

Quick answers to common problems

C++ Multithreading Cookbook

Over 60 recipes to help you create ultra-fast multithreaded applications using C++ with rules, guidelines, and best practices

Miloš Ljumović

[PACKT] open source*
PUBLISHING community experience distilled

C++ Multithreading Cookbook

Over 60 recipes to help you create ultra-fast multithreaded applications using C++ with rules, guidelines, and best practices

Miloš Ljumović

[PACKT] open source 
PUBLISHING community experience distilled

BIRMINGHAM - MUMBAI

C++ Multithreading Cookbook

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2014

Production reference: 1250714

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

ISBN 978-1-78328-979-0

www.packtpub.com

Cover image by Radivoje Todorović (mr.todorovic@gmail.com)

Credits

Author

Miloš Ljumović

Project Coordinator

Mary Alex

Reviewers

Abhishek Gupta
Venkateshwaran Loganathan
Walt Stoneburner
Dinesh Subedi

Proofreaders

Simran Bhogal
Kevin McGowan
Chris Smith

Commissioning Editor

Edward Gordon

Indexers

Hemangini Bari
Mehreen Deshmukh
Rekha Nair

Acquisition Editor

Nikhil Karkal

Graphics

Disha Haria

Content Development Editor

Akshay Nair

Production Coordinator

Komal Ramchandani

Technical Editors

Tanvi Bhatt
Faisal Siddiqui

Cover Work

Komal Ramchandani

Copy Editors

Janbal Dharmaraj
Karuna Narayanan
Sayanee Mukherjee
Alfida Paiva

About the Author

Miloš Ljumović was born on July 26 in Podgorica, the capital of Montenegro in Europe, where he attended elementary and high school. He also went to music school to play the guitar. He studied Computer Science in the Faculty of Natural Science and Mathematics, Montenegro University. The following year, he picked up interest in operating systems, which he majored in, in his Master's degree. In December 2009, along with his friend Danijel, he started his company, Programmer, along with a highly qualified team to provide high-tech IT solutions. Soon, many skilled developers joined them, and from this collaboration, many applications and system software, web applications, and database systems have arisen. His clients were not only the Government of Montenegro, but also some major domestic companies. He developed a new age financial system for the national domain company, MeNet; he also developed video recognition software, along with pictures and other digital media types. He also developed many fascinating websites and other web applications. The list of customers is too long to be incorporated here.

After working only for a few months as an international consultant, he worked with an American company in a project involving large e-commerce businesses and data scraping from the Internet. All this was the spark to open his new company, EXPERT.ITS.ME, in the beginning of July 2014. Besides development, he provides consulting services and helps small businesses in the IT sector to manage problems while inspiring them to be and think big. He is also a member of the Committee of National Chamber (ICT) of Montenegro and MENSA. He likes programming in C/C++/C# even though he is skilled in HTML, PHP, TSQL, and more while going toward his dream of developing his own operating system.

In his spare time, he plays tennis, dives, hunts, or plays chess. He favors brainstorming with his team and fresh, modern ideas in the area of IT and computer science. He is continuously going towards new beginnings and next-generation software solutions. He specially likes teaching computer science and math students through private classes and courses and shaping them to be qualified programmers and helping them find all the beauty of science. To see what his interests are and what he is doing, visit his company's website (<http://expert.its.me>) or his website (<http://milos.expert.its.me>), or contact him at milos@expert.its.me.

Acknowledgments

I am grateful to many for writing this book.

I dedicate this book to my parents Radoslav and Slavka and to my sisters Natalija and Dušanka, who are always there for me no matter what. I am specially grateful to my mother, without whom I wouldn't become a programmer.

An even greater thanks goes to my beautiful wife, Lara, for putting up with me, for all her patience and love towards me, for all her unconditional support, and for teaching me never to give up. Volim te.

My appreciation goes to my dear friend Danijel who taught me how to be great businessman and pushed me over the edge to be better programmer each day.

I would also like to thank my professors from the University of Montenegro, without whom I wouldn't be the expert I am today. A special thanks to Rajko Čalasan for endless hours teaching me programming, Milo Tomašević for providing me with expertise in object-orienting programming, and making me love C++ the way I do today, and especially to Goran Šuković, the greatest teacher of all, for constantly guiding me and teaching me various areas of computer science and making me wish and get more and more knowledge each day.

About the Reviewers

Abhishek Gupta is a young embedded software engineer from Bangalore, India. He has been working on software for Automotive In-Vehicle Infotainment (IVI) for the past few years. He completed his MTech from IIT Kharagpur, India in Visual Information and Embedded Systems in 2011. He is passionate about video processing and loves to work on embedded multimedia systems. His technical knowledge revolves mostly around C and Linux.

You can find more information about him at www.abhitak.wordpress.com/about-me.

Venkateshwaran Loganathan is an eminent software developer who has been involved in the design, development, and testing of software products for more than 6 years now. He was introduced to computer programming at an early age of 11 with FoxPro, and then started to learn and master various computer languages such as C, C++, Perl, Python, Node.js, and Unix shell scripting. Fascinated by open source development, he has involved himself in contributing to various Open Source technologies.

He is now working for Cognizant Technology Solutions as an associate in technology, where he has involved himself in research and development for the Internet of Things domain. He is now actively involved in using RFID devices to evolve Future of Technology concepts. Before joining Cognizant, he had worked with some of the major IT firms such as Infosys, Virtusa, and NuVeda. Starting his career as a network developer, he has gained expertise in various domains such as networking, e-learning, and healthcare. He has won various awards and accolades in the companies he has worked for.

He holds a Bachelor's degree in Computer Science and Engineering from Anna University and is currently pursuing an M.S. in Software Systems from BITS, Pilani. Apart from programming, he is actively involved in handling various technical and soft skills classes for budding engineers and college students. He also likes singing and trekking. He likes to get involved in social service and moves with people a lot. Visit him online at <http://www.venkateshwaranloganathan.com> and write to him at anandvenkat4@gmail.com.

He has also published the book PySide GUI Application Development, Packt Publishing.

I am indebted to many. First of all, I would like to thank my mother, Anbuselvi, and grandmother, Saraswathi, for their endless effort and perseverance in bringing me up to this level. I would also like to thank all my friends and brothers; the list is too long to mention here. They all have been my well-wishers and have helped me in my tough times. I have not mentioned the names of many people here, but my thanks are always due to those who directly or indirectly influenced my life.

Above all, thanks to the Almighty for the showering his blessings on me.

Walt Stoneburner is a software architect with over 25 years of commercial application development and consulting experience. Fringe passions involve quality assurance, configuration management, and security. If cornered, he may actually admit to liking statistics and authoring documentation as well.

He's easily amused by programming language design, collaborative applications, big data, knowledge management, data visualization, and ASCII art. Self-described as a closet geek, Walt also evaluates software products and consumer electronics, draws comics, runs a freelance photography studio specializing in portraits and art (CharismaticMoments.com), writes humor pieces, performs sleights of hand, enjoys game design, and can occasionally be found on ham radio.

Walt may be reached directly via e-mail at wls@wwco.com or at Walt.Stoneburner@gmail.com. He publishes a tech and humor blog called the Walt-O-Matic at <http://www.wwco.com/~wls/blog/>.

His other book reviews and contributions include:

- ▶ *AntiPatterns and Patterns in Software Configuration Management* (ISBN 978-0-471-32929-9, p. xi)
- ▶ *Exploiting Software: How to Break Code* (ISBN 978-0-201-78695-8, p. xxxiii)
- ▶ *Ruby on Rails: Web Mashup Projects* (ISBN 978-1-847193-93-3)
- ▶ *Building Dynamic Web 2.0 Websites with Ruby on Rails* (ISBN 978-1-847193-41-4)
- ▶ *Instant Sinatra Starter* (ISBN 978-1782168218)
- ▶ *Learning Selenium Testing Tools with Python* (978-1-78398-350-6)
- ▶ *Whittier* (ASIN B00GTD1RBS)
- ▶ *Cooter Brown's South Mouth Book of Hillbilly Wisdom* (ISBN 978-1-482340-99-0)

Dinesh Subedi is a software developer at Yomari Incorporated Pvt. Ltd. He is currently working on data warehouse technology and business intelligence. He is a blogger at www.codeincodeblock.com writing articles related to software development using C++ and has four years experience with it. He has completed his B.E. in Computer Engineering from Pulchowk Campus IOE Kathmandu, Nepal.

I would like to thank my brother Bharat Subedi who helped me while reviewing this book.

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.



<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	1
Chapter 1: Introduction to C++ Concepts and Features	7
Introduction	7
Creating a C++ project	8
Program structure, execution flow, and runtime objects	10
The structural programming approach	14
Understanding the object-oriented programming approach	17
Explaining inheritance, overloading, and overriding	19
Understanding polymorphism	24
Event handlers and Message Passing Interface	28
Linked list, queue, and stack examples	34
Chapter 2: The Concepts of Process and Thread	45
Introduction	45
Processes and threads	46
Explaining the process model	47
Implementation of processes	51
IPC – Interprocess Communication	55
Solving classical IPC problems	65
Implementation of the thread model	75
Thread usage	81
Implementing threads in user space	89
Implementing threads in the kernel	99
Chapter 3: Managing Threads	105
Introduction	105
Processes versus threads	106
Permissive versus preemptive multitasking	110
Explaining the Windows Thread object	111
Basic thread management	112

Implementing threads without synchronization	121
Using synchronized threads	127
Win32 synchronization objects and techniques	133
Chapter 4: Message Passing	141
Introduction	141
Explaining the Message Passing Interface	142
Understanding a message queue	147
Using the thread message queue	156
Communicating through the pipe object	161
Chapter 5: Thread Synchronization and Concurrent Operations	167
Introduction	167
Pseudoparallelism	168
Understanding process and thread priority	168
The Windows dispatcher object and scheduling	176
Using mutex	177
Using semaphore	187
Using event	197
Using critical section	206
Using pipes	215
Chapter 6: Threads in the .NET Framework	233
Introduction	233
Managed versus unmanaged code	234
How threading works in .NET	235
The difference between foreground and background threads	243
Understanding the .NET synchronization essentials	247
Locking and avoiding deadlocks	254
Thread safety and types of the .NET framework	261
Signaling with event wait handles	263
Event-based Asynchronous Pattern	269
Using the BackgroundWorker class	276
Interrupting, aborting, and safe canceling the thread execution	281
Non-blocking synchronization	291
Signaling with Wait and Pulse	294
The Barrier class	299

Chapter 7: Understanding Concurrent Code Design	309
Introduction	309
How to design parallel applications	310
Understanding parallelism in code design	316
Turning on to a parallel approach	324
Improving the performance factors	332
Chapter 8: Advanced Thread Management	341
Introduction	341
Using thread pools	342
Customizing the thread pool dispatcher	354
Using remote threading	371
Appendix	381
Installing MySQL Connector/C	381
Installing WinDDK – Driver Development Kit	384
Setting up a Visual Studio project for driver compilation	386
Using the DebugView application	392
Setting up a Visual Studio project for OpenMP compilation	393
Index	397

Preface

Creating multithreaded applications is a present-day approach towards programming. Developers expect their applications to be user friendly, with a rich interface and concurrent execution. The power of the C++ language alongside the native Win32 API features will give you a head start over all other languages and frameworks. With the power of C++, you can easily create various types of applications and perform parallelism and optimizations in your existing work.

This book is a practical, powerful, and easy-to-understand guide to C++ multithreading. You will learn how to benefit from the multithreaded approach and enhance your development skills to build better applications. This book will not only help you avoid problems when creating parallel code, but also help you understand synchronization techniques in detail. The book also covers the Windows process model alongside scheduling techniques and Interprocess Communication.

Starting from the basics, you will be introduced to the most powerful Integrated Development Environment ever made, that is Microsoft Visual Studio. You will then learn to use the native features of a Windows kernel as well as the characteristics of the .NET framework. You will then understand how to solve some common concurrent problems and learn how to properly think in a multithreaded environment.

Using mutexes, semaphores, critical sections, monitors, events, and pipes, you will learn the most efficient way of synchronization in your C++ application. The book will teach you the best possible approach to learn concurrency in C++.

Using the C++ native calls, the book will show you how to leverage machine hardware for optimum performance. The end goal of the book is to impart various multithreading concepts that will enable you to do parallel computing and concurrent programming quickly and efficiently.

What this book covers

Chapter 1, Introduction to C++ Concepts and Features, introduces the C++ programming language along with its large number of features. It focuses on concepts such as the structure of the program, execution flow, and Windows OS runtime objects. The structural and object-oriented approach is also covered in detail.

Chapter 2, The Concepts of Process and Thread, covers process and thread objects in detail. The idea behind the process model and implementation of Windows processes are covered thoroughly. We go through Interprocess Communication along with classical IPC problems. We then go through the overview of thread implementation in the user space as well as in the kernel.

Chapter 3, Managing Threads, gives you the logic behind the process and the thread. We cover Windows OS features such as permissive and preemptive multitasking. We also cover thread synchronization and synchronization objects and techniques in detail.

