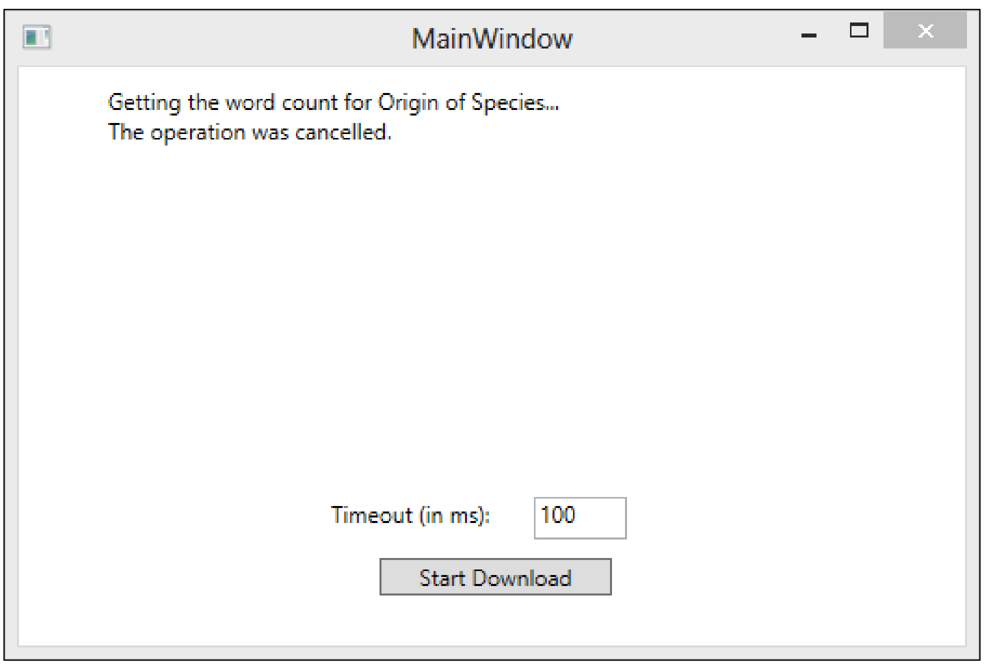
```
public async Task<int> GetWordCountAsync(CancellationToken ct)
{
  TextResult.Text += "Getting the word count for Origin of
Species...\n";
  var client = new HttpClient();
  await Task.Delay(500);
  try
  {
    HttpResponseMessage response = await client.GetAsync(@"http://
www.gutenberg.org/files/2009/2009.txt", ct);
    var words = await response.Content.ReadAsStringAsync();
    var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
    return wordArray.Count();
  }
  finally
  {
    client.Dispose();
  }
}
```

7. Ok, let's finish up by adding a `Click` event handler for `StartButton`. This method is similar to the last recipe, except we call the `CancellationTokenSource.CancelAfter` method, passing it the value of our `timeout` textbox. Don't forget to mark the method with the `async` modifier.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{
  StartButton.IsEnabled = false;
```

```
    try
    {
      tokenSource = new CancellationTokenSource();
      var timeoutPeriod = int.Parse(TextTimeout.Text);
      tokenSource.CancelAfter(timeoutPeriod);
      await GetWordCount(tokenSource.Token);
    }
    catch (OperationCanceledException)
    {
      TextResult.Text += "The operation was cancelled. \n";
    }
    catch (Exception)
    {
      TextResult.Text += "The operation failed to complete due to an 
exception. \n";
    }
    finally
    {
      StartButton.IsEnabled = true;
    }
  }
```

8. In Visual Studio 2012, press *F5* to run the project. Set the `timeout` value to `100`. Your application should appear as shown in the following screenshot:



How it works...

The application is very similar to the application we created in the last recipe, except this time the `Cancel` button isn't used. The actual cancellation follows a similar flow however.

1. The caller (start button click event handler) creates a `CancellationTokenSource` object, and then calls the `CancelAfter` method to pass in the `timeout` value.

   ```
   private async void StartButton_Click(object sender,
   RoutedEventArgs e)
   {
     StartButton.IsEnabled = false;
     try
     {
       tokenSource = new CancellationTokenSource();
       var timeoutPeriod = int.Parse(TextTimeout.Text);
       tokenSource.CancelAfter(timeoutPeriod);
       ...
     }
     ...
   }
   ```

2. The caller calls a cancelable method, and passes `CancellationToken` from `CancellationTokenSource` (`CancellationTokenSource.Token`).

   ```
   public async Task<int> GetWordCountAsync(CancellationToken ct)
   {
     ...
     HttpResponseMessage response = await client.GetAsync(@"http://
   www.gutenberg.org/files/2009/2009.txt", ct);
     ...
   }
   ```

3. After the timeout period expires, `CancellationTokenSource` triggers a cancellation same as if we had made a call to `CancellationTokenSource.Cancel`.

   ```
   private void CancelButton_Click(object sender, RoutedEventArgs e)
   {
     if (cts != null)
     {
       cts.Cancel();
     }
   }
   ```

The task acknowledges the cancellation by throwing `OperationCancelledException`, which we handle in a `catch` block in the start button click event handler.

Processing multiple async tasks as they complete

Many of the methods of the `Task` class that we learned about in *Chapter 1*, *Getting Started with Task Parallel Library*, are as useful when writing an asynchronous code as they are when writing a parallel code. In this recipe, we are going to download the contents of multiple books and use `Task.WhenAny` to process them as they finish.

This application will use a `while` loop to create a collection of tasks. Each task downloads the contents of a specified book. In each iteration of a loop, an awaited call to `WhenAny` returns the task in the collection of tasks that finishes first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

How to do it...

Now, let's create a WPF application that creates multiple asynchronous tasks and processes them as they complete.

1. Start a new project using the **WPF Application** project template and assign `AsyncMultipleRequest` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create our user interface:

```
<Window x:Class="AsyncMultipleRequest.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton"
        Content="Start Download"
        HorizontalAlignment="Left"
        Margin="194,264,0,0"
        VerticalAlignment="Top"
        Width="125"
        RenderTransformOrigin="-0.2,0.45"
        Click="StartButton_Click"/>
        <TextBlock x:Name="TextResult"
           HorizontalAlignment="Left"
           Margin="48,10,0,0"
           TextWrapping="Wrap"
```

```
            VerticalAlignment="Top"
            Height="213" Width="420"/>
    </Grid>
</Window>
```

3. Next, open up `MainWindow.xaml.cs`. Go to the **Project Explorer**, and add a reference to `System.Net.Http`.
4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Documents;
using System.Net.Http;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
```

6. Let's start by creating a helper function that builds a list of `KeyValuePair<string,string>` which represents our book titles and URLs.

```
private List<KeyValuePair<string, string>> GetBookUrls()
{
  var urlList = new List<KeyValuePair<string, string>>
  {
    new KeyValuePair<string,string>("Origin of Species",
          "http://www.gutenberg.org/files/2009/2009.txt"),
    new KeyValuePair<string,string>("Beowulf",
          "http://www.gutenberg.org/files/16328/16328-8.txt"),
    new KeyValuePair<string,string>("Ulysses",
          "http://www.gutenberg.org/files/4300/4300.txt")
  };
  return urlList;
}
```

7. Now let's create an `async` method that does the book download and returns `KeyValuePair<string, int>` that represents our book titles and word count. This method will need to accept a `KeyValuePair<string, string>` parameter representing the book title and URL. The method also needs an `HttpClient` parameter.

```
async Task<KeyValuePair<string,int>> ProcessBook(KeyValuePair<stri
ng,string> book, HttpClient client)
{
  var bookContents = await client.GetStringAsync(book.Value);
  var wordArray = bookContents.Split(delimiters,
StringSplitOptions.RemoveEmptyEntries);
  return new KeyValuePair<string,int>(book.Key,wordArray.Count());
}
```

8. Now we need to create another `async` method called `GetMultipleWordCount`. This method executes a query on our list of books. Each query calls the `ProcessBook` method to actually do the download and obtain the word count. After the query, we set up a `while` loop that loops while our list of book processing tasks is greater than zero. Each iteration of the loop awaits a call to `Task.WhenAny`. When a task is completed, the results are written out and `Task` is removed from the `Task` list. This method takes no parameters and returns `Task`.

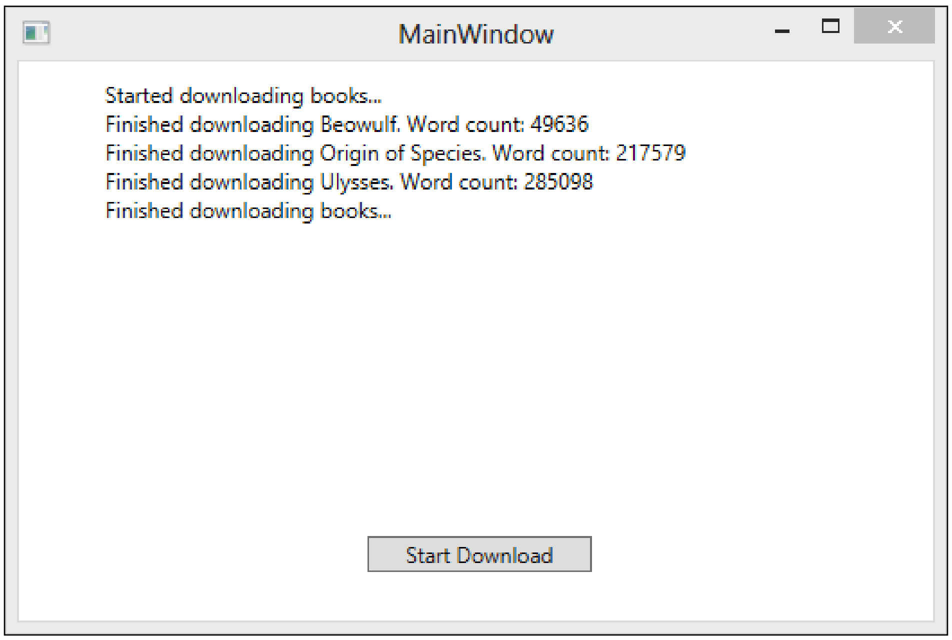
```
public async Task GetMultipleWordCount()
{
  var client = new HttpClient();
  var results = new List<KeyValuePair<string, int>>();
  var urlList = GetBookUrls();
  IEnumerable<Task<KeyValuePair<string,int>>> bookQuery =
    from book in urlList select ProcessBook(book, client);
  List<Task<KeyValuePair<string,int>>> bookTasks = bookQuery.
ToList();
  while (bookTasks.Count > 0)
  {
    Task<KeyValuePair<string, int>> firstFinished = await Task.
WhenAny(bookTasks);
    bookTasks.Remove(firstFinished);
    var thisBook = await firstFinished;
    TextResult.Text += String.Format("Finished downloading {0}.
Word count: {1}\n",
      thisBook.Key,
      thisBook.Value);
  }
}
```

9. Finally, let's create our start button click event handler. The handler only needs to call the `GetMultipleWordCount` method.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{
  TextResult.Text += "Started downloading books...\n";
```

```
        await GetMultipleWordCount();
        TextResult.Text += "Finished downloading books...\n";
    }
```

10. In Visual Studio 2012, press *F5* to run the project. Your application should appear as shown in the following screenshot:



How it works...

We have already seen in the previous recipes that the `WhenAny` method of the `Task` class can be used on a list of parallel tasks to continue processing when any of the tasks in the array is complete.

Even though a task in the `async` context doesn't mean that our list of `async` tasks are running in parallel on separate worker threads, we can still use the `WhenAny` method to handle `async` requests as they complete.

In this recipe, we downloaded the text of multiple books and displayed the word count of each of the books as the download finished. The start button's click event handler doesn't do much other than add some text to `TextBlock` and `await` a call to the `GetMultipleWordCount` method. After creating `HttpClient`, the `GetMultipleWordCount` method makes a call to the `GetBookUrls` helper method that we created, which just returns a list of three books and their URLs.

After getting the list of books and their URLs, the `GetMultipleWordCount` method creates `IEnumerable<Task<TResult>>` by executing a LINQ query that calls the `ProcessBook` method on each `book` in the list.