Chapter 4, Message Passing, focuses on message-passing techniques, window handlers, along with message queues and pipe communication.

Chapter 5, Thread Synchronization and Concurrent Operations, talks about parallelism, priority, the dispatcher object, and scheduling techniques. We also explain synchronization objects such as mutex, semaphore, event, and critical section.

Chapter 6, Threads in the .NET Framework, gives an overview of the C++/CLI .NET thread object. We briefly cover the Managed approach, .NET synchronization essentials, .NET thread safety, event-based asynchronous pattern, and the BackgroundWorker object along with some other topics.

Chapter 7, Understanding Concurrent Code Design, covers features such as performance factors, correctness, and liveness problems. In this chapter, users can find a better perspective of concurrency and parallel application design.

Chapter 8, Advanced Thread Management, focuses on a higher aspect of thread management. Abstractions of the thread pool, along with a custom dispatching object with deadlock resolution are covered in detail. Remote threading is given as a final example of advanced management.

Appendix covers the Installation of MySQL Connector C and WinDDK—Driver Development Kit. Other than that it contains the setting Visual Studio project for driver compilation as well as for OpenMP compilation. It covers the installation of DebugView application and shows the steps to use it.

What you need for this book

To execute the examples in the book, the following software will be required:

- ▶ Visual Studio 2013
 - <http://www.visualstudio.com/downloads/download-visual-studio-vs#d-express-windows-8>
- ▶ Windows Drivers Kit: WinDDK
 - <http://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx>
- ▶ MySQL Connector C
 - <http://dev.mysql.com/downloads/connector/c/>

Who this book is for

This book is intended primarily for intermediate and advanced users. Synchronization concepts are covered from the very beginning, thus making the book readable for all developers unassociated to any particular branch of expertise. The last two chapters will provide great knowledge to advanced users, providing an excellent overview on concepts such as concurrent design and advanced thread management.

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, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
class CLock
{
public:
    CLock(TCHAR* szMutexName);
    ~CLock();
```

```
private:
    HANDLE hMutex;
};

inline CLock::CLock(TCHAR* szMutexName)
{
    hMutex = CreateMutex(NULL, FALSE, szMutexName);
    WaitForSingleObject(hMutex, INFINITE);
}

inline CLock::~CLock()
{
    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}
```

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

```
class CLock
{
public:
    CLock(TCHAR* szMutexName);
    ~CLock();
private:
    HANDLE hMutex;
};


inline CLock::CLock(TCHAR* szMutexName)
{
    hMutex = CreateMutex(NULL, FALSE, szMutexName);
    WaitForSingleObject(hMutex, INFINITE);
}


inline CLock::~CLock()
{
    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}
```

It is very important is for reader to pay attention when copying and pasting code directly from the book. Due to book dimensions, some code lines could not be put into a single line. We tried to overcome this issue but simply it wasn't possible in certain situations. We strongly recommend you to check such cases before start compiling the examples. Especially in cases of double quoted strings, if in some accidental case the string is split as it should not be.

Also very important is a step in the "How to do it" sections when we say, for example, "Add existing header file CQueue.h previously implemented in Chapter 1, Introduction to C++ Concepts and Features". What we mean by this is that you should navigate to the folder where CQueue.h file resides using Windows Explorer and you should then copy the file along with all its dependences—in this case CList.h—to the example project working folder, before you add it to the project, using the "Add Existing Header File" option. By following this procedure you will be able to properly compile and run the example code.

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: "Open **Solution Explorer** and right-click on **Header files**."

 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

Introduction to C++ Concepts and Features

In this chapter, we will cover the following topics:

- ▶ Creating a C++ project
- ▶ Program structure, execution flow, and runtime objects
- ▶ The structural programming approach
- ▶ Understanding the object-oriented programming approach
- ▶ Explaining inheritance, overloading, and overriding
- ▶ Understanding polymorphism
- ▶ Event handlers and Message Passing Interface
- ▶ Linked list, queue, and stack examples

Introduction

The central concept in all operating systems is the process or abstraction of an executing program. Most modern operating systems can do more than one operation at a time. For example, if the user is typing in a Word document, the computer can print the document, read from the hard disc drive buffer, play music, and so on. In multitasking operating systems, the central processing unit (from now on, CPU) makes a fast switch between programs, executing each program for only a few milliseconds.

While on a single processor system, strictly speaking, only one process can be executed by the processor in one unit of time—the operating system creates an impression that more than one process is running simultaneously by switching them extremely quickly. This is called *pseudoparallelism* in contrast to hardware-supported native parallelism in multiprocessor systems.

Multithreading is a very important concept for today's modern operating systems. It allows more than one thread of execution, which is very important for parallel tasks as well as for user-friendly application behavior.

In traditional operating systems, every process has its own address space and one thread of execution—often called the *primary thread* or *main thread*. Usually, more than one thread in the same address space of a single process is needed, executing in a quasi-parallel context and acting like separated processes, with the exception of a common address space.



Pseudoparallelism is an OS feature executed over a single-processor environment. The quasi-parallel address space concept is a Windows OS feature. When executed over a multiprocessor system, Windows provides every process with a so-called virtual address space, which is much larger than real physical address space, and hence it is a quasi-parallel context.

A very important concept in all operating systems is the *thread*. A thread object maintains a program counter that takes care of what instruction has to be executed the next time the thread gets the processor time. It also has registers, which save the current values of variables that the thread is manipulating, stack where data related to function calls and parameters are stored, and so on. Even though threads are executed in the context of the process, there is a big difference between processes and threads. The process is greedy, wanting all resources for itself; threads are more "friendly", they cooperate and communicate with each other and share resources such as processor time, memory, and shared variables.

Creating a C++ project

All our examples will require Visual Studio IDE. We will now show you how to properly set up an IDE. We'll also point out specific settings that will impact multithreaded applications.

Getting ready

Make sure **Visual Studio (VS)** is up and running.

How to do it...

Run Visual Studio and select a new project on the **Start** page. A new window with multiple options will appear. Under **Templates** on the left side, select **C++** and expand the C++ node. You'll see the **CLR**, **General**, **Test**, and **Win32** options. Then, perform the following steps:

1. Select **Win32**. You'll see two options in the middle row: **Win32 console application** and **Win32 project**.

The **Win32 project** option is used for applications that have a **Graphical User Interface (GUI)** other than the console. The console can still be used, but with additional option settings in project properties. For now, we will use the **Win32 console application** option.

2. Select **Win32 console application** and enter a unique name for your project. We will name our first Win32 console application project `TestProject`. Under **Location**, select the folder where the project files will be stored. VS will create a folder for you, and it will remember the folder that you have just entered under **Location** as a default folder when creating future projects.

Now, you'll see the Win32 application wizard window. You can select **Finish** directly and VS will create all the necessary files for you, or you can select **Next** and choose between options where VS creates an empty project. You will have to create the source and header files yourself, unless VS generates all the necessary files for you. For now, select **Finish** and you are ready to create an application.

3. In **Solution Explorer**, you'll see the `stdafx.h` and `targetver.h` header files. The `stdafx.cpp` and `Test Project.cpp` source files are also created. The `stdafx.h` and `stdafx.cpp` files are a part of the precompiled header which is used for an IntelliSense engine that mimics the command-line compiler by using a **Translation Unit (TU)** model to service IntelliSense requests. A typical translation unit consists of a single source file and the several header files included in that source file. Other headers can be referenced so that they are included too. The IntelliSense engine exists to provide information to the users, such as what a particular type is, what the prototype of a function (and its overloads) is, or what variables are available in the current scope, beginning with a particular substring. More information is available at the MSDN reference (<http://msdn.microsoft.com>).
4. The `Test Project.cpp` file appears in the central window, and this is where you'll write your code. Later, we will create and use more files in more complex projects, but this should do for now.

How it works...

Every program must have its main routine, which is called `main`. When you want to run your program, the operating system starts it by calling `main`. This is the starting point of execution for all C++ programs. If you write code that adheres to the Unicode programming model, you can use the wide-character version of `main` called `wmain`. You can also use `_tmain`, which is defined in `TCHAR.h`. The `_tmain` function will resolve to `main`, unless `_UNICODE` is defined, in which case `_tmain` will resolve to `wmain`.

Above the **Test Project** window, you'll see various options; among them is a combobox with the **Win32** option selected, which is called **Solution platform**. If you want to create a 32-bit executable, leave it as is. For 64-bit executables, expand the combobox, select **Configuration manager**, and select **New** under **Active solution platforms**. A new **x64** option will appear. Click on **OK**, and close the **Configuration manager** window.

One of the most important settings that you need to change when building 64-bit executables is under **Project properties**. Press **Alt + F7**, or select **Project properties** in **Solution explorer** after right-clicking on the `TestProject` project. The **Test Project Property Pages** window will appear. Expand **Configuration Properties** and **C/C++**, and select **Preprocessor**. Under **Preprocessor definitions**, you'll see certain values copied from the Win32 settings. You'll need to change `WIN32` to `_WIN64` in order to build a 64-bit executable. Leave the rest as is.

Whether you build 32-bit or 64-bit code, there is one more thing that you need to set properly, which is code generation. When creating C++ projects, you can select whether your application will depend on **Dynamic Link Libraries (DLL)** required for C++ runtime on the user PC. If you create an application that runs on a PC other than the PC where it is created, you'll need to think about this. On the PC where you use VS, the required C++ runtime is already installed, so this does not matter. However, on another PC, where, for example, C++ runtime isn't installed, this might cause issues. If you want to be sure that no dependency is required, you'll need to change the **Runtime Library** option to **Multi-threaded Debug (/MTd)** for the Debug mode, or change it to **Multi-threaded (/MT)** for the Release mode. The Debug or Release mode can be switched in the Solution Configurations combobox.

For our examples, you can leave everything as it is because 32-bit executables can be run on both 32-bit and 64-bit machines. The runtime library can also be left as default because the application will run fine on the PC where it is built as the required C++ Redistributable Package framework is already installed.

Program structure, execution flow, and runtime objects

A programming paradigm is a fundamental style of computer programming. There are four main paradigms: imperative, declarative, functional (or structural), and object-oriented. The C++ language is certainly the most popular object-oriented language today. It is a very powerful, flexible, and comfortable programming language. Programmers adopted it very gladly and quickly, just like its predecessor, C. The key to such success lies in the fact that it was made by a single programmer, adjusting it to his needs.

Unfortunately, C++ isn't an easy language at all. Sometimes, you can think that it is a limitless language that cannot be learned and understood entirely, but you don't need to worry about that. It is not important to know everything; it is important to use the parts that are required in specific situations correctly. Practice is the best teacher, so it's better to understand how to use as many of the features as needed.

In examples to come, we will use author Charles Simonyi's Hungarian notation. In his PhD in 1977, he used *Meta-Programming – A software production method* to make a standard for notation in programming, which says that the first letter of type or variable should represent the data type. For the example class that we want to call, a `Test` data type should be `CTest`, where the first letter says that `Test` is a class. This is good practice because a programmer who is not familiar with the `Test` data type will immediately know that `Test` is a class. The same standard stands for primitive types, such as `int` or `double`. For example, `iCount` stands for an integer variable `Count`, while `dValue` stands for a double variable `Value`. Using the given prefixes, it is easy to read code even if you are not so familiar with it.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our first program by performing the following steps and explaining its structure:

1. Create a new C++ console application named `TestDemo`.
2. Open `TestDemo.cpp`.
3. Add the following code:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Hello world" << endl;
    return 0;
}
```

How it works...

The structure of the C++ program varies due to different programming techniques. What most programs must have is the `#include` or preprocessor directives.

The `#include <iostream>` header tells the compiler to include the `iostream.h` header file where the available function prototypes reside. It also means that libraries with the functions' implementations should be built into executables. So, if we want to use some API or function, we need to include an appropriate header file, and maybe we will have to add an additional input library that contains the function/API implementation. One more important difference when including files is `<header>` versus `"header"`. The first (`<>`) targets solution-configured project paths, while the other (`" "`) targets folders relative to the C++ project.

The `using` command instructs the compiler to use the `std` namespace. Namespaces are packages with object declarations and function implementations. Namespaces have a very important usage. They are used to minimize ambiguity while including third-party libraries when the same function name is used in two different packages.

We need to implement a program entry point—the `main` function. As we said before, we can use `main` for an ANSI signature, `wmain` for a Unicode signature, or `_tmain` where the compiler will resolve its signature depending on the preprocessor definitions in the project property pages. For a console application, the `main` function can have the following four different prototypes:

- ▶ `int _tmain(int argc, TCHAR* argv[])`
- ▶ `void _tmain(int argc, TCHAR* argv[])`
- ▶ `int _tmain(void)`
- ▶ `void _tmain(void)`

The first prototype has two arguments, `argc` and `argv`. The first argument, `argc`, or the argument count, says how many arguments are present in the second argument, `argv`, or the argument values. The `argv` parameter is an array of strings, where each string represents one command-line argument. The first string in `argv` is always the current program name. The second prototype is the same as the first one, except the return type. This means that the `main` function may or may not return a value. This value is returned to the OS. The third prototype has no arguments and returns an integer value, while the fourth prototype neither has arguments nor returns a value. It is good practice to use the first format.