```
var bookQuery = from book in urlList select ProcessBook(book, client);
var bookTasks = bookQuery.ToList();
```

Next, we set up a `while` loop on the condition that `bookTasks.Count` is greater than zero. In the body of the `while` loop, we `await` a call to the `Task.WhenAny` method, which will return when the first list of tasks is complete. We then remove this `Task` from `bookTasks` so the count is decremented. Below that, we await the `firstFinished` task variable. This has the effect of the compiler creating a continuation for us at this point that will run, as soon as the `task` variable `firstFinished` is completed, the compiler-created continuation will contain the code to update the `TextBlock` with the word count for the book.

```
var firstFinished = await Task.WhenAny(bookTasks);
bookTasks.Remove(firstFinished);
var thisBook = await firstFinished;

// The compiler will create a continuation at this point that will run
//  when the task referenced by the firstFinished variable completes.
TextResult.Text += String.Format("Finished downloading {0}. Word
count: {1}\n",
  thisBook.Key,
  thisBook.Value);
```

Improving performance of async solution with Task.WhenAll

We have already seen how we can use the `Task.WhenAny` method to handle asynchronous tasks as they complete. You will also find the `Task.WhenAll` method very useful in the asynchronous context. In some applications that create multiple asynchronous requests, it can improve application performance by using `Task.WhenAll` to hold off on processing results until all the asynchronous tasks have completed.

In this recipe, we are going to create a WPF application that downloads the contents of multiple books asynchronously, but holds off on processing the results until all the tasks have completed.

How to do it...

1. Start a new project using the **WPF Application** project template and assign `AsyncMultipleRequest` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create our user interface:

```
<Window x:Class="AsyncMultipleRequest.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button x:Name="StartButton"
        Content="Start Download"
        HorizontalAlignment="Left"
        Margin="194,264,0,0"
        VerticalAlignment="Top"
        Width="125"
        RenderTransformOrigin="-0.2,0.45"
        Click="StartButton_Click"/>
        <TextBlock x:Name="TextResult"
           HorizontalAlignment="Left"
           Margin="48,10,0,0"
           TextWrapping="Wrap"
           VerticalAlignment="Top"
           Height="213" Width="420"/>
    </Grid>
</Window>
```

3. Next, open up `MainWindow.xaml.cs`. Go to the **Project Explorer**, and add a reference to `System.Net.Http`.
4. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Documents;
using System.Net.Http;
```

5. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array.

   ```
   char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
   u000A' };
   ```

6. Let's start by creating a helper function that builds a list of `KeyValuePair<string,string>`, which represents our book titles and URLs.

   ```
   private List<KeyValuePair<string, string>> GetBookUrls()
   {
     var urlList = new List<KeyValuePair<string, string>>
     {
       new KeyValuePair<string,string>("Origin of Species",
             "http://www.gutenberg.org/files/2009/2009.txt"),
       new KeyValuePair<string,string>("Beowulf",
             "http://www.gutenberg.org/files/16328/16328-8.txt"),
       new KeyValuePair<string,string>("Ulysses",
             "http://www.gutenberg.org/files/4300/4300.txt")
     };
     return urlList;
   }
   ```

7. Now let's create a `async` method that performs the book download and returns `KeyValuePair<string, int>` that represents our book titles and word count. This method will need to accept a `KeyValuePair<string, string>` parameter representing the book title and URL. The method also needs a `HttpClient` parameter.

   ```
   async Task<KeyValuePair<string,int>> ProcessBook(KeyValuePair<stri
   ng,string> book, HttpClient client)
   {
     var bookContents = await client.GetStringAsync(book.Value);
     var wordArray = bookContents.Split(delimiters,
   StringSplitOptions.RemoveEmptyEntries);
     return new KeyValuePair<string,int>(book.Key,wordArray.Count());
   }
   ```

8. Next, we need to create the `GetWordCount` method. This method will execute a LINQ query to call the `ProcessBook` method on each `book` in the list of books. It then calls `Task.WhenAll` to `await` the tasks completed of all of the tasks. When all tasks have finished, it needs to write the results to the `TextBlock` in a `for` loop.

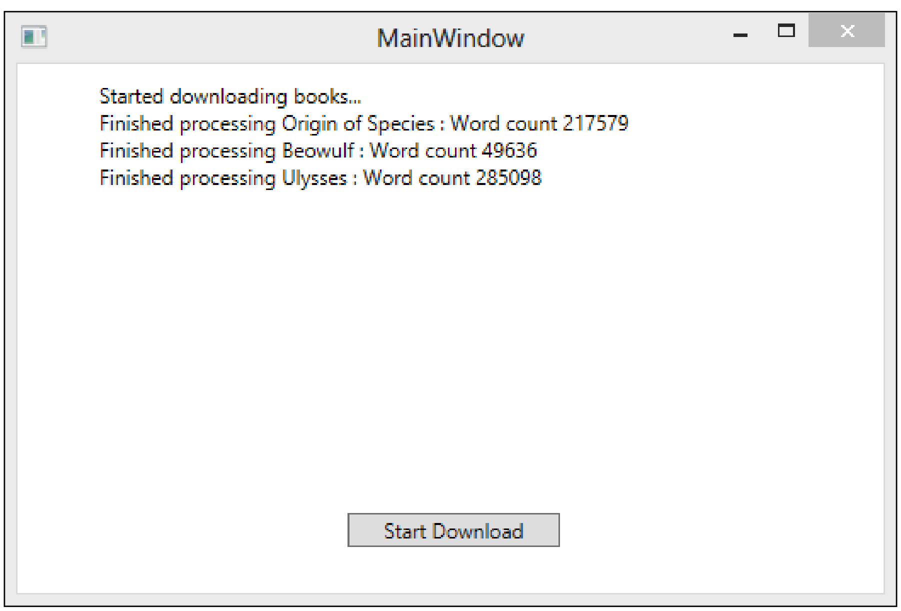
   ```
   public async Task GetWordCount()
   {
     var urlList = GetBookUrls();
   ```

```
  var wordCountQuery = from book in urlList select
ProcessBook(book);
  Task<KeyValuePair<string,int>>[] wordCountTasks =
wordCountQuery.ToArray();
  KeyValuePair<string, int>[] wordCounts = await Task.
WhenAll(wordCountTasks);
  foreach (var book in wordCounts)
  {
    TextResult.Text += String.Format("Finished processing {0} :
Word count {1} \n",
      book.Key, book.Value);
  }
}
```

9. Lastly, the start button click event handler just needs to call the `GetWordCount` method and `await` the task.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{

  TextResult.Text = "Started downloading books...\n";
  Task countTask = GetWordCount();
  await countTask;
}
```

10. In Visual Studio 2012, press *F5* to run the project. Your application should have results as shown in the following screenshot:



How it works...

In this recipe, the `GetWordCount` method calls the `ProcessBook` method for each book in the list by executing a LINQ query. This returns an `IEnumerable<Task<TResult>>`, when we turn in to an array of tasks by calling the `ToArray` method.

```
var urlList = GetBookUrls();
var wordCountQuery = from book in urlList select ProcessBook(book);
var wordCountTasks = wordCountQuery.ToArray();
```

Next, we just `await` a call to the `Task.WhenAll` method which will return when all of the asynchronous tasks complete. Finally, we just use a `for` loop to update the `TextBlock`.

```
var wordCounts = await Task.WhenAll(wordCountTasks);
foreach (var book in wordCounts)
{
  TextResult.Text += String.Format("Finished processing {0} : Word
count {1} \n", book.Key, book.Value);
}
```

Using async for file access

Until now, we have created applications that use `async` for web access, using `HttpClient`. Another common use for `async` is performing asynchronous file I/O without blocking the main thread.

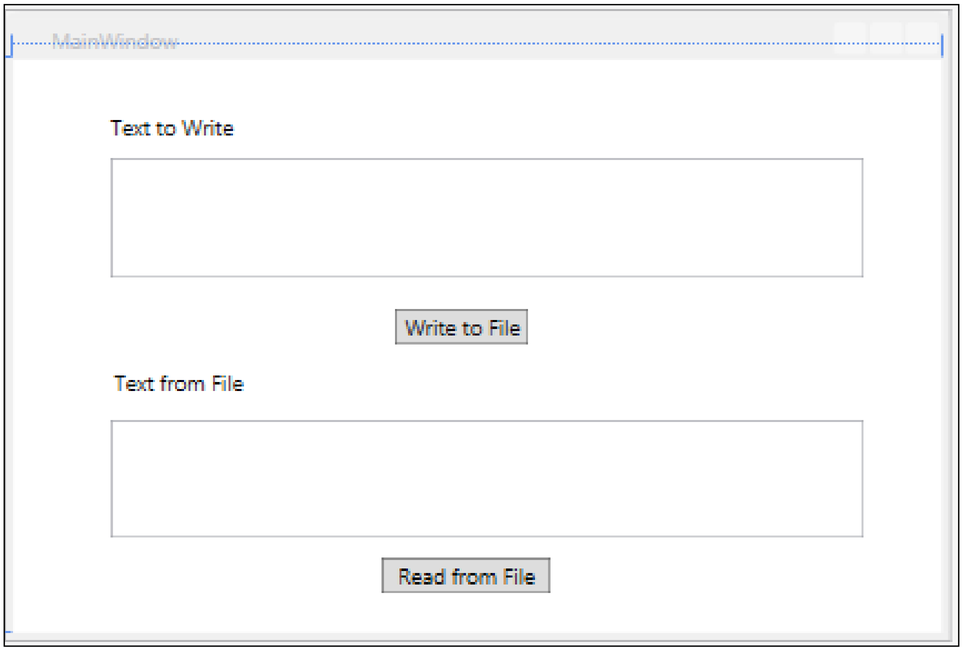
In this recipe, we are going to create a WPF application that can write to and read from a file asynchronously. The application will have two text boxes, one containing the text to write to a file, and the other containing text to read from a file.

How to do it...

1. Start a new project using the **WPF Application** project template and assign `AsyncFileAccess` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and adding the following XAML to create our user interface:

```
<Window x:Class="AsyncFileAccess.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

```
    <Grid>
        <Label Content="Text to Write"
        HorizontalAlignment="Left"
        Margin="49,24,0,0"
        VerticalAlignment="Top"
        Width="87"/>
        <Button x:Name="WriteButton"
        Content="Write to File"
        HorizontalAlignment="Left"
        Margin="212,139,0,0"
        VerticalAlignment="Top"
        Width="75"
        Click="WriteButton_Click"/>
        <TextBox x:Name="TextWrite"
        HorizontalAlignment="Left"
        Height="66" Margin="54,55,0,0"
        TextWrapping="Wrap"
        VerticalAlignment="Top" Width="420"/>
        <TextBox x:Name="TextRead"
        HorizontalAlignment="Left"
        Height="66"
        Margin="54,200,0,0"
        TextWrapping="Wrap"
        VerticalAlignment="Top"
        Width="420"/>
        <Label Content="Text from File"
        HorizontalAlignment="Left"
        Margin="51,167,0,0"
        VerticalAlignment="Top"
        Width="87"/>
        <Button x:Name="ReadButton"
        Content="Read from File"
        HorizontalAlignment="Left"
        Margin="205,277,0,0"
        VerticalAlignment="Top"
        Width="94"
        Click="ReadButton_Click"/>
    </Grid>
</Window>.
```



3. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.IO;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
```

4. At the top of the `MainWindow` class, add a `path` constant for the path of the text file you will be writing. This can be any path you like.

```
string path = @"C:\temp\temp.txt";
```

5. Next, let's create a `async` method called `WriteToFileAsync` that returns `Task`. This method gets a `Unicode` encoded `byte` array of the text in our `TextBox`, creates a file stream, and writes the text to the file.

```
private async Task WriteToFileAsync(string path, string content)
{
  byte[] encodedContent = Encoding.Unicode.GetBytes(content);
  using(FileStream stream = new FileStream(path,FileMode.Append,
       FileAccess.Write,
       FileShare.None,
       bufferSize: 4096,
```

```
            useAsync:true))
        {
          await stream.WriteAsync(encodedContent,
                      0,
                      encodedContent.Length);
        };
}
```

6. Now create an asynchronous `Click` event handler for the `Write` button. Here, we just need to `await` a call to `WriteFileAsync`, passing it the input string.