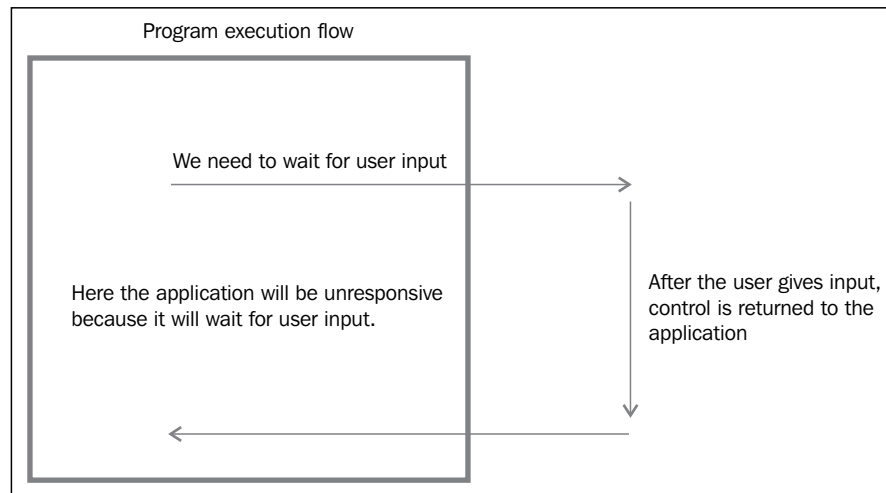
The next command uses the `cout` object. The `cout` object is the name of the standard output stream in C++, and the meaning of the entire statement is to insert a sequence of characters (in this case, the *Hello world* sequence of characters) into the standard output stream (usually corresponds to the screen).

The `cout` object is declared in the `iostream` standard file within the `std` namespace. This is why we need to include the specific file and declare that we will use this specific namespace earlier in our code.

In our usual selection of the `(int _tmain(int, _TCHAR*))` prototype, `_tmain` returns an integer. We must specify some `int` value after the return command, in this case `0`. When returning a value to the operating system, `0` usually means success, but this is operating system dependent.

This simple program is very easy to create. We use this simple example to demonstrate the basic program structure and usage of the `main` routine as the entry point for every C++ program.

Programs with one thread are executed sequentially line-by-line. This is why our program is not user friendly if we place all code into one thread.



After the user gives the input, control is returned to the application. Only now can the application continue the execution. In order to overcome such an issue, we can create concurrent threads that will handle the user input. In this way, the application does not stall and is responsive all the time. After a thread handles its task, it can signal that a user has performed the requested operation to the application.

There's more...

Every time we have an operation that needs to be executed separately from the main execution flow, we have to think about a separate thread. The simplest example is when we have some calculations, and we want to have a progress bar where the calculation progress will be shown. If the same thread was responsible for calculation as well as to update the progress bar, probably it wouldn't work. This occurs because, if both work and a UI update are performed from a single thread, such threads can't interact with OS painting adequately; so almost always, the UI thread is separate from the working threads.

Let's review the following example. Assume that we have created a function where we will calculate something, for example, sines or cosines of some angle, and we want to display progress in every step of the calculation:

```

void CalculateSomething(int iCount)
{
    int iCounter = 0;
    while (iCounter++ < iCount)
    {
        //make calculation
        //update progress bar
    }
}
  
```

As commands are executed one after another inside each iteration of the while loop, the operating system doesn't have the required time to properly update the user interface (in this case, the progress bar), so we would see an empty progress bar. After the function returns, a fully filled progress bar appears. The solution for this is to create a progress bar on the main thread. A separate thread should execute the `CalculateSomething` function; in each step of iteration, it should somehow signal the main thread to update the progress bar step by step. As we said before, threads are switched extremely fast on the CPU, so we can get the impression that the progress bar is updated at the same time at the calculation is performed.

To conclude, each time we have to make a parallel task, wait for some kind of user input, or wait for an external dependency such as some response from a remote server, we will create a separate thread so that our program won't hang and be unresponsive.

In our future examples, we will discuss static and dynamic libraries, so let's say a few words about both. A static library (`*.lib`) is usually some code placed in separate files and already compiled for future use. We will add it to the project when we want to use some of its features. When we wrote `#include <iostream>` earlier, we instructed the compiler to include a static library where implementations of input-output stream functions reside. A static library is built into the executable in compile time before we actually run the program. A dynamic library (`*.dll`) is similar to a static library, but the difference is that it is not resolved during compilation; it is linked later when we start the program, in another words—at runtime. Dynamic libraries are very useful when we have functions that a lot of programs will use. So, we don't need to include these functions into every program; we will simply link every program at runtime with one dynamic library. A good example is `User32.dll`, where the Windows OS placed a majority of GUI functions. So, if we create two programs where both have a window (GUI form), we do not need to include `CreateWindow` in both programs. We will simply link `User32.dll` at runtime, and the `CreateWindow` API will be available.

The structural programming approach

As we said before, there are four programming paradigms. Both structural and object-oriented paradigms can be used when making programs in C++. Even though C++ is an object-oriented language, you can make programs with a structural approach. Usually, a program contains one or more functions because every program must have a main routine; so, we can place all our code into the `main` function. However, in that way, any large program will be very hard to read. Then, we will have to split the code in certain program units that we call functions. Let's give an example of a program that needs to calculate the sum of two complex numbers in the following sections.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Perform the following steps:

1. Create a new default console application. Name it `ComplexTest`.
2. Open the `ComplexTest.cpp` file and insert the following code:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

void ComplexAdd( double dReal1, double dImg1, double dReal2,
double dImg2, double& dReal, double& dImg )
{
    dReal = dReal1 + dReal2;
    dImg = dImg1 + dImg2;
}

double Rand(double dMin, double dMax)
{
    double dVal = (double)rand() / RAND_MAX;
    return dMin + dVal * (dMax - dMin);
}

int _tmain(int argc, TCHAR* argv[])
{
    double dReal1 = Rand( -10, 10 );
    double dImg1 = Rand( -10, 10 );
    double dReal2 = Rand( -10, 10 );
    double dImg2 = Rand( -10, 10 );
    double dReal = 0;
    double dImg = 0;
    ComplexAdd(dReal1, dImg1, dReal2, dImg2, dReal, dImg);
    cout << dReal<< "+" <<dImg << "i" << endl;
    return 0;
}
```

How it works...

We created the `ComplexAdd` function that has six parameters or three complex numbers. The first two parameters are the real and imaginary parts of the first complex number, parameters three and four are the real and imaginary parts of the second complex number, and the fifth and sixth parameters are the real and imaginary parts of the complex number that will represent the result or the sum of the first two complex numbers. Note that the fifth and sixth parameters are passed by reference. We also created a `Rand` function that will return a random real number from a range between `dMin` and `dMax`.

As you can see, we have achieved what we wanted; however, you'll agree that the given code is not very readable and the `ComplexAdd` function has many parameters. Also, we have six variables in the `main` function, so the conclusion is that this kind of approach is not very "friendly". Let's review a slightly different approach by using structures, as shown:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

struct SComplex
{
    double dReal;
    double dImg;
};

SComplex ComplexAdd(SComplex c1, SComplex c2)
{
    SComplex c;
    c.dReal = c1.dReal + c2.dReal;
    c.dImg = c1.dImg + c2.dImg;
    return c;
}

double Rand(double dMin, double dMax)
{
    double dVal = (double)rand() / RAND_MAX;
    return dMin + dVal * (dMax - dMin);
}

int _tmain(int argc, TCHAR* argv[])
{
    SComplex c1;
    c1.dReal = Rand( -10, 10 );
    c1.dImg = Rand( -10, 10 );
```

```

SComplex c2;
c2.dReal = Rand( -10, 10 );
c2.dImg = Rand( -10, 10 );

SComplex c = ComplexAdd( c1, c2 );

cout<<c.dReal<< "+" <<c.dImg << "i" << endl;
return 0;
}

```

There's more...

As you can see, by creating a new type (in this case, structure), `SComplex`, which represents complex numbers, our code is more readable and more meaningful than the one shown in the previous example. The conclusion is that if we can somehow create objects that can represent an abstraction of tasks that we need to perform, our code, and the program itself, will make much more sense, and it will be easier for us to write and understand the code.

Understanding the object-oriented programming approach

Object-oriented programming (OOP) is a new approach to create software as a model for the real world. This is a particular way of designing a program. OOP includes several key concepts, such as classes, objects, inheritance, and polymorphism.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Let's review the previous example with the OOP approach by performing the following steps:

1. Create a new console application. Name it `ComplexTestOO`.
2. Open the `ComplexTestOO.cpp` file and insert the following code:

```

#include "stdafx.h"
#include <iostream>

using namespace std;

double Rand(double dMin, double dMax)
{

```



```
        double dVal = (double)rand() / RAND_MAX;
        return dMin + dVal * (dMax - dMin);
    }

class CComplex
{
public:
    CComplex( )
    {
        dReal = Rand( -10, 10 );
        dImg = Rand( -10, 10 );
    }
    CComplex(double dReal, double dImg)
    {
        this->dReal = dReal;
        this->dImg = dImg;
    }
    friend CComplex operator+ (const CComplex& c1, const CComplex&
c2);
    friend ostream& operator<< (ostream& os, const CComplex& c);
private:
    double dReal;
    double dImg;
};

CComplex operator+ (const CComplex& c1, const CComplex& c2)
{
    CComplex c(c1.dReal + c2.dReal, c1.dImg + c2.dImg);
    return c;
}
ostream& operator<< ( ostream& os, const CComplex& c )
{
    return os<<c.dReal<< "+" <<c.dImg << "i";
}

int _tmain(int argc, TCHAR* argv[])
{
    CComplex c1;
    CComplex c2(-2.3, 0.9);
    CComplex c = c1 + c2;
    cout << c << endl;

    return 0;
}
```

How it works...

If you look at the `main` function, you'll notice that it doesn't differ from the program where you use integers, for example. Even though an integer is a primitive type and a complex is not, a programmer's work is made easier by adding a class and properly setting its methods.

Here we have defined a new type of class—`CComplex`. The defined class has its own attributes and methods. It can also have friend methods that can access its private members. We have left the `Rand` function as it is, but we tried to make the class that represents a complex number as similar to the abstraction of a complex number as possible. We created the `dReal` and `dImag` attributes that describe the real and imaginary parts of a complex number. We created the `operator+` method so that the `+` (plus) sign has meaning for the compiler. Now, the code is more readable and much more comfortable to use because it is easy to understand how to use it. We created two constructors: the default constructor that uses random real numbers from -10 to 10, and another constructor that enables the user to set the real and imaginary parts of a complex number directly.

When you want to overload (this will be explained later in detail) a certain method, you can choose between two approaches. The first approach is to set the method inside a class, and this method will change the state (or values) of the calling object. The second approach is to make the method independent of the class, the so-called *outer*, but it has to be declared as *friend* so that it can access the private and protected members of the objects.

There's more...

The following code tells the compiler that the `operator+` function which expects two complex numbers will be defined somewhere in the code, and it has to be able to access the private and protected members of the `CComplex` class:

```
friend CComplex operator+ (const CComplex& c1, const
CComplex& c2);
```

This makes the `main` function the simplest possible. Again, we use user-defined types such as primitive types, which is very good practice because even programmers not so familiar with the code will quickly understand what the code means.

Explaining inheritance, overloading, and overriding

Inheritance is a very important characteristic of OOP. It is the relation between two (or more) classes: if class B is some kind of class A, then the objects of class B have the same properties as the objects of class A. In addition to that, class B can implement new methods and properties, and thus supersede the base class A.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Let's review this example by performing the following steps:

1. Create a new default console application. Name it `InheritanceTest`.
2. Open the `InheritanceTest.cpp` file and insert the following code:

```
#include "stdafx.h"
#include <iostream>

using namespace std;

class CPerson
{
public:
    CPerson(int iAge, char* sName)
    {
        this->iAge = iAge;
        strcpy_s(this->sName, 32, sName);
    }
    virtual char* WhoAmI()
    {
        return "I am a person";
    }
private:
    int iAge;
    char sName[32];
};

class CWorker : public CPerson
{
public:
    CWorker(int iAge, char* sName, char* sEmploymentStatus)
        : CPerson(iAge, sName)
    {
        strcpy_s(this->sEmploymentStatus, 32, sEmploymentStatus);
    }
}
```

```
        virtual char* WhoAmI()
        {
            return "I am a worker";
        }
private:
    char sEmploymentStatus[32];
};

class CStudent : public CPerson
{
public:
    CStudent(int iAge, char* sName, char* sStudentIdentityCard)
        : CPerson(iAge, sName)
    {
        strcpy_s(this->sStudentIdentityCard, 32,
sStudentIdentityCard);
    }
    virtual char* WhoAmI()
    {
        return "I am a student";
    }
private:
    char sStudentIdentityCard[32];
};

int _tmain(int argc, TCHAR* argv[])
{
    CPerson cPerson(10, "John");
    cout << cPerson.WhoAmI() << endl;

    CWorker cWorker(35, "Mary", "On wacation");
    cout << cWorker.WhoAmI() << endl;

    CStudent cStudent(22, "Sandra", "Phisician");
    cout << cStudent.WhoAmI() << endl;

    return 0;
}
```

How it works...