```
private async void WriteButton_Click(object sender,
RoutedEventArgs e)
{
  WriteButton.IsEnabled = false;
  string content = TextWrite.Text;
  await WriteToFileAsync(path, content);
  WriteButton.IsEnabled = true;
}
```

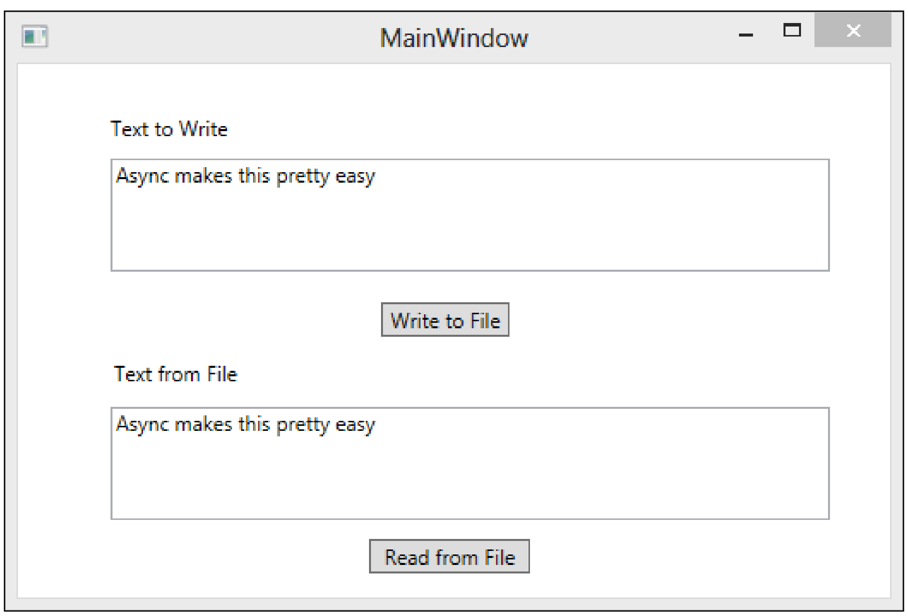
7. Now, let's create an asynchronous method called `ReadFromFileAsync` that returns `Task<string>`. This method creates `FileStream` and reads the contents of the file into `string`.

```
private async Task<string> ReadFromFileAsync(string path)
{
  using (FileStream stream = new FileStream(path,
                  FileMode.Open,
                  FileAccess.Read,
                  FileShare.Read,
                  bufferSize: 4096,
                  useAsync: true))
  {
    var sb = new StringBuilder();
    byte[] buffer = new byte[0x1000];
    int bytesRead;
    while((bytesRead = await stream.ReadAsync(buffer,
                            0,
                            buffer.Length))!=0)
    {
      string content = Encoding.Unicode.GetString(buffer,
                            0,
                            bytesRead);
      sb.Append(content);
    }
    return sb.ToString();
  }
}
```

8. Finally, let's create the read button click handler. This `async` method just needs to check for the existence of the file, and `await` a call to `ReadFromFileAsync`. Set the results of the method call to the proper `TextBox`.

```
private async void ReadButton_Click(object sender, RoutedEventArgs
e)
{
  if (File.Exists(path) == false)
  {
    TextRead.Text = "There was an error reading the file.";
  }
  else
  {
    try
    {
      string content = await ReadFromFileAsync(path);
      TextRead.Text = content;
    }
    catch(Exception ex)
    {
      TextRead.Text = ex.Message;
    }
  }
}
```

9. In Visual Studio 2012, press *F5* to run the project. Your application should appear as shown in the following screenshot:



How it works...

The `Click` event handler is pretty straightforward. It is marked with the `async` keyword because it awaits a call to `WriteToFileAsync`. You must have noticed that we disabled the `Write` button at the start of the method and enabled it again at the end. This is a good practice to control reentrancy with `async` methods. The UI is free to respond to clicks and will fire the `Click` event handler again, if it receives a click.

```
private async void WriteButton_Click(object sender, RoutedEventArgs e)
{
  WriteButton.IsEnabled = false;
  string content = TextWrite.Text;
  await WriteToFileAsync(path, content);
  WriteButton.IsEnabled = true;
}
```

The `WriteToFileAsync` method gets a `Unicode` encoded `byte` array of the input string then creates `FileStream` with `Write` access in the `Append` mode. Once `stream` is open, we `await` a call to the `WriteAsync` method of `FileStream`, passing it our `byte` array.

```
byte[] encodedContent = Encoding.Unicode.GetBytes(content);
using(FileStream stream = new FileStream(path,FileMode.Append,
     FileAccess.Write,FileShare.None,bufferSize: 4096,useAsync:true))
     {
       await stream.WriteAsync(encodedContent,0,encodedContent.
Length);
     };
```

The `ReadFromFileAsync` method just creates `FileStream` in open mode with read access. Once the `stream` is open, we `await` a call to the `ReadAsync` method of `FileStream` in a `while` loop, and read its contents.

```
var sb = new StringBuilder();
byte[] buffer = new byte[0x1000];
int bytesRead;
while((bytesRead = await stream.ReadAsync(buffer, 0, buffer.
Length))!=0)
{
  string content = Encoding.Unicode.GetString(buffer,0,bytesRead);
  sb.Append(content);
}
return sb.ToString();
```

Checking the progress of an asynchronous task

If an asynchronous functionality in your application involves a noticeable delay while the user waits for the result, you might want to inform users that there will be a wait and provide a sense of how long the wait might be. The progress and cancellation features of the `async` programming model enable you to deliver on these needs.

In this recipe, we are going to create a WPF application that uses the progress events of `WebClient` to display the status of a `Download` task with a `ProgressBar`.

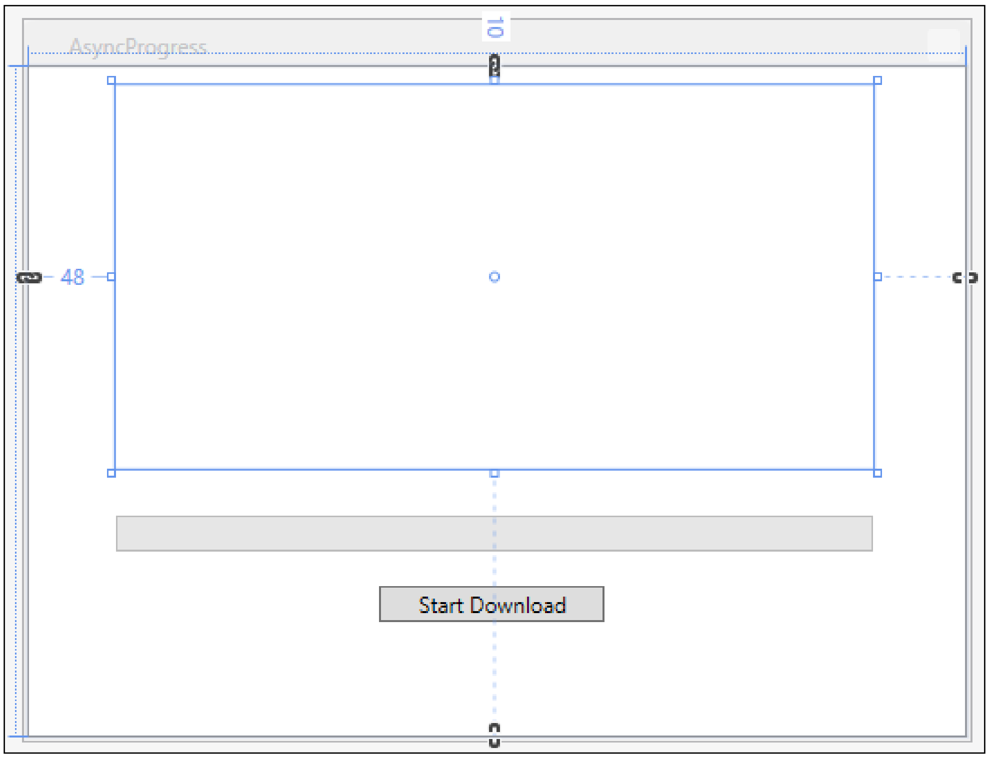
How to do it...

Let's create a WPF application and see how we can add progress reporting to our asynchronous operations.

1. Start a new project using the **WPF Application** project template and assign `AsyncProgress` as **Solution name**.
2. Begin by opening `MainWindow.xaml` and add the following XAML to create our user interface:

```
<Window x:Class="AsyncProgress.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="AsyncProgress" Height="400" Width="525"
ResizeMode="NoResize">
    <Grid>
        <Button x:Name="StartButton"
        Content="Start Download"
        HorizontalAlignment="Left"
        Margin="194,288,0,0"
        VerticalAlignment="Top"
        Width="125"
        RenderTransformOrigin="-0.2,0.45"
        Click="StartButton_Click"/>
        <TextBlock x:Name="TextResult"
        HorizontalAlignment="Left"
        Margin="48,10,0,0"
        TextWrapping="Wrap"
        VerticalAlignment="Top"
        Height="213"
```

```
            Width="420"/>
            <ProgressBar x:Name="DownloadProgress"
            HorizontalAlignment="Left"
            Height="20"
            Margin="48,249,0,0"
            VerticalAlignment="Top"
            Width="420"/>
        </Grid>
    </Window>
```



3. Add the following `using` directives to the top of your `MainWindow` class:

```
using System;
using System.ComponentModel;
using System.Linq;
using System.Net;
using System.Threading.Tasks;
using System.Windows;
```

4. At the top of the `MainWindow` class, add a `character` array constant that will be used to split the contents of the book into a word array. Also add a `string` constant for the user agent header for a `WebClient`.

```
char[] delimiters = { ' ', ',', '.', ';', ':', '-', '_', '/', '\
u000A' };
const string headerText = "Mozilla/5.0 (compatible; MSIE 10.0;
Windows NT 6.1; Trident/6.0)";
```

5. Now let's add a method called `client_DownloadFileCompleted` that will be the event handler for the `DownloadFileCompleted` event of a `WebClient`. This method just needs to add some text to `TextBlock` to indicate that the download has finished.

```
void client_DownloadFileCompleted(object sender,
AsyncCompletedEventArgs e)
{
  TextResult.Text += " Download completed. \n";
}
```

6. Next, create a method called `client_DownloadProgressChanged`. This method will be the event handler for the WebClient's `DownloadProgressChanged` event, and needs to calculate the bytes received, the total bytes, the percentage complete, and update the progress bar.

```
void client_DownloadProgressChanged(object sender,
DownloadProgressChangedEventArgs e)
{
  double bytesIn = double.Parse(e.BytesReceived.ToString());
  double totalBytes = double.Parse(e.TotalBytesToReceive.
ToString());
  double percentage = bytesIn / totalBytes * 100;
  DownloadProgress.Value = int.Parse(Math.Truncate(percentage).
ToString());
}
```

7. Now we need to create our asynchronous `GetWordCountAsync` method. This method returns `Task<int>`, and after creating `WebClient`, and wiring up the `DownloadFileCompletedEvent` and `DownloadProgressChanged` events, it awaits a call to the `DownloadStringTaskAsync` method of `WebClient` to download the contents of the book and split the words into an array of strings.

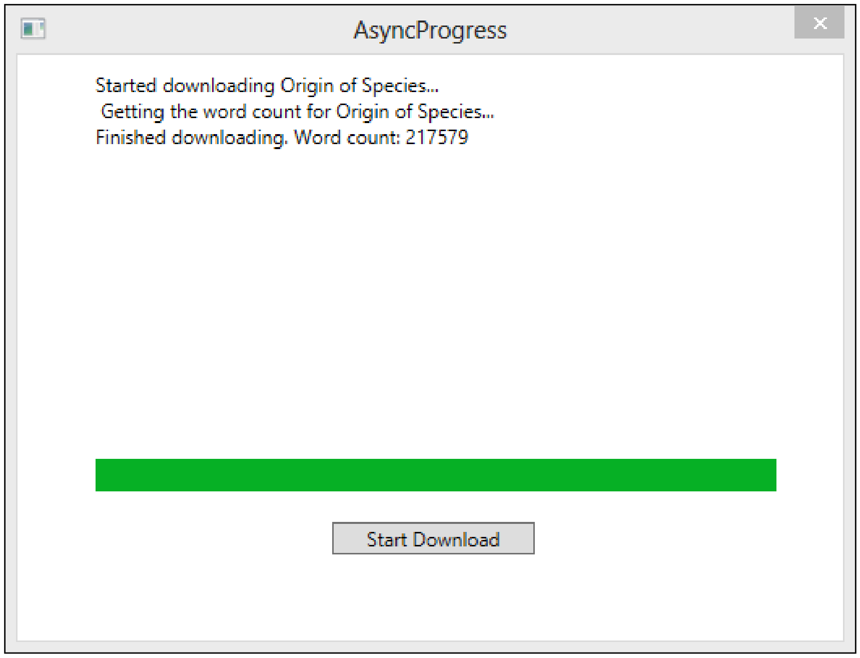
```
public async Task<int> GetWordCount()
{
  TextResult.Text += " Getting the word count for Origin of
Species...\n";
  var client = new WebClient();
```

```
  client.Headers.Add("user-agent", headerText);
  client.DownloadProgressChanged += new
      DownloadProgressChangedEventHandler(client_
DownloadProgressChanged);
  client.DownloadFileCompleted +=
      new AsyncCompletedEventHandler(client_
DownloadFileCompleted);
  Task<string> wordsTask =
    client.DownloadStringTaskAsync(new Uri("http://www.gutenberg.
org/files/2009/2009.txt"));
  var words = await wordsTask;
  var wordArray = words.Split(delimiters, StringSplitOptions.
RemoveEmptyEntries);
  return wordArray.Count();
}
```

8. Finally, let's create an asynchronous `Click` event handler for `StartButton`. This button just writes some text to `TextBlock` and awaits a call to `GetWordCountAsync`.