We created a new `CPerson` data type that represents one human. It has `iAge` and `sName` as attributes that describe some person. Let's assume that we need some worker or student as another data type. OOP provides a great mechanism, inheritance, which can be used to achieve this. As a worker is still a person, but with some additional attributes, we will extend `CPerson` with `CWorker` using the following line of code:

```
class CWorker : public CPerson
```

So, `CWorker` is a derived class of `CPerson`. It has all the attributes and methods from the base class, `CPerson`, and an additional `sEmploymentStatus` attribute that is important to a worker only. Further, let's say we need a Student data type. Since a student is a person who has some additional attributes besides age and name, we will extend a person again by using the following line of code:

```
class CStudent : public CPerson
```

What is important for us to know is that when you declare an object, its constructor is being called. When you declare an object of a derived class, the constructor from the base class is called first, and the constructor from the derived class is called later. The following is the code sample:

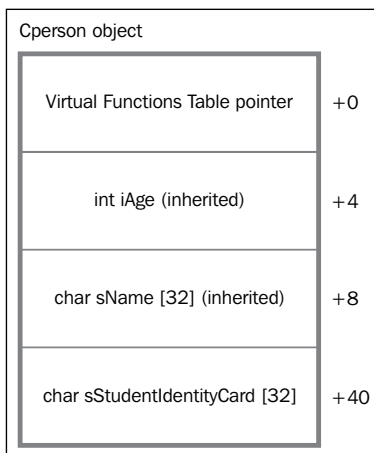
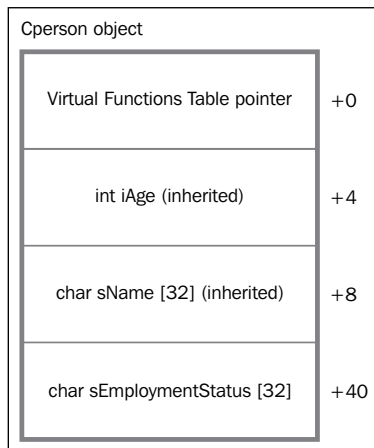
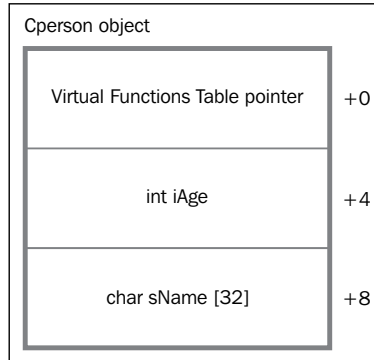
```
CWorker( int iAge, char* sName, char* sEmploymentStatus )
: CPerson( iAge, sName )
{
    strcpy_s( this->sEmploymentStatus, 32, sEmploymentStatus );
}
```

Take a look at the `CWorker` constructor prototype; after the parameter list, you'll notice a : (colon) sign after which the base class constructor is called, as shown in the following code, with the needed parameters `iAge` and `sName`, for `CPerson` to be created:

```
CPerson(iAge, sName)
```

The situation with destructors is the opposite. The destructor from the derived class is called before, from the base class.

Graphically, the `CPerson`, `CWorker`, and `CStudent` objects look like the following diagrams, respectively:



The meaning of the operators can be defined even for user-defined types, such as `CComplex` in the previous example. It is very handy, as you saw, because it is more meaningful to write `c = c1 + c2`, where `c`, `c1`, and `c2` are complex numbers, than to write `c = ComplexAdd(c1, c2)`.

The user must implement the operator functions or overload them in order for the compiler to know how to deal with user-specific types. The compiler knows what to do with primitive types, such as adding two integers, but imagine two matrices `m1` and `m2` and the expression `matrix m = m1 + m2`. Well, if we do not define the `CMatrix operator + (const CMatrix& m1, const CMatrix& m2)` function, the compiler won't know what to do because it doesn't know how to add matrices.

Overriding a method is a feature that allows a derived class to provide a specific implementation of a method that is already provided in the base class. Take a look at the `WhoAmI` method from the previous example. Its output will be as follows:

```
I am a person
I am a worker
I am a student
```

Even though a method from each class has the same name, it is a different method and has a different functionality. We can say that the `WhoAmI` method is overridden in the `CPerson` derived class.

There's more...

Overriding is an excellent feature in OOP and C++, but polymorphism is even better. Let's review the following example.

Understanding polymorphism

Polymorphism is a feature where an object executes an operation in the way that it is implemented in the derived class that it belongs to, even if we access it through a pointer or a reference of the base class. Polymorphism is a feature that allows an object to take many forms.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Perform the following steps:

1. Create a new default console application. Name it `PolymorphismTest`.

2. Open the `PolymorphismTest.cpp` file and insert the following code:

```
#include "stdafx.h"
#include <iostream>

#define M_PI 3.14159265358979323846

using namespace std;

class CFigure
{
public:
    virtual char* FigureType() = 0;
    virtual double Circumference() = 0;
    virtual double Area() = 0;
    virtual ~CFigure() { }
};

class CTriangle : public CFigure
{
public:
    CTriangle()
    {
        a = b = c = 0;
    }
    CTriangle(double a, double b, double c) : a(a), b(b), c(c) { }
    virtual char* FigureType()
    {
        return "Triangle";
    }
    virtual double Circumference()
    {
        return a + b + c;
    }
    virtual double Area()
    {
        double S = Circumference() / 2;
        return sqrt(S * (S - a) * (S - b) * (S - c));
    }
private:
    double a, b, c;
};
```



```
class CSquare : public CFigure
{
public:
    CSquare()
    {
        a = b = 0;
    }
    CSquare(double a, double b) : a(a), b(b) { }
    virtual char* FigureType()

    {
        return "Square";
    }
    virtual double Circumference()
    {
        return 2 * a + 2 * b;
    }
    virtual double Area()
    {
        return a * b;
    }
private:
    double a, b;
};

class CCircle : public CFigure
{
public:
    CCircle()
    {
        r = 0;
    }
    CCircle(double r) : r(r) { }
    virtual char* FigureType()
    {
        return "Circle";
    }
    virtual double Circumference()
    {
        return 2 * r * M_PI;
    }
    virtual double Area()
    {
```

```
        return r * r * M_PI;
    }
private:
    double r;
};

int _tmain(int argc, _TCHAR* argv[])
{
    CFigure* figures[3];

    figures[0] = new CTriangle(2.1, 3.2, 4.3);
    figures[1] = new CSquare(5.4, 6.5);
    figures[2] = new CCircle(8.8);

    for (int i = 0; i < 3; i++)
    {
        cout << "Figure type:\t" << figures[i]->FigureType()
              << "\nCircumference:\t" <<
              figures[i]->Circumference()
              << "\nArea:\t\t" << figures[i]->Area()
              << endl << endl;
    }

    return 0;
}
```

How it works...

First, we created a new type, `CFigure`. We would like to create some real figures, such as triangles, squares, or circles, and methods that will calculate their circumference and area. You'll agree that we can't calculate these features for the figure itself because we don't know what type of real figure it is. This is why we created `CFigure` as an abstract class. An abstract class is a class where at least one of its virtual methods is declared with no implementation and has `= 0` after its prototype. Functions declared in such a manner are called pure virtual functions. An abstract class can't have objects, but inherited classes can. You can instantiate pointers and references of abstract classes. We extend the `CTriangle`, `CSquare`, and `CCircle` classes that represent triangles, squares, and circles, respectively. In these derived classes, we implement the `FigureType`, `Circumference`, and `Area` methods because we want to have the ability to instantiate such types of objects.

Each of these three classes has the same method names but different implementations, similar to overriding but different in meaning. How? Well, if you look at the `main` function, you'll notice that we declared an array of three pointers of the `CFigure` type. As a pointer or reference to a base class can always point to any of its derived classes, it is possible to create an object of type `CTriangle` and set the `CFigure` pointer to point at it, using the following code:

```
figures[ 0 ] = new CTriangle( 2.1, 3.2, 4.3 );
```

In the same way, we set other figures using the following code:

```
figures[ 1 ] = new CSquare( 5.4, 6.5 );
figures[ 2 ] = new CCircle( 8.8 );
```

Now, refer to the following code:

```
for ( int i = 0; i < 3; i++ )
{
    cout << "Figure type:\t" << figures[ i ]->FigureType( )
    << "\nCircumference:\t" << figures[ i ]->Circumference( )
    << "\nArea:\t\t" << figures[ i ]->Area( ) << endl<<endl;
}
```

The compiler will use a C++ feature called `dynamic binding` to resolve the type of object to which the figure points. It'll then call an appropriate virtual method. Dynamic binding is possible only if the method is declared virtually and if you access it via a pointer or reference.

Let's go back to the example with persons. We had to declare objects of type `CPerson`, `CWorker`, and `CStudent`. All three are of different types, and you can call the `WhoAmI` method, for example, by using the following code:

```
cPerson.WhoAmI( )
```

The compiler knows that you are referring to the `cPerson` object, which is of a `CPerson` type, and its method is `WhoAmI`. On the other hand, in the example with figures, the compiler doesn't know what type of object the figure pointer will point to during compilation. This will be resolved during the runtime. This is why it is called `dynamic binding`.

Event handlers and Message Passing Interface

Many programs respond to some events, such as when a user presses a button or inputs some text. A very important mechanism is event handling or the ability of a program to react to user action. If we want to handle when a user presses a button, we need to create some kind of a listener that will listen to button events, in this case a press action.

An event handler is a function that the operating system calls each time the control sends some kind of message, such as **I have been pressed** in the case of buttons or **I received a character** in the case of a textbox.

Event handlers are very important. Timers are events that trigger after a certain portion of time has elapsed. When you press a key on the keyboard, the operating system raises a "key is pressed" event, and so on.

Event handlers for windows are very important for us. The majority of applications have windows or forms. Every window needs its event handler, which will be called each time some event occurs in its space. For example, if you create a window with a few buttons and textboxes, you must have a window procedure associated with that window in order to handle such events.

The Windows operating system provides such a mechanism in the form of a window procedure, usually called `WndProc` (but you can name it as you like). This procedure is called by the operating system each time some event specific to that window occurs. In the following example, we will create our first Windows application, where we will have a window and explain the usage of a window procedure.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Perform the following steps:

1. Create a new empty Win32 C++ project. Name it `GUIProject`, and then click on **OK**. Now, right-click on the source files and navigate to **Add | New item in Solution Explorer**. Name it `main`, and then click on **Add**.
2. Now, let's create the code. First, we will add a header that we need, as shown:

```
#include <windows.h>
```

The `windows.h` header is needed for the majority of APIs that are responsible for visual features such as windows, controls, enumerations, and styles. Before we create an application entry point, we have to declare a window procedure prototype so that we can use it in the window structure, as shown in the following code:

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

We will implement `WndProc` later but for now the declaration is enough. Now, we need an application entry point. Unlike a console application, the `main` function in Win32 applications has a slightly different prototype, as shown in the following code:

```
int WINAPI WinMain(HINSTANCE hThis, HINSTANCE hPrev, LPSTR szCmdLine,
int iCmdShow)
```

After the return type (`int`), you will notice the `WINAPI` macro. It represents a calling convention.

`WINAPI` or `stdcall` means that a call stack will clean the callee. `WinMain` is the name and it must have four parameters in a given order. The first parameter, `hThis`, is a handle to the current instance of the application. The second parameter, `hPrev`, is a handle to the previous instance of the application. If you refer to the MSDN documentation (<http://msdn.microsoft.com/en-us/library/windows/desktop/ms633559%28v=vs.85%29.aspx>), you'll notice that the `hPrev` parameter is always `NULL`, so I guess that this has been left for compatibility with previous versions of Windows operating systems, as it can't have values in current versions. The third parameter is `szCmdLine` or the command line for the application, excluding the program name. The final parameter controls how the window is to be shown.

It can have one or more values combining bit values with the OR (`|`) operator (more information available on MSDN).

We will then use the `UNREFERENCED_PARAMETER` macro to tell the compiler that we will not use certain parameters so that it can do some additional optimization. Use the following code:

```
UNREFERENCED_PARAMETER( hPrev );
UNREFERENCED_PARAMETER( szCmdLine );
```

Next, we need to instantiate the `WNDCLASSEX` window structure. This is an object where details of future windows, such as stack size, handle to current application instance, window style, window color, icon, and mouse pointer, will be stored. The code to instantiate the `WNDCLASSEX` window structure is as follows:

```
WNDCLASSEX wndEx = { 0 };
```

The number of extra bytes to allocate after instantiating the window class structure can be defined by using the following code:

```
wndEx.cbClsExtra = 0;
```

The size, in bytes, of this structure can be defined by using the following code:

```
wndEx.cbSize = sizeof( wndEx );
```

The number of extra bytes to allocate after instantiating the window instance can be defined by using the following code:

```
wndEx.cbWndExtra = 0;
```

A handle to the class background brush can be defined by using the following code:

```
wndEx.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
```

A handle to the class cursor can be defined by using the following code:

```
wndEx.hCursor = LoadCursor( NULL, IDC_ARROW );
```

Handles to the class icon can be defined by using the following code:

```
wndEx.hIcon = LoadIcon( NULL, IDI_APPLICATION );  
wndEx.hIconSm = LoadIcon( NULL, IDI_APPLICATION );
```

A handle to the instance that contains the window procedure for the class can be defined by using the following code:

```
wndEx.hInstance = hThis;
```

A pointer to the window procedure can be defined by using the following code:

```
wndEx.lpfnWndProc = WndProc;
```

A pointer to a null-terminated string or an atom can be defined by using the following code:

```
wndEx.lpszClassName = TEXT("GUIProject");
```

A pointer to a null-terminated character string that specifies the resource name of the class menu, as the name appears in the resource file, can be defined using the following code:

```
wndEx.lpszMenuName = NULL;
```

The class style(s) can be defined using the following code:

```
wndEx.style = CS_HREDRAW | CS_VREDRAW;
```

The following code registers a window class for subsequent use in calls to the `CreateWindow` or `CreateWindowEx` function:

```
if ( !RegisterClassEx( &wndEx ) )  
{  
    return -1;  
}
```

The `CreateWindow` API creates an overlapped, pop-up, or child window. It specifies the window class, window title, window style, and (optionally) the initial position and size of the window. The function also specifies the window's parent or owner, if any, and the window's menu, as shown in the following code:

```
HWND hWnd = CreateWindow( wndEx.lpszClassName, TEXT("GUI Project"),
    WS_OVERLAPPEDWINDOW, 200, 200, 400, 300, HWND_DESKTOP, NULL, hThis, 0
);

if ( !hWnd )
{
    return -1;
}
```

The `UpdateWindow` function updates the client area of the specified window by sending a `WM_PAINT` message to the window if the window's update region is not empty. The function sends a `WM_PAINT` message directly to the window procedure of the specified window bypassing the application queue. The usage is shown in the following code:

```
UpdateWindow( hWnd );
```

The following code sets the specified window's show state:

```
ShowWindow( hWnd, iCmdShow );
```

Now, we need the instance of the `MSG` structure that represents the window message.

```
MSG msg = { 0 };
```

Here, we enter the message loop. Windows-based applications are event driven. They do not make explicit function calls (such as C runtime library calls) to obtain input. Instead, they wait for the system to pass input to them. The system passes all input for an application to the various windows in the application. Each window has a function, called a window procedure, which the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. The `GetMessage` API retrieves a message from the calling thread's message queue, as shown in the following code:

```
while ( GetMessage( &msg, NULL, NULL, NULL ) )
{
    // Translates virtual-key messages into character messages.
    TranslateMessage(&msg);
    // Dispatches a message to a window procedure.
    DispatchMessage(&msg);
}
```

When we close the application or send some command that signals it to exit, the system releases an application message queue. This means that our application will have no more messages, and it will end the `while` loop. The `DestroyWindow` API destroys the specified window. The function sends the `WM_DESTROY` and `WM_NCDESTROY` messages to the window to deactivate it and removes the keyboard focus from it. The function also destroys the window's menu, flushes the thread message queue, destroys timers, removes clipboard ownership, and breaks the clipboard viewer chain (if the window is at the top of the viewer chain).

```
DestroyWindow( hWnd );
```

The function also unregisters a window class, freeing the memory required for the class.

```
UnregisterClass( wndEx.lpszClassName, hThis );
```

The function returns a success exit code or a last message code from the application message queue, as shown in the following code:

```
return (int) msg.wParam;
```

Now, we need to implement a window procedure or an application main event handler. For our first example, we will create a simple `WndProc` with only the functionality of handling mandatory events. The window procedure returns 64-bit long signed integer values. It has four parameters: the `hWnd` structure that represents the window identifier, the `uMsg` unsigned integer that represents a window message code, the `wParam` unsigned 64-bit long integer to pass application-defined data, and the `lParam` signed 64-bit long integer to pass application-defined data too.

```
LRESULT CALLBACK WndProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam )
{
```

Messages are handled by their message code. For example, we need to handle default messages (in this case, `WM_CLOSE`), which the system sends when the application is closing. Then, we will call the `PostQuitMessage` API to free system resources and to safely close the application, using the following code:

```
switch ( uMsg )
{
    case WM_CLOSE:
    {
        PostQuitMessage( 0 );
        break;
    }

    default:
    {
```


Call the default window procedure to provide default processing for any window message that an application does not process. This function ensures that every message is processed. The default window procedure can be used in the following manner:

```
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}
return 0;
}
```

Though this was a very simple example of a window application, it completely reflects the event-driven system features and the event-handling mechanism. In chapters to come, we will use event handling all the time, so it is very important for you to understand the basic process.

Linked list, queue, and stack examples

The following example demonstrates the OOP usage of a linear linked list that can contain any generic type *T*. The idea behind this example is to show inheritance usage as a model for the relation that "B is a kind of A".

A linear list is a structure of linearly arranged elements in which we know that the first element links to the second, the second element links to the third, and so on. Basic operations with a linked list are insert element (*PUT*) and obtain element (*GET*). A queue is a linear list where the elements are arranged in order so that what enters a queue first gets out first, in other words, *FIFO—First In First Out*. Therefore, when you get an element, it is obtained from the top of the list, and you put a new element at the bottom of the list. A stack is a linear list where elements are arranged such that they are obtained in the reverse order of how they were inserted. The element is placed at the top of the list, and it is obtained from the top of the list, or *LIFO—Last In First Out*.

A linear list is an arranged collection of elements. Each list has methods that can set element value, obtain an element from the list, or simply peek an element. A list can be used to store any type of objects. Yet, for specializations of list class, queue, and stack, it is defined exactly where elements must be inserted or obtained from. So, a list as a general object will be a base class.

You should be aware that a list designed in such a manner can be implemented as a static or dynamic structure. In other words, it can be implemented as an array of elements or as a structure of elements linked by pointers to the next (or previous or both) element. Implementation using pointers is mostly used in practice and will be shown here.

The list, in the following example, will be implemented as a collection of pointers that point to the original objects that are placed in the list using its methods. So, the original objects won't be copied to the list, which enables polymorphism for list elements. Semantics, as such, requires additional attention and careful design.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Perform the following steps:

1. Create a new empty C++ console application project. Name it `LinkedList`.
2. Add a new header file named `CList.h`. Copy and paste the following code into it:

```
#ifndef _LIST_
#define _LIST_

#include <Windows.h>

template <class T>
class CNode
{
public:
    CNode(T* tElement) : tElement(tElement), next(0) { }
    T* Element() const { return tElement; }
    CNode*& Next(){ return next; }
private:
    T* tElement;
    CNode* next;
};

template <class T>
class CList
{
public:
    CList() : dwCount(0), head(0){ }
    CList(T* tElement) : dwCount(1), head(new CNode<T>(tElement)){ }
    virtual ~CList(){ }
    void Append(CNode<T>*& node, T* tElement);
    void Insert(T* tElement);
    bool Remove(T* tElement);
    DWORD Count() const { return dwCount; }
    CNode<T>*& Head() { return head; }
    T* GetFirst(){ return head != NULL ? head->Element() : NULL; }
    T* GetLast();
};
```

```

        T* GetNext(T* tElement);
        T* Find(DWORD(*Function)(T* tParameter), DWORD dwValue);
protected:
        CList(const CList& list);
        CList& operator = (const CList& list);
private:
        CNode<T>* head;
        DWORD dwCount;
};

template <class T>
void CList<T>::Append(CNode<T>*& node, T* tElement)
{
    if (node == NULL)
    {
        dwCount++;
        node = new CNode<T>(tElement);

        return;
    }

    Append(node->Next(), tElement);
}

template <class T>
void CList<T>::Insert(T* tElement)
{
    dwCount++;
    if (head == NULL)
    {
        head = new CNode<T>(tElement);
        return;
    }

    CNode<T>* tmp = head;
    head = new CNode<T>(tElement);
    head->Next() = tmp;
}

template <class T>
bool CList<T>::Remove(T* tElement)
{
    if (head == NULL)
    {

```

```
        return NULL;
    }

    if (head->Element() == tElement)
    {
        CNode<T>* tmp = head;
        head = head->Next();

        delete tmp;
        dwCount--;

        return true;
    }

    CNode<T>* tmp = head;
    CNode<T>* lst = head->Next();

    while (lst != NULL)
    {
        if (lst->Element() == tElement)
        {
            tmp->Next() = lst->Next();

            delete lst;
            dwCount--;

            return true;
        }

        lst = lst->Next();
        tmp = tmp->Next();
    }

    return false;
}

template <class T>
T* CList<T>::GetLast()
{
    if (head)
    {
        CNode<T>* tmp = head;
        while (tmp->Next())
        {
```

```

        tmp = tmp->Next();
    }
    return tmp->Element();
}

return NULL;
}

template <class T>
T* CList<T>::GetNext(T* tElement)
{
    if (head == NULL)
    {
        return NULL;
    }

    if (tElement == NULL)
    {
        return GetFirst();
    }

    if (head->Element() == tElement)
    {
        return head->Next() != NULL ? head->Next()->Element() :
NULL;
    }

    CNode<T>* lst = head->Next();
    while (lst != NULL)
    {
        if (lst->Element() == tElement)
        {
            return lst->Next() != NULL ? lst->Next()->Element() :
NULL;
        }

        lst = lst->Next();
    }

    return NULL;
}

template <class T>
T* CList<T>::Find(DWORD(*Function)(T* tParameter), DWORD dwValue)

```

```

{
    try
    {
        T* tElement = NULL;
        while (tElement = GetNext(tElement))
        {
            if (Function(tElement) == dwValue)
            {
                return tElement;
            }
        }
    }
    catch (...) { }

    return NULL;
}

#endif

```

3. The provided files contain definitions and implementations of the `CList` class. Now, we can create `CQueue` and `CStack` very easily. Under the header files, create a new header file named `CQueue.h` and insert the following code:

```

#ifndef __QUEUE__
#define __QUEUE__

#include "CList.h"

template<class T>
class CQueue : CList<T>
{
public:
    CQueue() : CList<T>() { }
    CQueue(T* tElement) : CList<T>(tElement) { }
    virtual ~CQueue() { }
    virtual void Enqueue(T* tElement)
    {
        Append(Head(), tElement);
    }
    virtual T* Dequeue()
    {
        T* tElement = GetFirst();
        Remove(tElement);
        return tElement;
    }
}

```

```
        virtual T* Peek()
        {
            return GetFirst();
        }
        CList<T>::Count;
protected:
        CQueue(const CQueue<T>& cQueue);
        CQueue<T>& operator = (const CQueue<T>& cQueue);
};

#endif
```

4. For CStack, we will do something similar. Under the header files, create a new header file named CStack.h and insert the following code:

```
#ifndef __STACK__
#define __STACK__
#include "CList.h"

template<class T>
class CStack : CList<T>
{
public:
    CStack() : CList<T>() { }
    CStack(T* tElement) : CList<T>(tElement) { }
    virtual ~CStack() { }
    virtual void Push(T* tElement)
    {
        Insert(tElement);
    }
    virtual T* Pop()
    {
        T* tElement = GetFirst();
        Remove(tElement);
        return tElement;
    }
    virtual T* Peek()
    {
        return GetFirst();
    }
    CList<T>::Count;
protected:
    CStack(const CStack<T>& cStack);
    CStack<T>& operator = (const CStack<T>& cStack);
};

#endif
```

5. Finally, let's implement the following code in `LinkedList.cpp`, our placeholder for the main routine:

```
#include <iostream>

using namespace std;

#include "CQueue.h"
#include "CStack.h"

int main()
{
    CQueue<int>* cQueue = new CQueue<int>();
    CStack<double>* cStack = new CStack<double>();

    for (int i = 0; i < 10; i++)
    {
        cQueue->Enqueue(new int(i));
        cStack->Push(new double(i / 10.0));
    }

    cout << "Queue - integer collection:" << endl;
    for (; cQueue->Count(); )
    {
        cout << *cQueue->Dequeue() << " ";
    }

    cout << endl << endl << "Stack - double collection:" << endl;
    for (; cStack->Count(); )
    {
        cout << *cStack->Pop() << " ";
    }

    delete cQueue;
    delete cStack;

    cout << endl << endl;
    return system("pause");
}
```


How it works...