```
private async void StartButton_Click(object sender,
RoutedEventArgs e)
{
  TextResult.Text += "Started downloading Origin of Species...\n";
  Task<int> countTask = GetWordCountAsync();
  int result = await countTask;
  TextResult.Text += String.Format("Finished downloading. Word
count: {0}\n", result);
}
```

9. In Visual Studio 2012, press *F5* to run the project. Your application should display the results as shown in the following screenshot:



How it works...

This application was able to show the progress of its download by wiring up two events of the `WebClient` class: `DownloadProgressChanged` and `DownloadFileCompleted`. It then calls the `DownloadStringTaskAsync` method of `WebClient`, which triggers the events as the download progresses.

```
public async Task<int> GetWordCountAsync()
{
  ...
  client.DownloadProgressChanged +=
      new DownloadProgressChangedEventHandler(client_
DownloadProgressChanged);
  client.DownloadFileCompleted +=
      new AsyncCompletedEventHandler(client_DownloadFileCompleted);
  Task<string> wordsTask =
      client.DownloadStringTaskAsync(new Uri("http://www.gutenberg.
org/files/2009/2009.txt"));
  ...
}
```

The event handler for `DownloadFileCompleted` is pretty self-explanatory. The event handler for `DownloadProgressChanged` is where the calculation of the progress actually happens. Each time the event fires, we get the number of bytes the `WebClient` has received, the total number of bytes to receive, and we calculate the percentage completed of the download. Finally, we set the `Value` property of `ProgressBar` with the results of the `percentage` calculation.

```
void client_DownloadProgressChanged(object sender,
DownloadProgressChangedEventArgs e)
{
  double bytesIn = double.Parse(e.BytesReceived.ToString());
  double totalBytes = double.Parse(e.TotalBytesToReceive.ToString());
  double percentage = bytesIn / totalBytes * 100;
  DownloadProgress.Value = int.Parse(Math.Truncate(percentage).
ToString());
}
```

9
Dataflow Library

In this chapter, we will cover the following recipes:

- Reading from and writing to a dataflow block synchronously
- Reading from and writing to a dataflow block asynchronously
- Implementing a producer-consumer dataflow pattern
- Creating a dataflow pipeline
- Cancelling a dataflow block
- Specifying the degree of parallelism
- Unlink dataflow blocks
- Using `JoinBlock` to read from multiple data sources

Introduction

The Task Parallel Library's dataflow is a new library that is designed to increase the robustness of highly concurrent applications. TPL dataflow uses asynchronous message passing and pipelining to obtain more control and better performance than manual threading.

A dataflow consists of a series of blocks. Each block can be a source or target for data. Data typically enters into a dataflow by being posted to a propagation block, which is a block that implements `ISourceBlock<T>` and `ITargetBlock<T>`. The source block can be linked to other target or propagation blocks. The data flows from one block to the next block in the chain asynchronously. The data is buffered at the source or target block until it is needed.

The predefined blocks fall into three categories. There are buffering blocks which hold data for use by data consumers, there are execution blocks that call a user-provided delegate for each piece of received data, and there are grouping blocks which combine data from one or more sources and under various constraints.

The TPL dataflow library provides three types of buffering blocks. There is the System.Threading.Tasks.Dataflow.BufferBlock<T> class, the System.Threading.Tasks.Dataflow.BroadcastBlock<T> class, and the System.Threading.Tasks.Dataflow.WriteOnceBlock<T> class. The BufferBlock<T> class is a general-purpose asynchronous messaging class. BufferBlock<T> stores a **First-In- First-Out** (**FIFO**) queue of messages that can be written to by multiple sources or read from by multiple targets.

The BroadcastBlock<T> class is useful when you pass multiple messages to another component, or a message to multiple components.

The WriteOnceBlock<T> class is similar the BroadcastBlock<T> class, except that a WriteOnceBlock<T> object can be written one time only.

The TPL dataflow library provides three types of execution blocks. There is the ActionBlock<TInput> class, the System.Threading.Tasks.Dataflow.TransformBlock<TInput, TOutput> class, and the System.Threading.Tasks.Dataflow.TransformManyBlock<TInput, TOutput> class.

The ActionBlock<TInput> class is a target block that calls a delegate when it receives data. You can think of a ActionBlock<TInput> object as a delegate that runs asynchronously when data becomes available.

The TransformBlock<TInput, TOutput> class resembles the ActionBlock<TInput> class, except that it acts both as a source and as a target.

The TransformManyBlock<TInput, TOutput> class resembles the TransformBlock<TInput, TOutput> class, except that TransformManyBlock<TInput, TOutput> produces zero or more output values for each input value, instead of only one output value for each input value.

The TPL dataflow library also provides three types of join blocks. There is the BatchBlock<T> class, the JoinBlock<T1, T2> class, and the BatchedJoinBlock<T1, T2> class.

The BatchBlock<T> class combines sets of input data, which are known as batches, into arrays of output data.

The JoinBlock<T1, T2> and JoinBlock<T1, T2, T3> classes collect input elements and propagate out System.Tuple<T1, T2> or System.Tuple<T1, T2, T3> objects that contain those elements.

The `BatchedJoinBlock<T1, T2>` and `BatchedJoinBlock<T1, T2, T3>` classes collect batches of input elements and propagate out the `System.Tuple(IList(T1), IList(T2))` or `System.Tuple(IList(T1), IList(T2), IList(T3))` objects that contain those elements.

The Dataflow library's infrastructure is built on .NET 4.5's Task Parallel Library. These dataflow components are useful when you have multiple operations that must communicate with one another asynchronously or when you want to process data as it becomes available.

Reading from and writing to a dataflow block synchronously

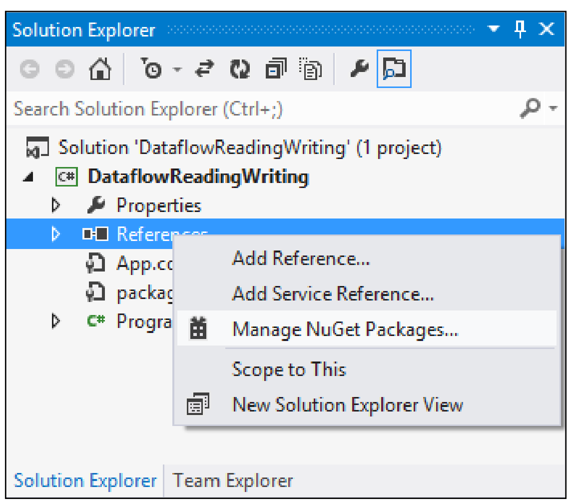
Writing a message synchronously to a dataflow block is done by calling the `Post<TInput>` method of a block. Let's use the `Receive` method of the block to receive data.

In this recipe, we are going to create a `Console` application that uses a `for` loop to synchronously write some numbers to `BufferBlock` using the `Post` method. The application then reads the data back from `BufferBlock` using the `Receive` method and writes the data to `Console`.

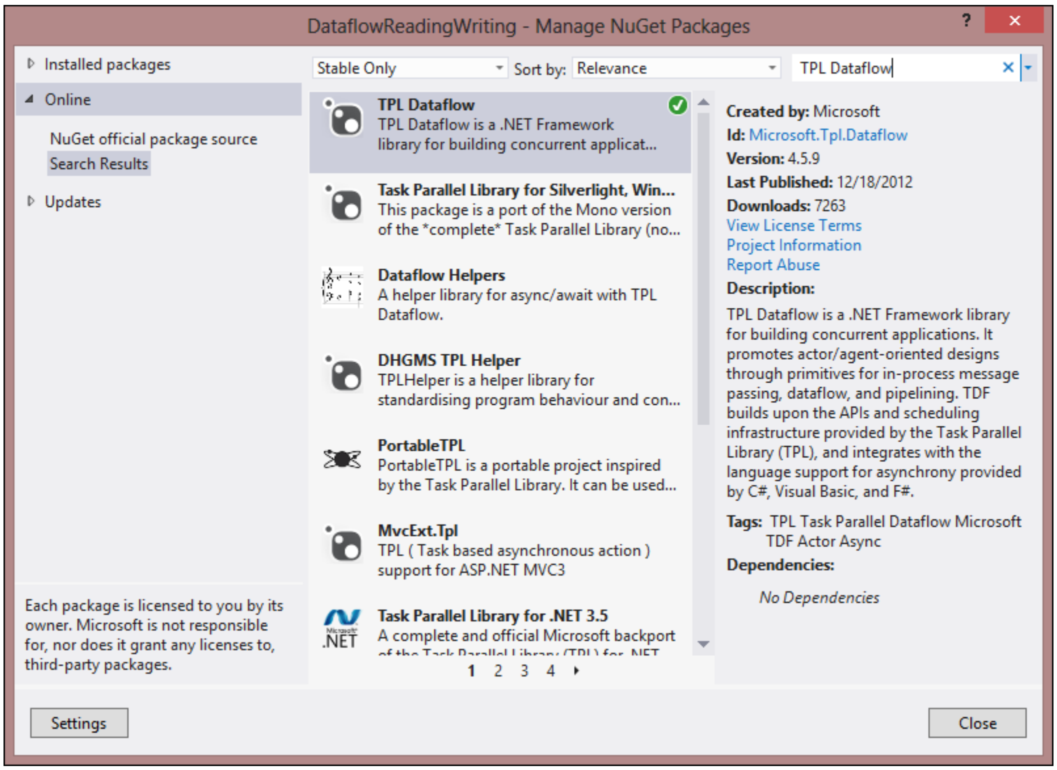
Getting ready...

The TPL dataflow library doesn't ship with the rest of the TPL. To use the TPL dataflow library in your solutions, you need to use NuGet Package Manager to set your reference.

1. After creating your new project in Visual Studio 2012, go to the **Solution Explorer**, right-click on **References**, and click on **Manage NuGet Package**.



2. In the NuGet Package Manager window, click on **Online** from the menu on the left and search for **TPL Dataflow**.



How to do it...

Let's create a new `Console` application so we can see how to create a dataflow block and write to it synchronously.

1. Start a new project using the **Console Application** project template and assign `Dataflow ReadingWriting` as the **Solution name**.
2. Next, go to the **Solution Explorer**, right-click on **References**, click on **Manage NuGetPackages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class.

```
using System;
using System.Threading.Tasks.Dataflow;
```

4. In the `Main` method of the `Program` class, create a `BufferBlock<int>` object.

```
var bufferingBlock = new BufferBlock<int>();
```

5. Now let's create a `for` loop that loops from zero to ten and writes the square of the loop index to the buffer block using the `Post` method.

```
for (int i = 0; i < 10; i++)
{
  bufferingBlock.Post(i*i);
}
```

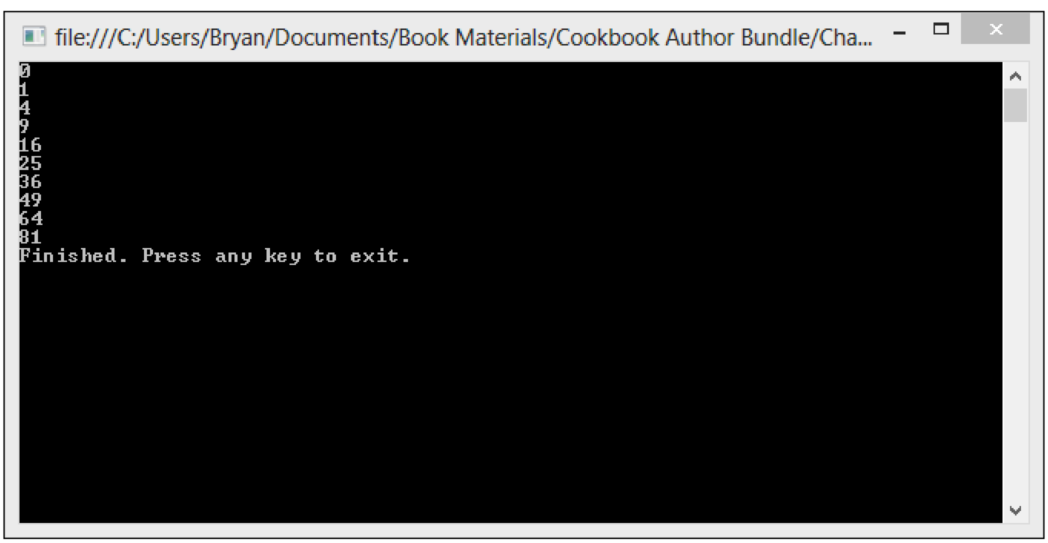
6. Now, let's create another `for` loop that loops from zero to ten that calls the `Receive` method on the buffer block for each iteration, and writes the results to `Console`.

```
for (int i = 0; i < 10; i++)
{
  Console.WriteLine(bufferingBlock.Receive.ToString());
}
```

7. Finish up by writing a message to the `Console` application and waiting for user input before exiting.

```
Console.WriteLine("Finished. Press any key to exit.");
Console.ReadLine();
```

8. In Visual Studio 2012, press *F5* to run the project. Click on the **Start** button, and your application should appear as shown in the following screenshot:



How it works...

This simple example shows how to read from and write to a message block directly. More often, you will be connecting dataflow blocks to form pipelines, or linear sequences of blocks.

Writing to a block directly is a pretty easy matter; you just need to call the `Post<TInput>` method.

```
bufferingBlock.Post(i*i);
```

The `Post` method acts synchronously, and returns once the target block has decided to accept or reject the item.

Conversely, data can be directly received from `bufferingBlock` by calling the `Receive` method.

```
bufferingBlock.Receive()
```

The `Receive` method has a few convenient overloads that can accept a time out period, `CancellationToken` or both.

Reading from and writing to a dataflow block asynchronously

Writing a message asynchronously to a dataflow block is done by calling the `SendAsync<TInput>` method of a block. You use the `ReceiveAsync` method of the block to receive data.

In this recipe, we are going to create a `Console` application that uses a `for` loop to asynchronously write some numbers to `BufferBlock` using the `SendAsync` method. The application then reads the data back from `BufferBlock` using the `ReceiveAsync` method and writes the data to `Console`.

How to do it...

1. Start a new project using the **Console Application** project template and assign `Dataflow ReadWriteAsync` as the **Solution name**.
2. Next, go to **Solution Explorer**, right-click on **references**, click on **Manage NuGet Packages** and add a reference to the TPL Dataflow library.

3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Threading.Tasks.Dataflow;
using System.Threading.Tasks;
```

4. Now let's create a `static async` method that returns `Task` called `WriteDataAsync`. The method takes a `BufferBlock<int>` parameter, and uses a `for` loop to iterate from zero to ten. In each iteration of the loop, we need to use the `SendAsync` method to write the square of the loop indexer to `bufferingBlock`.

```
private static async Task WriteDataAsync(BufferBlock<int>
bufferingBlock)
{
  // Post some messages to the block.
  for (int i = 0; i < 10; i++)
  {
    await bufferingBlock.SendAsync(i * i);
  }
}
```

5. Next, let's create a `static async` method that returns `Task` called `ReadDataAsync`. The method takes a `BufferBlock<int>` parameter, and uses a `for` loop to read the data from `BufferBlock` and display it to `Console`.

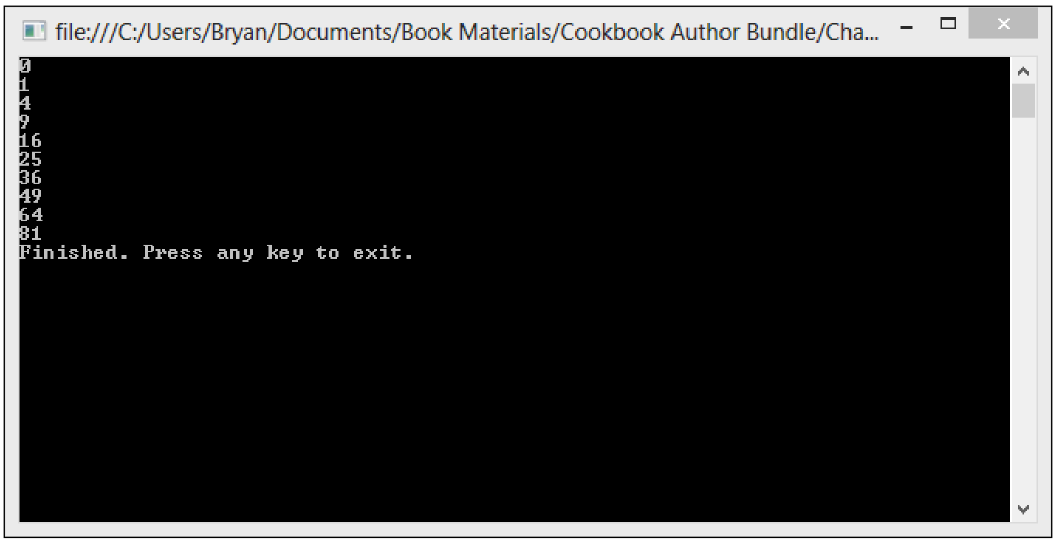
```
private static async Task ReadDataAsync(BufferBlock<int>
bufferingBlock)
{
  // Receive the messages back .
  for (int i = 0; i < 10; i++)
  {
    Console.WriteLine(await bufferingBlock.ReceiveAsync());
  }
}
```

6. Finally, let's create the implementation of the `Main` method. Here we just need to wait on calls to `WriteDataAsync` and `ReadDataAsync`. We also need to wait on user input before exiting.

```
static void Main(string[] args)
{
  // Create a BufferBlock object.
  var bufferingBlock = new BufferBlock<int>();
```

```
        WriteDataAsync(bufferingBlock).Wait();
        ReadDataAsync(bufferingBlock).Wait();

        Console.WriteLine("Finished. Press any key to exit.");
        Console.ReadLine();
      }
```

7. Now, in Visual Studio 2012, press *F5* to run the project. Click on the **Start** button. Your application should appear as shown in the following screenshot:



How it works...

This application uses the `SendAsync` method to asynchronously write to a `BufferBlock<int>` object and the `ReceiveAsync` method to read from the same object. We also use the `async` and `await` operators to send data to and read data from the target block.

Notice that both the `ReadDataAsync` and `WriteDataAsync` methods are marked as `async`, with a return type of `Task`. We use the `await` keyword to asynchronously make the call to `SendAsync` and `ReceiveAsync`.

```
private static async Task ReadDataAsync(BufferBlock<int>
bufferingBlock)
{
  for (int i = 0; i < 10; i++)
```

```
    {
      Console.WriteLine(await bufferingBlock.ReceiveAsync());
    }
  }
```

The `ReceiveAsync` method is especially useful when you want to act on data as the data becomes available.

Implementing a producer-consumer dataflow pattern

TPL dataflow blocks can also be used in a producer-consumer pattern, where a producer sends messages to a block, and the consumer reads messages from a block.

In this recipe, we are going to create a `Console` application to demonstrate a basic producer-consumer pattern that uses dataflow. The producer will use a `for` loop to create some random numbers and add them to `BufferBlock<int>`. The consumer task will asynchronously receive the data from `BufferBlock` as it becomes available, and returns a sum of all the numbers.

How to do it...

Let's create another `Console` application and see how we can use dataflow blocks to implement a producer-consumer pattern.

1. Start a new project using the **Console Application** project template and assign `Dataflow ProducerConsumer` as the **Solution name**.
2. Next, go to **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class.

   ```
   using System;
   using System.Threading.Tasks.Dataflow;
   using System.Threading.Tasks;
   ```

4. Let's start by creating a `static` method on the `Program` class called `Produce`. This method returns `void`, and takes a parameter of type `ITargetBlock<int>`. This method will use a `for` loop to generate random numbers, and then use the `Post` method to send them to the block. When the `Produce` method is finished adding, it calls the `Complete` method on the block.

   ```
   static void Produce(ITargetBlock<int> target)
   ```

```
{
  // Create a Random object.
  Random rand = new Random();

  // fill a buffer with random data
  for (int i = 0; i < 100; i++)
  {
    // get the next random number
    int number = rand.Next();

    // Post the result .
    target.Post(number);
  }

  // Set the target to the completed state
  target.Complete();
}
```

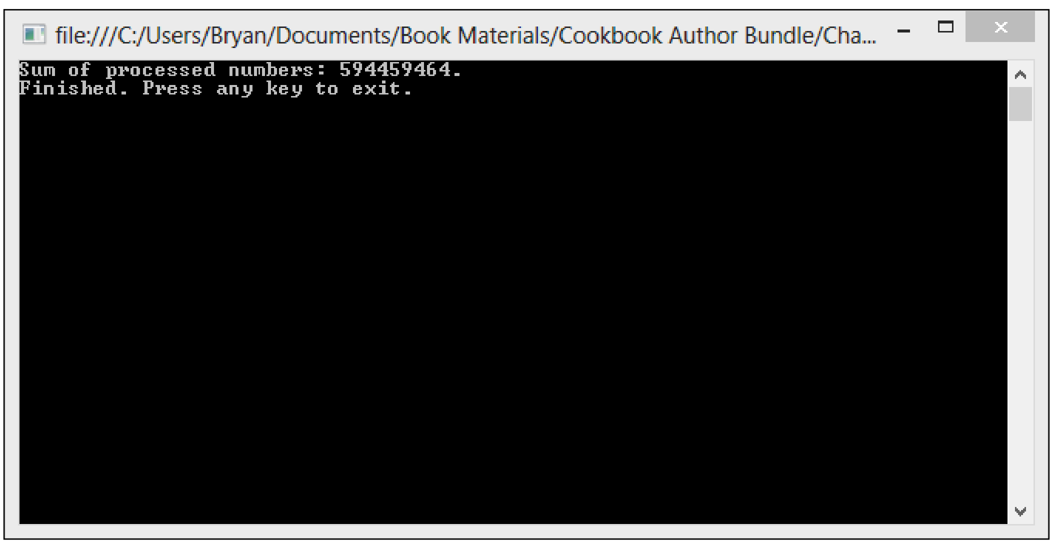
5. Next, let's create a method called `ConsumeAsync`. As you probably guessed from the method name, this is an `async` method that returns `Task<int>`. The `ConsumeAsync` method needs a parameter type of `ISourceBlock<int>`. This method used a `while` loop to get data from the block as it becomes available, and produces a sum of the numbers.

```
static async Task<int> ConsumeAsync(ISourceBlock<int> source)
{
  // Initialize a counter to track the sum.
  int sumOfProcessed = 0;

  // Read from the source buffer until empty
  while (await source.OutputAvailableAsync())
  {
    int data = source.Receive();

    // calculate the sum.
    sumOfProcessed += data;
  }

  return sumOfProcessed;
}
```

6. Ok, let's finish up by implementing the `Main` method of the `Program` class. This method just needs to start `producer` and `consumer`, and display the results when finished.

```
static void Main(string[] args)
{
  var buffer = new BufferBlock<int>();

  // Start the consumer.
  var consumer = ConsumeAsync(buffer);

  // Post source data.
  Produce(buffer);

  // Wait for the consumer to process data.
  consumer.Wait();

  // Print the count of bytes processed to the console.
  Console.WriteLine("Sum of processed numbers: {0}.", consumer.
Result);
  Console.WriteLine("Finished. Press any key to exit.");
  Console.ReadLine();
}
```

7. In Visual Studio 2012, press *F5* to run the project. Your application should appear as shown in the following screenshot:



How it works...