Let's explain all the parts of the `CList` class first. The list consists of elements of type `CNode` for easier handling. The `CNode` class has two attributes: the `tElement` pointer to the user-defined element and the `next` pointer to the next list item. It implements two methods: `Element` and `Next`; the former returns a reference to the element address while the latter returns a reference to the next item address.

In literature constructor is referred as `ctor` and destructor as `dtor`. The `CList` default constructor is the public method that creates an empty list. The second constructor creates a list with a single starting element with a supplied pointer. As a list is a dynamic structure, the destructor is required. The `Append` method inserts an element as the last element in a collection. The `Insert` method inserts an element as the first element in a collection. The `Count` method returns the number of elements currently in the list. The `Head` method returns a reference to the list starting node.

The `GetFirst` method returns the first element from the list, or `NULL` if the list is empty. The `GetLast` method returns the last element from the list, or `NULL` if it's empty. The `GetNext` method returns the next list item, relative from the item for which the address is provided as the `T* tElement` parameter. If such an item is not found, the method returns `NULL`.

The `Find` method is most certainly an advanced feature. It is designed to use the undefined type `T`, and use its undefined `Function` method with the `tParameter` parameter. If we wanted to use a list of some objects, for example students, we would like to iterate them (using the `GetNext` method, for example) or find a specific student. What if it's possible for a future defined type to implement some method that will return `unsigned long (DWORD)` and can compare it with the `dwValue` parameter? For example, if we want to find a student based on their ID, we can use the following code:

```
#include <windows.h>
#include "CList.h"

class CStudent
{
public:
    CStudent(DWORD dwStudentId) : dwStudentId(dwStudentId) { }
    static DWORD GetStudentId(CStudent* student)
    {
        DWORD dwValue = student->GetId();
        return dwValue;
    }
    DWORD GetId() const
    {
        return dwStudentId;
    }
}
```

```
private:
    DWORD dwStudentId;
};

int main()
{
    CList<CStudent>* list = new CList<CStudent>();
    list->Insert(new CStudent(1));
    list->Insert(new CStudent(2));
    list->Insert(new CStudent(3));

    CStudent* s = list->Find(&CStudent::GetStudentId, 2);
    if (s != NULL)
    {
        // s FOUND
    }

    return 0;
}
```

If a list is used for primitive types, such as int, we can use the list as shown in the following code:

```
#include <windows.h>
#include "CList.h"

DWORD Predicate(int* iElement)
{
    return (DWORD)(*iElement);
}

int main()
{
    CList<int>* list = new CList<int>();
    list->Insert(new int(1));
    list->Insert(new int(2));
    list->Insert(new int(3));

    int* iElement = list->Find(Predicate, 2);
    if (iElement != NULL)
    {
        // iElement FOUND
    }

    return 0;
}
```

The protected methods are copy constructor (copy constructor) and `operator=` (operator equal). Why do we set these two methods as protected? The list implemented here, as we said, saves only pointers to objects that "live" outside the list itself. It would be dangerous to allow the user to copy the list intentionally or unintentionally and without caution, either using copy constructor or equal op for this reason. This is why we decided to protect copy constructor and disable the user from using it. In order to achieve this behavior, we need to declare, and later implement, both methods; else the compiler will generate them as `public`, by default, implementing them with copying pointers one by one, which is wrong. It is not enough to declare them as `private` either; in such a case, the same problem will occur for classes derived from the `CList` base class. They too need to declare copy constructor and equal op, else these methods will be `public`, generated from the compiler by default. When copy constructor exists in base class, the derived copy constructor will call it by default, except if the derived class calls it explicitly in another way. As we want to enable the `CList` copy constructor to be made available to the derived classes, it is declared as `protected`.

The `private` section of the `CList` class contains objects needed for the implementation of the linked list as a linear linked list. This means that every element points to the next, while the head node points to the first element.

The `CQueue` and `CStack` classes implement queue and stack, respectively. Notice how easy their implementation was after properly designing the `CList` base class (especially the `Enqueue`, `Dequeue`, `Push`, `Pop`, and `Peek` methods). Once the `CList` class is carefully designed, it is very simple to design and implement `CQueue` and `CStack`, or even some other classes later.

2

The Concepts of Process and Thread

In this chapter we will cover the following topics:

- ▶ Processes and threads
- ▶ Explaining the process model
- ▶ Implementation of processes
- ▶ IPC – Interprocess Communication
- ▶ Solving classical IPC problems
- ▶ Implementation of the thread model
- ▶ Thread usage
- ▶ Implementing threads in user space
- ▶ Implementing threads in the kernel

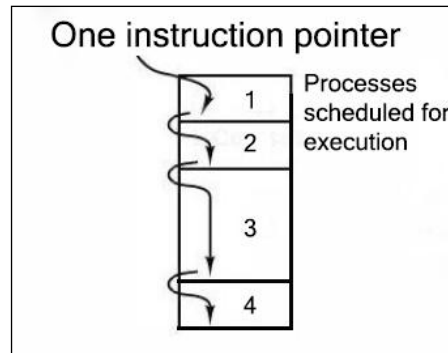
Introduction

Modern computers can do several things at the same time. Windows users are not fully aware of this, so let's give an example that will illustrate things more closely. When the PC system is booted, many processes that are hidden from the user are started. For example, a process that manages incoming e-mails, a process in charge of obtaining latest antivirus definitions, and so on. Explicit user processes may also be running: printing files, or burning CDs, all while the user is performing some other tasks such as surfing the Web. Activities such as these must be managed, and a multitasking system, which supports multiple processes, can be very handy here. In such a multitasking system, the CPU switches between processes quickly, running each process for a few milliseconds. Strictly speaking, at any instant of time, the CPU is running only one process by switching quickly among processes—giving the illusion of parallelism.

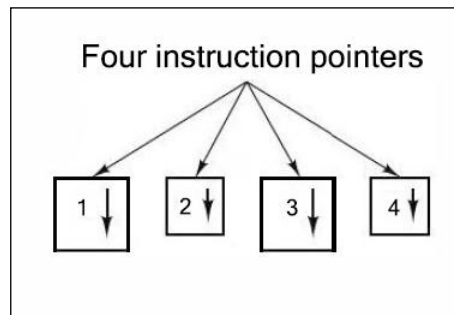
Operating systems over the years have evolved to a sequential conceptual model (sequential processes) where all the runnable software on the computer, including the operating system, is represented as a set of sequential processes. A process is an instance of an executing program. Each process has its own virtual address space and a thread of control. Thread is the basic unit for which **Scheduler** (operating system) allocates processor time. We can think of a system as a collection of processes running in a quasi-parallel environment. Rapid switching between processes (programs), back and forth, is called multitasking.

Processes and threads

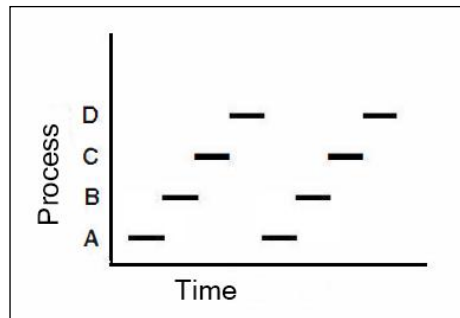
In the following figure, we see a single core CPU multitasking system with four programs scheduled for execution:



In the following figure, we have a multicore CPU multitasking system with four executing processes, each one with its own control flow and each one running independently of the other ones:



In the following figure, we see that over a time interval, processes have made progress, but at any instant of time, only one process can run on one CPU core:



As we said earlier, in any traditional operating system, each process has an address space and a single thread of control. We have many situations where multiple threads are executing in the address space of a process, running in a quasi-parallel context as they were separate processes. The main reason to have the threads is that in many applications, multiple operations are executed at once. Some of these operations may wait (block) for some time. By decomposing such an application into multiple sequential threads that run in a quasi-parallel context, the programming model becomes simpler. By adding threads, the operating system provides us with a new feature: the ability for parallel entities to share an address space and all of its data among themselves. This ability is essential for concurrent execution.

Explaining the process model

Traditional operating systems must provide a way to create and terminate processes. There are four principal events that cause processes to be created; they are as follows:

- ▶ System initialization
- ▶ Execution of a process-creation system call by a running process
- ▶ A user request to create a new process
- ▶ Initiation of a batch job

After the operating system has been started, several processes are created. Some of these are foreground processes. They are processes that interact with users (humans) and perform work for them. Others are background processes that are not associated with any particular user action but, instead, have some specific functions. For example, one background process may be designed to accept incoming e-mails; it might sleep most of the day but suddenly come to life when an incoming e-mail arrives. Background processes usually handle activities such as e-mails, printing, and so on.

In Windows, a single Win32 function call, `CreateProcess`, handles both process creation and loading process context. You can find more about `CreateProcess` on MSDN (<http://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx>), but we will demonstrate basic process creation and synchronization in the following example.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ Console application named `ProcessDemo`.
2. Open `ProcessDemo.cpp`.
3. Add the following code to it:

```
#include "stdafx.h"
#include <Windows.h>
#include <iostream>

using namespace std;

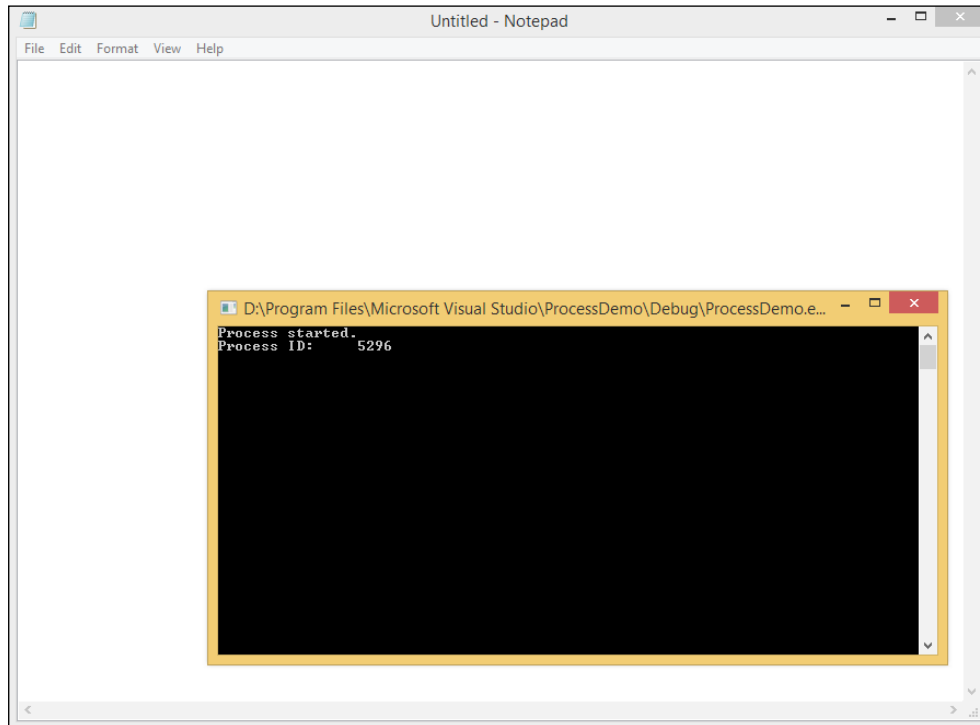
int _tmain(int argc, _TCHAR* argv[])
{
    STARTUPINFO startupInfo = { 0 };
    PROCESS_INFORMATION processInformation = { 0 };

    BOOL bSuccess = CreateProcess(
        TEXT("C:\\Windows\\notepad.exe"), NULL, NULL,
        NULL, FALSE, NULL, NULL, NULL, &startupInfo,
        &processInformation);

    if (bSuccess)
    {
        cout << "Process started." << endl
             << "Process ID:\\t"
             << processInformation.dwProcessId << endl;
    }
    else
    {
        cout << "Cannot start process!" << endl
             << "Error code:\\t" << GetLastError() << endl;
    }

    return system("pause");
}
```

The output of the program is shown in the following screenshot:



As you can see in the preceding screenshot, a new process (Notepad) has been started.

How it works...

The `CreateProcess` function is used to create a new process and its primary thread. The new process runs in the security context of the calling process.

The operating system assigns the process identifier to the process. The identifier is valid until the process terminates. It is used to identify the process, or for some APIs, such as the `OpenProcess` function, it is used to obtain a handle to the process. A thread identifier is also assigned to the initial thread in the process. It can be specified in the `OpenThread` function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the `PROCESS_INFORMATION` structure.

The calling thread can use the `WaitForInputIdle` function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because `CreateProcess` returns without waiting for the new process to finish its initialization. For example, the creating process would use the `WaitForInputIdle` function before trying to find a window associated with the new process.

The proper way to terminate a process is by calling the `ExitProcess` function, as it sends a notification of approaching termination to all DLLs attached to the process. Other means to shut down a process, such as using the `TerminateProcess` API, do not notify the attached DLLs. Note that when a thread calls the `ExitProcess` function, the other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of the attached DLLs).