The `Produce` method is very straightforward. We declared a parameter of the interface type `ITargetBlock<TInput>`. `ITargetBlock` is an interface implemented by `BufferBlock` that represents a dataflow block that is a target for data.

The `Produce` method just uses a `for` loop to send data to the target block using the `Post` method. After it is finished adding the data, it calls the complete method to signal that it is finished.

```
static void Produce(ITargetBlock<int> target)
{
  ...
  for (int i = 0; i < 100; i++)
  {
    ...
    target.Post(number);
  }.
  target.Complete();
}
```

`ConsumeAsync` accepts a parameter of the interface type `ISourceBlock`, which represents a dataflow block that is a source of data. To act asynchronously, the `ConsumeAsync` method calls the `OutputAvailiableAsync` method to receive a notification when the source block has data available when the source block is finished, and will never have additional data. Other than that, it just uses the `Receive` method to receive the data and sums up the results.

```
static async Task<int> ConsumeAsync(ISourceBlock<int> source)
{
  ...
  while (await source.OutputAvailableAsync())
  {
    int data = source.Receive();
      sumOfProcessed += data;
  }
  return sumOfProcessed;
}
```

Creating a dataflow pipeline

As we have seen so far, you can use `Post`, `Receive`, and `RecieveAsync` to send and receive messages from source blocks. You can also connect message blocks to form a dataflow pipeline. A dataflow pipeline is a chain of dataflow blocks, each of which performs a specific task and contributes to a larger goal. Each block in the pipeline performs its work when it receives a message from another dataflow block.

In this recipe, we are to return to our WordCount example one final time. We are going to create a `Console` application that forms a dataflow pipeline for downloading the contents of a classic book, filters out the small words from the book contents, and returns a count of the words.

How to do it...

Now, let's see how we can chain dataflow blocks together to form a pipeline.

1. Start a new project using the **Console Application** project template and assign `DataflowPipeline` as the **Solution name**.
2. Next, go to the **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Linq;
using System.Threading.Tasks.Dataflow;
using System.Net;
```

4. The first step is to create the dataflow blocks that participate in the pipeline. In the `Main` method, create `TransformBlock<string,string>` that takes the string input parameter and uses `WebClient` to download the book contents as a string.

```
// Download a book as a string
var downloadBook = new TransformBlock<string, string>(url =>
{
  Console.WriteLine("Downloading the book...");

  return new WebClient().DownloadString(url);
});
```

5. Next, let's add `TransformBlock<string, string[]>` which receives the output from the previous block, removes the spaces, and splits the words into a string array.

```
// splits text into an array of strings.
var createWordList = new TransformBlock<string, string[]>(text =>
{
  Console.WriteLine("Creating list of words...");

  // Remove punctuation
  char[] tokens = text.ToArray();
  for (int i = 0; i < tokens.Length; i++)
  {
    if (!char.IsLetter(tokens[i]))
      tokens[i] = ' ';
  }
  text = new string(tokens);

  return text.Split(new char[] { ' ' },
     StringSplitOptions.RemoveEmptyEntries);
});
```

6. Ok, let's create `TransformBlock<string[],int>` that filters out words less than three characters and returns a count of the words.

```
// Remove short words and return the count
var filterWordList = new TransformBlock<string[], int>(words =>
{
  Console.WriteLine("Counting words...");

  var wordList = words.Where(word => word.Length > 3).OrderBy(word
=> word)
     .Distinct().ToArray();
  return wordList.Count();
});
```

7. Finally, let's create `ActionBlock<int>` to display the word count to `Console`.

```
var printWordCount = new ActionBlock<int>(wordcount =>
{
  Console.WriteLine("Found {0} words",
     wordcount);
});
```

8. Now, let's use the `LinkTo` method to connect the source blocks and target blocks to form the pipeline.

```
downloadBook.LinkTo(createWordList);
createWordList.LinkTo(filterWordList);
filterWordList.LinkTo(printWordCount);
```

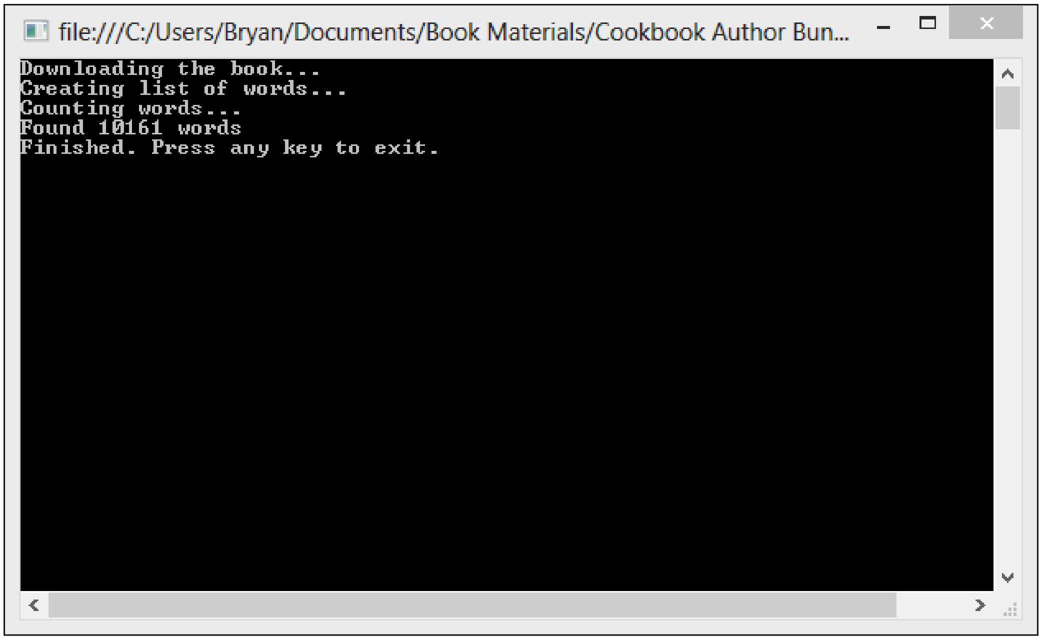
9. Next, we need to add some completion tasks to enable each dataflow block to perform a final action after processing all data elements.

```
downloadBook.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)createWordList).Fault(t.
Exception);
  else createWordList.Complete();
});
createWordList.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)filterWordList).Fault(t.
Exception);
  else filterWordList.Complete();
});
filterWordList.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)printWordCount).Fault(t.
Exception);
  else printWordCount.Complete();
});
```

10. Finally, let's add some code to `Main` to post the data to the pipeline, complete the pipeline activity, wait for the pipeline to finish, and wait for user input before exiting.

```
// Download Origin of Species
downloadBook.Post("http://www.gutenberg.org/files/2009/2009.txt");

// Mark the head of the pipeline as complete.
downloadBook.Complete();

printWordCount.Completion.Wait();

Console.WriteLine("Finished. Press any key to exit.");
Console.ReadLine();
```

11. In Visual Studio 2012, press *F5* to run the project. Your application should display the output as shown in the following screenshot:



How it works...

In this application we used TransformBlock<TInput, TOutput> to enable each member of the pipeline to perform an operation on its input data and send the results to the next step in the pipeline. For example, downloadBook TransformBlock takes a string input and returns a string output to the next step.

```
var downloadBook = new TransformBlock<string, string>(uri =>
{
  ...
  return new WebClient().DownloadString(uri);
});
```

The only exception is the tail of the pipeline is ActionBlock<TInput> because it performs an action on its input and does not produce a result.

The next step is to connect each block in the pipeline to the next. The LinkTo method of DataflowBlock is used to connect ISourceBlock<TOutput> to TargetBlock<TInput>. When you call the LinkTo method to connect a source to a target, the source will propagate data to the target as it becomes available.

```
downloadBook.LinkTo(createWordList);
createWordList.LinkTo(filterWordList);
filterWordList.LinkTo(printWordCount);
```

We also added some completion tasks to propagate completion through the pipeline. Each completion task sets the next dataflow block to the completed state.

```
downloadBook.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)createWordList).Fault(t.
Exception);
  else createWordList.Complete();
});
```

Finally, we used `DataflowBlock.Post<TInput>` to synchronously send data to the head of the pipeline. The following is the URL string of the book we are downloading:

```
downloadBook.Post("http://www.gutenberg.org/files/2009/2009.txt");
```

Cancelling a dataflow block

Since dataflow blocks are built on the `Task` infrastructure of the TPL, cancellation is supported by obtaining `CancellationToken` from `CancellationTokenSource`.

In this recipe, we will create a dataflow pipeline to download the contents of a classic book and perform a word count, except this time, we will enable the blocks that form the pipeline to be cancelled.

How to do it...

Let's see how we can add cancellation to our dataflow blocks.

1. Start a new project using the **Console Application** project template and assign `CancelDataflow` as the **Solution name**.
2. Next, go to the **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

   ```
   using System;
   using System.Linq;
   using System.Threading.Tasks.Dataflow;
   using System.Net;
   using System.Threading;
   ```

4. In the `Main` method, create a new `CancellationTokenSource` object.

   ```
   var cancellationSource = new CancellationTokenSource();
   ```

5. Next, let's create the blocks that form the pipeline. The blocks are exactly as before, except, this time they are created with a new `ExecutionDataflowBlockOptions` parameter that sets `CancellationToken`.

   ```
   // Download a book as a string
   var downloadBook = new TransformBlock<string, string>(uri =>
   {
     Console.WriteLine("Downloading the book...");

     return new WebClient().DownloadString(uri);
   },
   new ExecutionDataflowBlockOptions
   {
     CancellationToken = cancellationSource.Token
   });



   // splits text into an array of strings.
   var createWordList = new TransformBlock<string, string[]>(text =>
   {
     Console.WriteLine("Creating list of words...");

     // Remove punctuation
     char[] tokens = text.ToArray();
     for (int i = 0; i < tokens.Length; i++)
     {
       if (!char.IsLetter(tokens[i]))
         tokens[i] = ' ';
     }
     text = new string(tokens);
     return text.Split(new char[] { ' ' },
        StringSplitOptions.RemoveEmptyEntries);
   },
   new ExecutionDataflowBlockOptions
   {
     CancellationToken = cancellationSource.Token
   });

   // Remove short words and return the count
   ```

```
var filterWordList = new TransformBlock<string[], int>(words =>
{
  Console.WriteLine("Counting words...");

  var wordList = words.Where(word => word.Length > 3).OrderBy(word
=> word)
      .Distinct().ToArray();
  return wordList.Count();
},
new ExecutionDataflowBlockOptions
{
  CancellationToken = cancellationSource.Token
});

var printWordCount = new ActionBlock<int>(wordcount =>
{
  Console.WriteLine("Found {0} words",
      wordcount);
},
new ExecutionDataflowBlockOptions
{
  CancellationToken = cancellationSource.Token
});
```

6. Now, let's use the `LinkTo` method to connect the source blocks and target blocks to form the pipeline.

```
downloadBook.LinkTo(createWordList);
createWordList.LinkTo(filterWordList);
filterWordList.LinkTo(printWordCount);
```

7. Next, we need to add some completion tasks to enable each dataflow block to perform a final action after processing all data elements.

```
downloadBook.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)createWordList).Fault(t.
Exception);
  else createWordList.Complete();
});
createWordList.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)filterWordList).Fault(t.
Exception);
  else filterWordList.Complete();
```

```
});
filterWordList.Completion.ContinueWith(t =>
{
  if (t.IsFaulted) ((IDataflowBlock)printWordCount).Fault(t.
Exception);
  else printWordCount.Complete();
});
```

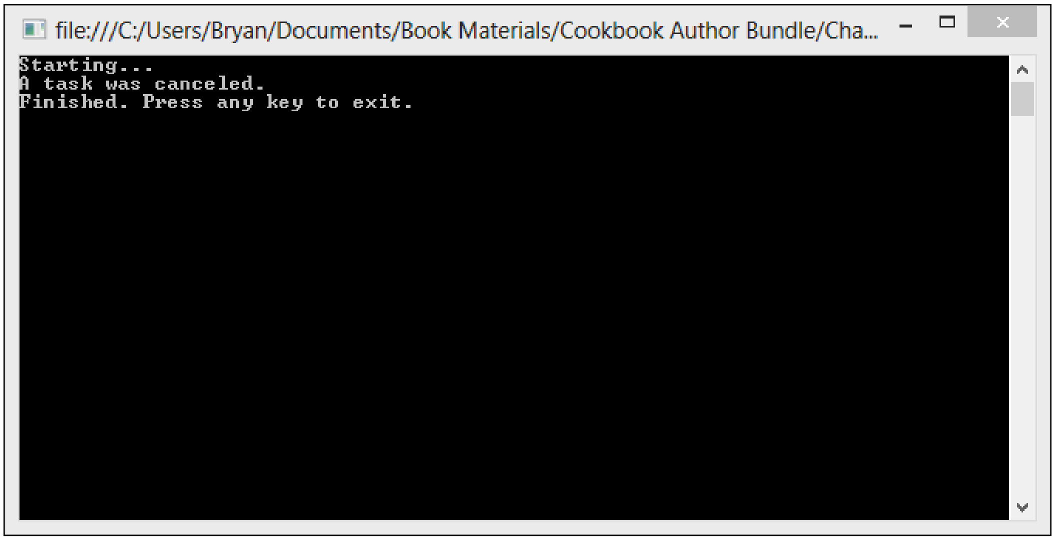
8. Now, add a `try` block to post the data to the head of the pipeline, complete the pipeline activity, and cancel the token.

```
try
{
  Console.WriteLine("Starting...");

  // Download Origin of Species
  downloadBook.Post("http://www.gutenberg.org/files/2009/2009.
txt");

  // Mark the head of the pipeline as complete.
  downloadBook.Complete();

  // Cancel the operation
  cancellationSource.Cancel();

  printWordCount.Completion.Wait();

}
```

9. Finally, add a `catch` block that handles `AggregateException` and a `finally` block to wait for user input before exiting.

```
catch (AggregateException ae)
{
  foreach (Exception ex in ae.InnerExceptions)
  {
    Console.WriteLine(ex.Message);
  }
}
finally
{
  Console.WriteLine("Finished. Press any key to exit.");
  Console.ReadLine();
}
```

10. In Visual Studio 2012, press *F5* to run the project. Your application should appear as shown in the following screenshot:



How it works...

This application forms a dataflow pipeline to process the contents of a book and return a word count as in the previous recipe. The difference is that we set up cancellation by creating a `CancellationTokenSource` object, and then setting the `CancellationToken` property of the `ExecutionDataflowBlockOptions` object associated with the blocks in our pipeline.

```
CancellationTokenSource cancellationSource = new
CancellationTokenSource();

var downloadBook = new TransformBlock<string, string>(uri =>
{
  ...

  return new WebClient().DownloadString(uri);
},
new ExecutionDataflowBlockOptions
{
  CancellationToken = cancellationSource.Token
});
```

We set the `CancellationToken` property of the `ExecutionDataflowBlockOptions` object to `CancellationToken` obtained from the `CancellationTokenSource` object.

As with all other cancellations in the TPL, we need to handle the `OperationCancelled` exception, which will be wrapped in a `AggregateException` object.

```
catch (AggregateException ae)
{
  foreach (Exception ex in ae.InnerExceptions)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Specifying the degree of parallelism

In the previous recipe, we saw how to use the `CancellationToken` property of the `ExecutionDataflowBlockOptions` object to enable cancellation of a pipeline. In this recipe, we will see how to use the `MaxDegreeOfParallelism` property to enable dataflow blocks to process more than one message at a time.

We are going to create a `Console` application that performs two dataflow calculations, and prints the elapsed time of each calculation. The first calculation sets the maximum degree of parallelism to one. The second operation is the same as the first, but sets the maximum degree of parallelism to the number of available processors on your machine.

How to do it...

Now, let's see how to add cancellation to our dataflow blocks.

1. Start a new project using the **Console Application** project template and assign `DegreeOfParallelism` as the **Solution name**.
2. Next, go to the **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

   ```
   using System;
   using System.Diagnostics;
   using System.Threading;
   using System.Threading.Tasks.Dataflow;
   ```

4. Let's start by creating a `static` method on the `Program` class called `ComputeTime`. The method needs to accept an `integer` parameter for `maxDegreeOfParallelism` and an `integer` parameter for `messageCount`.

```
static TimeSpan ComputeTime(int maxDegreeOfParallelism, int
messageCount)
{

}
```

5. In the ComputeTime method, create ActionBlock<int> that just sleeps for the time period of the integer parameter. ActionBlock also needs to use a ExecutionDataflowBlockOptions parameter and set the MaxDegreeOfParallelism property to the value of the method parameter.

```
var actionBlock = new ActionBlock<int>(
  millisecondsTimeout => Thread.Sleep(millisecondsTimeout),
  new ExecutionDataflowBlockOptions
  {
     MaxDegreeOfParallelism = maxDegreeOfParallelism
  });
```

6. Let's finish up the ComputeTime method by creating a Stopwatch object, using a for loop to post data to the action block, complete the action block, and return the elapsed time.

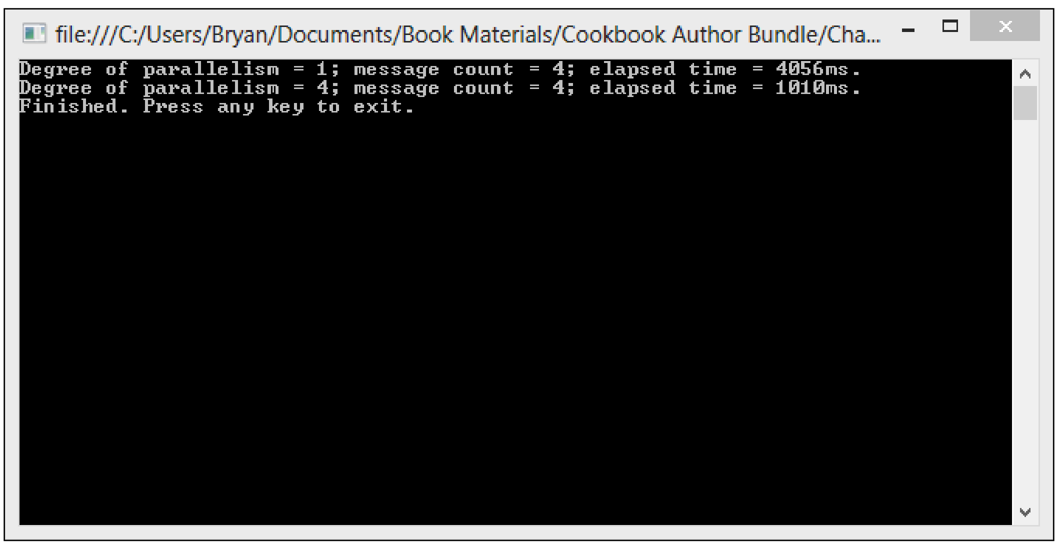
```
Stopwatch sw = new Stopwatch();
sw.Start();

for (int i = 0; i < messageCount; i++)
{
      actionBlock.Post(1000);
}
actionBlock.Complete();
actionBlock.Completion.Wait();
sw.Stop();

return sw.Elapsed;
```

7. Next, we need to implement the Main method. Let's start by getting the processor count of your machine; call the ComputeTime method twice (once with the maxDegreeOfParallelism parameter set to one, and once with the maxDegreeOfParallelism parameter set to your processor count), and display the results.

```
static void Main(string[] args)
{
  int processorCount = Environment.ProcessorCount;
  int messageCount = processorCount;
```

```
    TimeSpan elapsedTime;
    elapsedTime = ComputeTime(1, messageCount);
    Console.WriteLine("Degree of parallelism = {0}; message count =
  {1}; " +
        "elapsed time = {2}ms.", 1, messageCount, (int)elapsedTime.
  TotalMilliseconds);

    elapsedTime = ComputeTime(processorCount, messageCount);
    Console.WriteLine("Degree of parallelism = {0}; message count =
  {1}; " +
        "elapsed time = {2}ms.", processorCount, messageCount, (int)
  elapsedTime.TotalMilliseconds);

    Console.WriteLine("Finished. Press any key to exit.");
    Console.ReadLine();
  }
```

8. In Visual Studio 2012, press *F5* to run the project. Your application should have results as shown in the following screenshot:



How it works...

In this recipe, we just use the `ComputeTime` method to set the maximum degree of parallelism of `ActionBlock<int>` to the value of the `maxDegreeOfParallelism` method parameter.

```
static TimeSpan ComputeTime(int maxDegreeOfParallelism, int
messageCount)
{
  var actionBlock = new ActionBlock<int>(
     millisecondsTimeout => Thread.Sleep(millisecondsTimeout),
     new ExecutionDataflowBlockOptions
     {
       MaxDegreeOfParallelism = maxDegreeOfParallelism
     });
   ...

}
```

We call this method twice, once with the `maxDegreeOfParallelism` parameter set to one, and once with the `maxDegreeOfParallelism` parameter set to the processor count of your machine.

A maximum degree of parallelism of one causes the dataflow block to process messages serially, and a degree of parallelism of greater than one enables the dataflow block to process messages in parallel.

Unlink dataflow blocks

We have previously seen how to link dataflow blocks together to form a pipeline. This recipe is going to show how to unlink a dataflow block from its source.

We are going to show how to unlink a dataflow block by creating a `Console` application that creates three transform blocks, each of which calls a method to perform a calculation. The transform block objects will each be linked to a `WriteOnceBlock<T>` object, with the `MaxMessages` property set to one. This will instruct the source blocks to unlink after the first message is received at the target.

How to do it...

1. Start a new project using the **Console Application** project template and assign `DegreeOfParallelism` as the **Solution name**.
2. Next, go to **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.
3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;
```

4. First, let's create a `static` method called `DoCalculation`. This method accepts `integer` and `CancellationToken` as parameters, and returns `integer`. This method is going to simulate a lengthy calculation that takes a few seconds to complete, then returns a somewhat arbitrary value.

```
static int DoCalculation(int n, CancellationTokenSource
tokenSource)
{
  // simulate a workload and return result
  SpinWait.SpinUntil(() => tokenSource.IsCancellationRequested,
    new Random().Next(2000));
  return n + 5;
}
```

5. Next, let's create another `static` method called `ReceiveFromAny<T>`. This method will take a parameter array of `ISourceBlock<T>` and return `T`. This method will receive a value from the first source in the `source` array that returns a value. It will create new `WriteOnceBlock<T>` and link it to each source block, with a `DataFlowLinkOptions` parameter and the `MaxMessages` property set to one. Finally, it will receive the value produced by `WriteOnceBlock`.

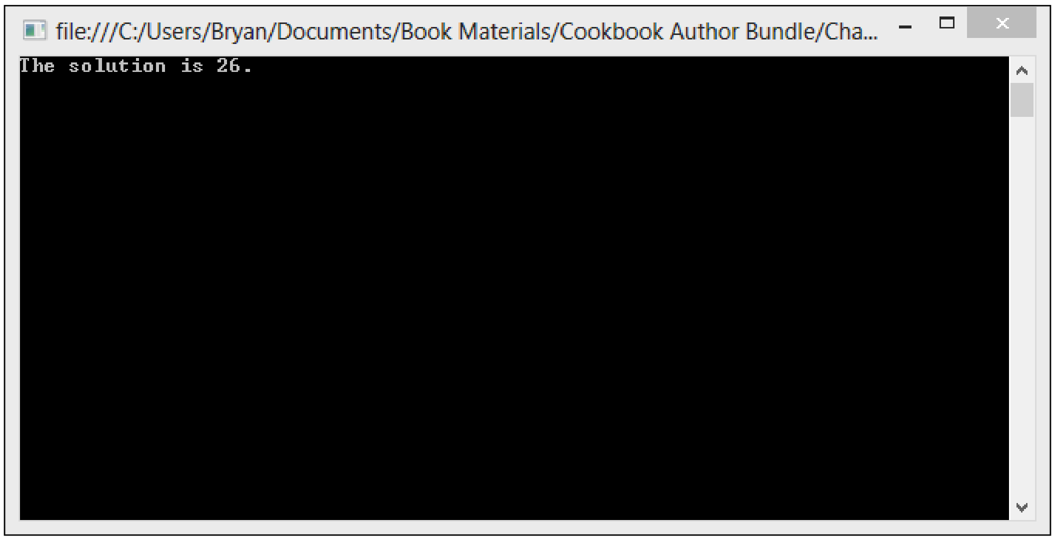
```
public static T ReceiveFromAny<T>(params ISourceBlock<T>[]
sources)
{
  var writeOnceBlock = new WriteOnceBlock<T>(e => e);
  foreach (var source in sources)
  {
    source.LinkTo(writeOnceBlock, new DataflowLinkOptions {
MaxMessages = 1 });
```

```
  }
  return writeOnceBlock.Receive();
}
```

6. Finally, let's implement the `Main` method. `Main` needs to create a new `CancellationToken` object, create three `System.Threading.Tasks.Dataflow.TransformBlock` objects that each call the `DoCalculation` method, posts data to each of `TransformBlocks`, and receives the result.

```
static void Main(string[] args)
{

  try
  {
    var tokenSource = new CancellationTokenSource();

    Func<int, int> action = n => DoCalculation(n, tokenSource);
    var calculation1 = new TransformBlock<int, int>(action);
    var calculation2 = new TransformBlock<int, int>(action);
    var calculation3 = new TransformBlock<int, int>(action);

    calculation1.Post(11);
    calculation2.Post(21);
    calculation3.Post(31);

    int result = ReceiveFromAny(calculation1, calculation2,
calculation3);

    // Cancel all calls to TrySolution that are still active.
    tokenSource.Cancel();

    // Print the result to the console.
    Console.WriteLine("The solution is {0}.", result);
  }
  catch (AggregateException) { }
  finally { Console.ReadLine(); }

}
```

7. In Visual Studio 2012, press *F5* to run the project. Your application should appear as shown in the following screenshot:



How it works...

The `Main` method of this application is responsible for creating our `TransformBlock` objects, creating `Func` to call the `DoCalculation` method, posting some data to `TransformBlocks`, and calling `RecieveFromAny` with `TransformBlocks` as parameter.

```
var tokenSource = new CancellationTokenSource();

Func<int, int> action = n => DoCalculation(n, tokenSource);
var calculation1 = new TransformBlock<int, int>(action);
var calculation2 = new TransformBlock<int, int>(action);
var calculation3 = new TransformBlock<int, int>(action);

calculation1.Post(11);
calculation2.Post(21);
calculation3.Post(31);

int result = ReceiveFromAny(calculation1, calculation2, calculation3);
```

The `RecieveFromAny` method creates a new `WriteOnceBlock<T>` object, uses a `for` loop to link the `WriteOnceBlocks` to the source `TransformBlocks`, and receives the first data the `WriteOnceBlock` produces.

```
public static T ReceiveFromAny<T>(params ISourceBlock<T>[] sources)
{
  var writeOnceBlock = new WriteOnceBlock<T>(e => e);
  foreach (var source in sources)
  {
    source.LinkTo(writeOnceBlock, new DataflowLinkOptions {
MaxMessages = 1 });
  }
  return writeOnceBlock.Receive();
}
```

The link to the source blocks is created using the `LinkTo` method as before, but this time a new `DataflowLinkOptions` object is used as a parameter, with the `MaxMessages` property set to one.

The `MaxMessages` property is used to set the maximum number of messages that may be consumed across a link.

Using JoinBlock to read from multiple data sources

This recipe is going to show how to use `JoinBlock` to perform an operation when data is available from multiple sources.

We are going to create a `Console` application that defines two types of resources: `NetworkResource` and `MemoryResource`. We will use the `NetworkResource` and `MemoryResource` pair to perform an operation. To enable the operation to occur when both required resources are available, we will use `JoinBlock<T1, T2>`.

How to do it...

Let's see how to use `JoinBlock` to perform an operation based on data from multiple sources.

1. Start a new project using the **Console Application** project template and assign `JoinBlock` as the **Solution name**.
2. Next, go to **Solution Explorer**, right-click on **References**, click on **Manage NuGet Packages**, and add a reference to the **TPL Dataflow** library.

3. Open up `Program.cs` and add the following `using` directives to the top of your `Program` class:

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;
```

4. Inside the `Program` class above the `Main` method, create an `abstract` class definition for resource, and a concrete `class` definition for `MemoryResource` and `NetworkResource`.

```
abstract class Resource
{
}

class MemoryResource : Resource
{
}

class NetworkResource : Resource
{
}
```

5. Now, in the `Main` method, create a `BufferBlock<MemoryResource>` and a `BufferBlock<NetworkResource>` object.

```
var networkResources = new BufferBlock<NetworkResource>();
var memoryResources = new BufferBlock<MemoryResource>();
```

6. Next, create a `JoinBlock<NetworkResource, MemoryResource>`. Create a `GroupingDataflowBlockOptions` object parameter and set the `Greedy` property to false.

```
var joinResources =
   new JoinBlock<NetworkResource, MemoryResource>(
    new GroupingDataflowBlockOptions
    {
      Greedy = false
    });
```

7. Now we need to create a `ActionBlock<Tuple<NetworkResource, MemoryResource>>` object to simulate `NetworkResource` doing a lengthy network access operation.

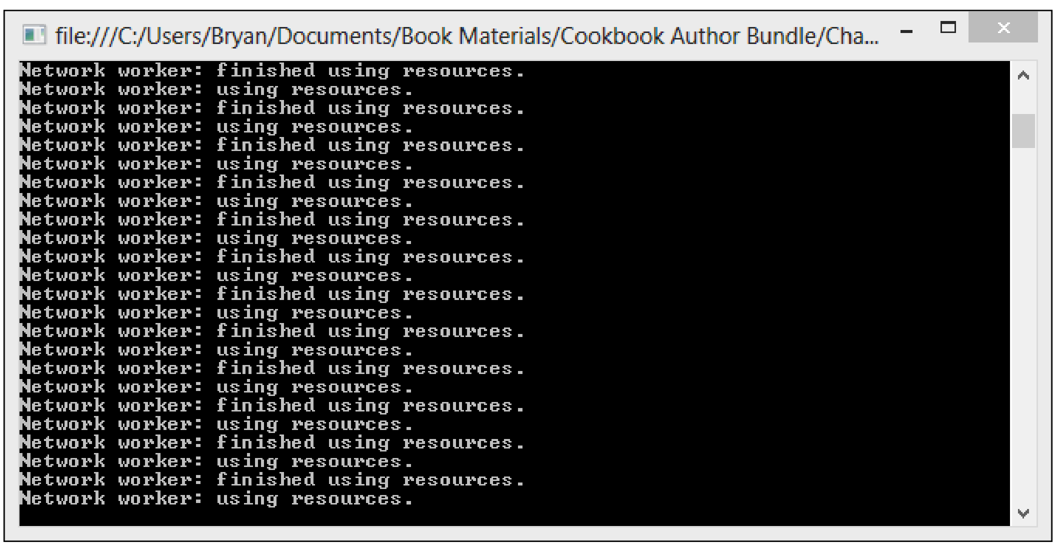
```
var networkMemoryAction =
   new ActionBlock<Tuple<NetworkResource, MemoryResource>>(
    data =>
```

```
      {
        Console.WriteLine("Network worker: using resources.");
        Thread.Sleep(new Random().Next(500, 2000));
        Console.WriteLine("Network worker: finished using
resources.");
        networkResources.Post(data.Item1);
        memoryResources.Post(data.Item2);
      });
```

8. Finally, let's finish up the `Main` method by linking the resource objects, linking `JoinBlock` to `ActionBlock`, and posting data to the resource blocks.

```
networkResources.LinkTo(joinResources.Target1);
memoryResources.LinkTo(joinResources.Target2);

joinResources.LinkTo(networkMemoryAction);

networkResources.Post(new NetworkResource());
networkResources.Post(new NetworkResource());
networkResources.Post(new NetworkResource());

memoryResources.Post(new MemoryResource());

Thread.Sleep(10000);
Console.ReadLine();
```

9. In Visual Studio 2012, press *F5* to run the project. Your application should display results as shown in the following screenshot:



How it works...

This application starts by creating two `BufferBlock<T>` objects; one holds network resources and one holds memory resources.

```
var networkResources = new BufferBlock<NetworkResource>();
var memoryResources = new BufferBlock<MemoryResource>();
```

Then we created a non-greedy `JoinBlock` to join the network resources to the memory resources by setting the `Greedy` property to `false`. In the default greedy mode, the join block will greedily take the data from the source, but it still won't produce a result tuple until all necessary data is available. This is primarily important when sources are connected to multiple join blocks. If all of the joins take data greedily from the sources, you can end up in situations where data would be available to satisfy one of the joins, but end up being taken greedily and split across multiple joins such that none of them could be satisfied, and thus, none will produce results until more data comes along.

```
var joinResources =
   new JoinBlock<NetworkResource, MemoryResource>(
    new GroupingDataflowBlockOptions
    {
      Greedy = false
   });
```

The next step is to create `ActionBlock` that operates network and memory resources. The action just simulates a lengthy operation on a network resource and then releases the resources back to their pools.

```
var networkMemoryAction =
   new ActionBlock<Tuple<NetworkResource, MemoryResource>>(
    data =>
    {
      ...
      Thread.Sleep(new Random().Next(500, 2000));
      ...
      networkResources.Post(data.Item1);
      memoryResources.Post(data.Item2);
    });
```

Finally, we link our resources together, populate our resource pools and allow data to flow through for a few seconds.

```
networkResources.LinkTo(joinResources.Target1);
memoryResources.LinkTo(joinResources.Target2);
```

```
joinResources.LinkTo(networkMemoryAction);

networkResources.Post(new NetworkResource());
networkResources.Post(new NetworkResource());
networkResources.Post(new NetworkResource());

memoryResources.Post(new MemoryResource());

Thread.Sleep(10000);
```

Index

Symbols

A

B

C

D

E

F

Z

Thank you for buying
.NET 4.5 Parallel Extensions Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

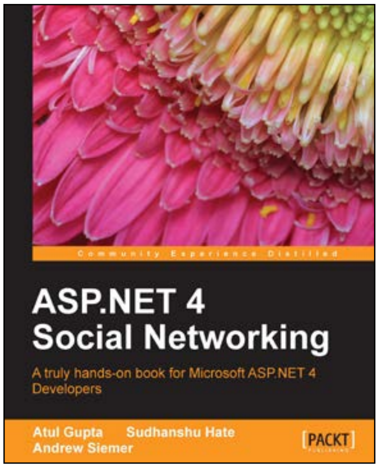
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

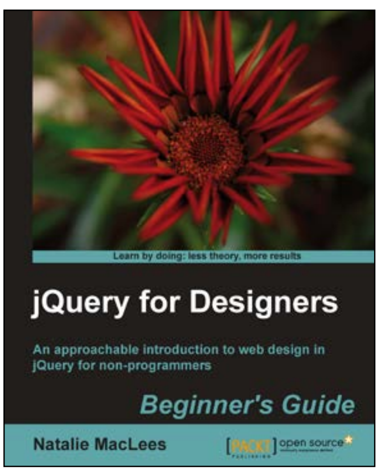


ASP.NET 4 Social Networking

ISBN: 978-1-849690-82-9 Paperback: 484 pages

A truly hands-on book for Microsoft ASP.NET 4 Developers

1. Create a full-featured, enterprise-grade social network using ASP.NET 4.0
2. Build friends lists, messaging systems, user profiles, blogs, forums, groups, and more
3. Learn key new ASP.NET and .NET Framework concepts like Managed Extensibility Framework (MEF), Entity Framework 4.0, LINQ, AJAX, C# 4.0, ASP.NET Routing,n-tier architectures, and MVP in a practical, hands-on way.



Visual Studio 2012 and .NET 4.5 Expert Development Cookbook

ISBN: 978-1-849686-70-9 Paperback: 380 pages

Over 40 recipes for successfully mixing the powerful capabilities of .NET 4.5 and Visual Studio 2012

1. Step-by-step instructions to learn the power of .NET development with Visual Studio 2012
2. Filled with examples that clearly illustrate how to integrate with the technologies and frameworks of your choice
3. Each sample demonstrates key conceptsto build your knowledge of the architecture in a practical and incremental way

Please check **www.PacktPub.com** for information on our titles