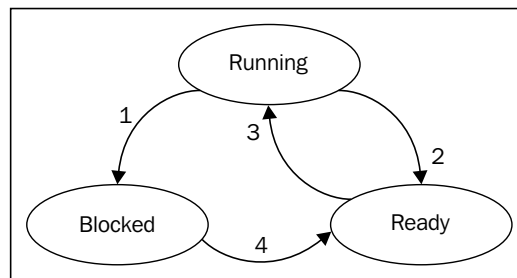
There's more...

Although each process is an independent entity, with its own instruction pointer and internal state, processes often need to interact with other processes. One process might generate some output data that another process might use as input data. Depending on the relative speed of the two processes, it might happen that in an example, the read operation is ready to run, but there is no input waiting for it. It must then be blocked until some input is available to read from. When a process is blocked, it logically cannot continue. This is because the process is waiting for input that is not yet available. It is also possible for a process to be stopped because the operating system has decided to allocate the CPU to another process for a while. These two conditions are completely different. In the first case, the suspension is inherent in the problem (you cannot parse the user's command line until it has been typed). In the second case, it is a technicality of the system (not enough CPUs to give each process its own time on the processor).

In the following figure, we see a state diagram that shows the three states a process may be in; they are as follows:

- ▶ **Running:** This means the process is actually using the CPU at that instant
- ▶ **Ready:** This means the process is runnable; it is temporarily stopped to let another process run
- ▶ **Blocked:** This means the process is unable to run until some external event occurs

The **Running** and **Ready** states are similar. In both cases, the process is willing to run. However, in the second one, there is temporarily no CPU available for the process. The **Blocked** state is different from the first two; the process cannot run, even if the CPU has nothing else to do.



Transition one sets the process in the blocked state for input. Transition two occurs when Scheduler picks another process for execution on the CPU. The third transition occurs when Scheduler picks this process, while the fourth transition occurs when the requested input becomes available.

Implementation of processes

In modern multitasking systems, the **Process Control Block (PCB)** stores many different items of data, all needed for correct and efficient process management. PCB is the data structure that resides in the operating system kernel which contains the information needed to manage the process. The PCB is the representation of the process in the operating system. Though the details of these structures are system-dependent, there are some very common parts, which could be classified into three main categories:

- ▶ Process identification data
- ▶ Processor state data
- ▶ Process control data

pointer	process state
process number	
instruction pointer	
registers	
memory limits	
list of open files	
⋮	

The role of the PCB is central in process management. It is accessed and/or modified by most operating system utilities, including those involved with scheduling, memory and I/O resource access, and performance monitoring. Data structuring for processes is often done in terms of PCBs. For example, pointers to other PCBs inside a PCB allow the creation of queues of processes in various scheduling states (**Ready**, **Blocked**, and so on), which we mentioned earlier.

The operating system must manage resources on behalf of processes. It must continuously take care of the state of each process and its resources and internal values. The following example will demonstrate how to obtain a *process basic information* structure address, where one of the attributes is the address of PCB. Another attribute is the unique process ID. For the simplicity of this example, we'll simply output a process ID read from the object.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Let's create another program that manipulates processes; only this time, we will obtain the process ID from the process basic information structure. Perform the following steps:

1. Create a new C++ Console application named `NtProcessDemo`.
2. Open `NtProcessDemo.cpp`.
3. Add the following code to it:

```
#include "stdafx.h"
#include <Windows.h>
#include <Winternl.h>
#include <iostream>

using namespace std;

typedef NTSTATUS(NTAPI* QEURYINFORMATIONPROCESS) (
    IN    HANDLE ProcessHandle,
    IN    PROCESSINFOCLASS ProcessInformationClass,
    OUT   PVOID ProcessInformation,
    IN    ULONG ProcessInformationLength,
    OUT   PULONG ReturnLength OPTIONAL
);

int _tmain(int argc, _TCHAR* argv[])
{
    STARTUPINFO startupInfo = { 0 };

```

```

PROCESS_INFORMATION processInformation = { 0 };

BOOL bSuccess = CreateProcess(
    TEXT("C:\\Windows\\notepad.exe"), NULL, NULL,
    NULL, FALSE, NULL, NULL, NULL, &startupInfo,
    &processInformation);

if (bSuccess)
{
    cout << "Process started." << endl << "Process ID:\\t"
        << processInformation.dwProcessId << endl;

    PROCESS_BASIC_INFORMATION pbi;
    ULONG uLength = 0;

    HMODULE hDll = LoadLibrary(
        TEXT("C:\\Windows\\System32\\ntdll.dll"));

    if (hDll)
    {
        QEURYINFORMATIONPROCESS QueryInformationProcess =
            (QEURYINFORMATIONPROCESS)GetProcAddress(
                hDll, "NtQueryInformationProcess");

        if (QueryInformationProcess)
        {
            NTSTATUS ntStatus = QueryInformationProcess(
                processInformation.hProcess,
                PROCESSINFOCLASS::ProcessBasicInformation,
                &pbi, sizeof(pbi), &uLength);

            if (NT_SUCCESS(ntStatus))
            {
                cout << "Process ID (from PCB):\\t"
                    << pbi.UniqueProcessId << endl;
            }
            else
            {
                cout << "Cannot open PCB!" << endl
                    << "Error code:\\t" << GetLastError()
                    << endl;
            }
        }
    }
    else
    {
        cout << "Cannot get "
            << "NtQueryInformationProcess function!"
            << endl << "Error code:\\t"
            << GetLastError() << endl;
    }
}

```

```
    }
    FreeLibrary(hDll);
}
else
{
    cout << "Cannot load ntdll.dll!" << endl
          << "Error code:\t" << GetLastError() << endl;
}
}
else
{
    cout << "Cannot start process!" << endl
          << "Error code:\t" << GetLastError() << endl;
}
return 0;
}
```

How it works...

For this example, we used some additional header files: `Winternl.h` and `Windows.h`. Header `Winternl` include prototypes for the majority of Windows' internal routines and data representation; this includes the `PROCESS_BASIC_INFORMATION` structure with the following definition:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    PVOID Reserved1;
    PPEB PebBaseAddress;
    PVOID Reserved2[2];
    ULONG_PTR UniqueProcessId;
    PVOID Reserved3;
} PROCESS_BASIC_INFORMATION;
```

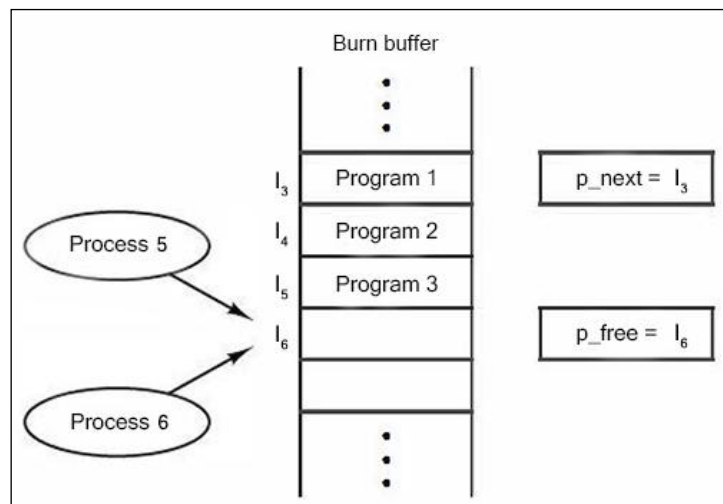
The preceding structure is used by the operating system in calls made between the user mode and kernel mode subroutines. In combination with the `PROCESSINFOCLASS::ProcessBasicInformation` enumeration, we are using the `UniqueProcessId` attribute to obtain the process identifier, only as a demonstration.

First, we will define `QEURYINFORMATIONPROCESS`, a **typedef** for the `NtQueryInformationProcess` function loaded from the `ntdll.dll` library. When the address of the function is obtained using the `GetProcAddress` Win32 API, we can query for the `PROCESS_BASIC_INFORMATION` object. Pay attention to the `PebBaseAddress` field; it is a pointer to the PCB of the newly created process. If you want to explore the PCB further to check the newly created process, you would have to use the `ReadProcessMemory` routine in runtime, because the memory to which `PebBaseAddress` points belongs to the newly created process.

IPC – Interprocess Communication

It is very important that processes have some way of communicating with each other. Operating systems provide such mechanisms, but we need to focus on issues related with it. For example, what if two processes in an airline reservation system are trying to sell the last seat on the plane at the same time? Of course, this is very wrong because you can't sell the same seat twice. There is also a dependency problem: if process A produces some kind of data and process B is reading that data (for example, to print it), then process B must wait until process A really produces the required data. The difference between processes and threads is that the first task for the thread is much easier to implement due to fact that threads share a common address space while processes don't. Regarding the second issue, it is equally applicable to threads; that's why it's very important to understand synchronizing mechanisms.

To see how IPC appears, let's consider a simple example: a CD burner. When a process wants to burn something, it sets the file handles (we are burning more files at once) in a special burn buffer. Another process, responsible for burning, checks to see if there are any files to be burned and if there are, it burns them and then removes their handles from the buffer. Let's assume that the burn buffer has enough indexes, numbered **10**, **11**, **12**, and so on, each one of them capable of holding an adequate number of the requested file handles. Let's also assume that there are two shared variables: `p_next`, which points to the next buffer index to be burned, and `p_free`, which points to the next free index in the buffer. These two variables must be available to all processes. At a certain instant of time, indexes **10** to **12** are empty (the files have already been burned) and slots **13** to **15** are filled. Simultaneously, processes **5** and **6** decide they want to queue the file handles for burning. This situation is shown in the following figure:



The following can happen when **Process 5** reads `p_free` and stores the value, **I6**, in a local variable called `f_slot`. Then, a clock interrupt occurs, and the CPU decides that **Process 5** has run long enough, so it switches to **Process 6**. **Process 6** also reads `p_free`, and also gets a 7. It too stores it `p_free` its local variable `f_slot`. At this instant, both processes think that the next available index is **I6**. **Process 6** now continues to run. It stores the handles of its files at index **I6** and updates `p_free` to be an **I7**. The system then puts **Process 6** to sleep. Now, **Process 5** runs again, starting from the place it left off. It looks at `f_slot`, finds an **I6** there, and writes its file handle at index **I6**, erasing the handle that **Process 6** just put there. Then, it computes `f_slot + 1`, which is **I7**, and sets `p_free` to **I7**. The burn buffer is now internally consistent, so the burner process will not notice anything wrong, but **Process 6** will never receive any output.

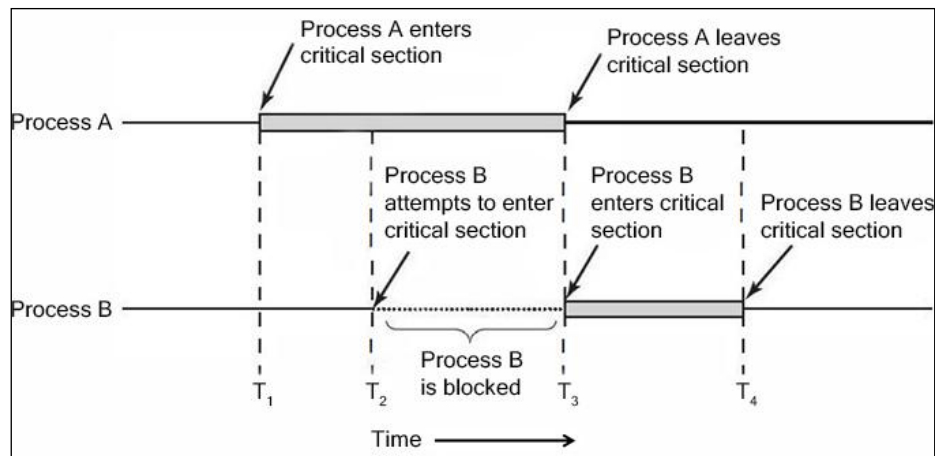
Process 6 will hang around for an infinite period of time, waiting for an output that will never occur. Situations like this, where two or more entities are reading or writing some shared data and the final result depends on which one runs precisely when, are called **race conditions**.

How to avoid race conditions? Most of the solutions involve shared memory, shared files, and everything else that can help us deny a process from reading while the other process is writing the shared data at the same time. In another words, we need **mutual exclusion** or a mechanism that will provide exclusive access to a shared object, whether it is a shared variable or file, or any other object. The difficulty can occur when **Process 6** starts using one of the shared objects before **Process 5** has finished with it.

That part of the program where the shared memory is accessed is called the **critical section** or **critical region**. We need to assure that no two processes ever get in their critical sections at the same time in order to avoid race condition. Even though race condition is avoided, this approach may not be efficient when having parallel processes, which are intended to cooperate correctly and efficiently. If we intend to use shared data, we must take care of the following four conditions:

- ▶ No two processes may be simultaneously inside their critical regions
- ▶ No assumptions may be made about speeds or the number of CPUs
- ▶ No process running outside its critical region may block other processes
- ▶ No process should have to wait forever to enter its critical region

What we described earlier is shown in the following figure. **Process A** enters its critical section at time **T1**. In some time, **T2** (later **Process B**) attempts to enter its critical section but fails because another process is already in its critical section, and we allow only one process at a time in the critical section. **Process B** must be temporarily suspended until time **T3** when **Process A** leaves its critical section, allowing **Process B** to enter immediately. Eventually, **Process B** leaves (at **T4**), and we are back to the beginning where no process is in its critical section.



We will show an example of Interprocess Communication by creating a program that will start two processes that need to draw random rectangles in a common window. Processes need to communicate with each other in a way that when one process is drawing, the other waits.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

1. Create a new C++ Console application. Name it IPCDemo.
2. Right click on **Solution explorer** and select **Add New Project**. Select another C++ **Win32 Console Application** and name it IPCWorker.
3. Open the IPCWorker.cpp file and insert the following code:

```
#include "stdafx.h"
#include <Windows.h>

#define COMMUNICATION_OBJECT_NAME TEXT("__FILE_MAPPING__")
#define SYNCHRONIZING_MUTEX_NAME TEXT( "__TEST_MUTEX__" )

typedef struct _tagCOMMUNICATIONOBJECT
{
    HWND hWndClient;
    BOOL bExitLoop;
    LONG lSleepTimeout;
```



```
    } COMMUNICATIONOBJECT, *PCOMMUNICATIONOBJECT;

int _tmain(int argc, _TCHAR* argv[])
{
    HBRUSH hBrush = NULL;

    if (_tcscmp(TEXT("blue"), argv[0]) == 0)
    {
        hBrush = CreateSolidBrush(RGB(0, 0, 255));
    }
    else
    {
        hBrush = CreateSolidBrush(RGB(255, 0, 0));
    }

    HWND hWnd = NULL;
    HDC hDC = NULL;
    RECT rectClient = { 0 };
    LONG lWaitTimeout = 0;
    HANDLE hMapping = NULL;
    PCOMMUNICATIONOBJECT pCommObject = NULL;
    BOOL bContinueLoop = TRUE;

    HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE,
        SYNCHRONIZING_MUTEX_NAME);
    hMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
        COMMUNICATION_OBJECT_NAME);

    if (hMapping)
    {
        while (bContinueLoop)
        {
            WaitForSingleObject(hMutex, INFINITE);
            pCommObject = (PCOMMUNICATIONOBJECT) MapViewOfFile(hMapping,
                FILE_MAP_READ, 0, 0, sizeof(COMMUNICATIONOBJECT));

            if (pCommObject)
            {
                bContinueLoop = !pCommObject->bExitLoop;
                hWnd = pCommObject->hWndClient;
                lWaitTimeout = pCommObject->lSleepTimeout;
                UnmapViewOfFile(pCommObject);
                hDC = GetDC(hWnd);
                if (GetClientRect(hWnd, &rectClient))
```

```

        {
            FillRect(hDC, &rectClient, hBrush);
        }
        ReleaseDC(hWnd, hDC);
        Sleep(lWaitTimeout);
    }
    ReleaseMutex(hMutex);
}
}
CloseHandle(hMapping);
CloseHandle(hMutex);
DeleteObject(hBrush);

return 0;
}

```

4. Open IPCDemo.cpp and insert following code:

```

#include "stdafx.h"
#include <Windows.h>
#include <iostream>

using namespace std;

#define COMMUNICATION_OBJECT_NAME TEXT("__FILE_MAPPING__")
#define SYNCHRONIZING_MUTEX_NAME TEXT( "__TEST_MUTEX__" )
#define WINDOW_CLASS_NAME TEXT( "__TMPWNDCLASS__" )
#define BUTTON_CLOSE 100

typedef struct _tagCOMMUNICATIONOBJECT
{
    HWND hWndClient;
    BOOL bExitLoop;
    LONG lSleepTimeout;
} COMMUNICATIONOBJECT, *PCOMMUNICATIONOBJECT;

LRESULT CALLBACK WndProc(HWND hDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam);
HWND InitializeWnd();
PCOMMUNICATIONOBJECT pCommObject = NULL;
HANDLE hMapping = NULL;

int _tmain(int argc, _TCHAR* argv[])
{
    cout << "Interprocess communication demo." << endl;

```

```

        Hwnd hWnd = InitializeWnd();
        if (!hWnd)
        {
            cout << "Cannot create window!" << endl << "Error:\t" <<
GetLastError() << endl;
            return 1;
        }
        HANDLE hMutex = CreateMutex(NULL, FALSE,
SYNCHRONIZING_MUTEX_NAME);
        if (!hMutex)
        {
            cout << "Cannot create mutex!" << endl << "Error:\t" <<
GetLastError() << endl;
            return 1;
        }
        hMapping = CreateFileMapping((HANDLE)-1, NULL, PAGE_READWRITE,
0, sizeof(COMMUNICATIONOBJECT), COMMUNICATION_OBJECT_NAME);
        if (!hMapping)
        {
            cout << "Cannot create mapping object!" << endl << "Error:\t"
<< GetLastError() << endl;
            return 1;
        }
        pCommObject = (PCOMMUNICATIONOBJECT) MapViewOfFile(hMapping,
FILE_MAP_WRITE, 0, 0, 0);
        if (pCommObject)
        {
            pCommObject->bExitLoop = FALSE;
            pCommObject->hWndClient = hWnd;
            pCommObject->lSleepTimeout = 250;
            UnmapViewOfFile(pCommObject);
        }

        STARTUPINFO startupInfoRed = { 0 };
        PROCESS_INFORMATION processInformationRed = { 0 };
        STARTUPINFO startupInfoBlue = { 0 };
        PROCESS_INFORMATION processInformationBlue = { 0 };

        BOOL bSuccess = CreateProcess( TEXT("../Debug\\IPCWorker.exe"),
TEXT("red"), NULL, NULL, FALSE, 0, NULL, NULL, &startupInfoRed,
&processInformationRed);
        if (!bSuccess)
        {

```

```

        cout << "Cannot create process red!" << endl << "Error:\t" <<
GetLastError() << endl;
        return 1;
    }
    bSuccess = CreateProcess( TEXT("../\\Debug\\IPCWorker.exe"),
TEXT("blue"), NULL, NULL, FALSE, 0, NULL, NULL, &startupInfoBlue,
&processInformationBlue);
    if (!bSuccess)
    {
        cout << "Cannot create process blue!" << endl << "Error:\t" <<
GetLastError() << endl;
        return 1;
    }
    MSG msg = { 0 };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    UnregisterClass(WINDOW_CLASS_NAME, GetModuleHandle(NULL));
    CloseHandle(hMapping);
    CloseHandle(hMutex);
    cout << "End program." << endl;
    return 0;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case BUTTON_CLOSE:
                {
                    PostMessage(hWnd, WM_CLOSE, 0, 0);
                    break;
                }
            }
            break;
        }
        case WM_DESTROY:

```

```

    {
        pCommObject = (PCOMMUNICATIONOBJECT) MapViewOfFile(hMapping,
FILE_MAP_WRITE, 0, 0, 0);
        if (pCommObject)
        {
            pCommObject->bExitLoop = TRUE;
            UnmapViewOfFile(pCommObject);
        }

        PostQuitMessage(0);
        break;
    }
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}

```

```

HWND InitializeWnd()
{
    WNDCLASSEX wndEx;
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WndProc;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = GetModuleHandle(NULL);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = WINDOW_CLASS_NAME;
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
    if (!RegisterClassEx(&wndEx))
    {
        return NULL;
    }
    HWND hWnd = CreateWindow(wndEx.lpszClassName,
        TEXT("Interprocess communication Demo"),

```

```

        WS_OVERLAPPEDWINDOW, 200, 200, 400, 300, NULL, NULL,
        wndEx.hInstance, NULL);
    if (!hWnd)
    {
        return NULL;
    }
    HWND hButton = CreateWindow(TEXT("BUTTON"), TEXT("Close"),
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP,
        275, 225, 100, 25, hWnd, (HMENU)BUTTON_CLOSE, wndEx.hInstance,
    NULL);
    HWND hStatic = CreateWindow(TEXT("STATIC"), TEXT(""), WS_CHILD |
    WS_VISIBLE, 10, 10, 365, 205, hWnd, NULL, wndEx.hInstance, NULL);
    ShowWindow(hWnd, SW_SHOW);
    UpdateWindow(hWnd);
    return hStatic;
}

```

How it works...

Our example was slightly different this time, because we had to create two projects in the same solution due to fact that we are working with separate processes.

Let's begin with `IPCWorker`. The idea is that we need process that will do some work for us. For the simplicity of the example, we will create two processes from our main application, `IPCDemo`, which will paint a certain region of the application window. Without proper communication and process synchronization, multiple access to shared resources would occur. The operating system switches between processes quickly and, considering the fact that on the majority of PCs there is more than one CPU core, most likely, processes would paint the region at the same time, creating an unwanted situation where multiple processes are accessing an unprotected region at once.

We are using a mapping object—the location in memory allocated for our processes to read/write from. `IPCWorker`, or simply `worker`, needs to ask for a named mutex and, if granted, it can proceed and obtain a pointer to the memory region (file mapping) where information for it will be stored. Mutex must be acquired because of exclusive access. A process gets mutex after `WaitForSingleObject` returns. Let's review the following statements:

```

HANDLE hMutex = OpenMutex( MUTEX_ALL_ACCESS, FALSE,
    SYNCHRONIZING_MUTEX_NAME );

```

We are allocating a handle for the mutex (`hMutex`). A call to the `OpenMutex` Win32 API results in obtaining a handle to the named mutex, that is, if mutex exists. Please check the next statement:

```

WaitForSingleObject( hMutex, INFINITE );

```

After the preceding statement, the execution continues when the `WaitForSingleObject` API returns.

```
pCommObject = ( PCOMMUNICATIONOBJECT )
MapViewOfFile( hMapping, FILE_MAP_READ, 0, 0,
sizeof( COMMUNICATIONOBJECT ) );
```

A call to `MapViewOfFile` Win32 API obtains the handle (pointer) to the file mapping object. Now, the process can read from the shared memory object and obtain the required information. The process needs to read the `bExitLoop` variable in order to sustain whether it will continue execution. After that, it needs to read the window handle (`hWnd`) whose window region needs to be painted. Finally, the `lSleepTimeout` variable interval is needed to sustain for how long the process will sleep. We intentionally added sleep time, as switching between processes takes place too quickly to be noticed.

```
ReleaseMutex( hMutex );
```

A call to the `ReleaseMutex` Win32 API releases mutex ownership, making the other process able to obtain mutex and continue execution. Let's go back to the `IPCDemo` project. We defined the `_tagCOMMUNICATIONOBJECT` struct for communication between objects through file mapping.

File mapping is the association of a file's contents with a portion of the virtual address space of a process. The system creates a file mapping object (also known as a section object) to maintain this association. A file view is the portion of virtual address space that a process uses to access the file's contents. File mapping allows the process to use both random I/O and sequential I/O. It also allows the process to work efficiently with a large data file, such as a database, without having to map the whole file into memory. Multiple processes can also use memory-mapped files to share data. You can find more on MSDN (<http://msdn.microsoft.com/en-us/library/windows/desktop/aa366883%28v=vs.85%29.aspx>).

`IPCDemo` creates the file mapping before it runs worker processes. That's why we could ask for file mapping from worker process without checking if it is created first. After `IPCDemo` creates and initializes the application window and region to paint, it creates a named mutex, and file mapping. Afterwards, processes are created with different command-line arguments to distinguish them.

The `WndProc` routine will handle the `WM_COMMAND` and `WM_DESTROY` messages. The first one will be triggered on a button push event when we need to inform the application that it needs to safely close. The other one release used file mapping and posts the following close message to the main thread message queue:

```
PostQuitMessage( 0 );
```

There's more...

File mapping operates with content on the disk and view in memory. View in memory is much faster to read/write from than to use the hard disk drive. Using file mapping is a good practice when we want to use shared objects between processes for simple purposes. When we supply -1 for the first argument in a call to Win32 API

```
CreateFileMapping( ( HANDLE ) -1, NULL, PAGE_READWRITE, 0,
sizeof( COMMUNICATIONOBJECT ), COMMUNICATION_OBJECT_NAME );
```

file on disk won't exist at all. This is good because we intend to use a portion of memory; it is faster and just enough for what we need.

Pay attention when calling IPCWorker processes. For debugging purposes, we set calls to the following:

```
bSuccess = CreateProcess( TEXT( "..\\Debug\\IPCWorker.exe" ),
TEXT( "red" ), NULL, NULL, FALSE, 0, NULL, NULL,
&startupInfoRed, &processInformationRed );
```

This is because, in debugging mode, Visual Studio doesn't start from the program `exe` folder, but from the project folder. Also, Visual Studio by default puts all Win32 project outputs in the same folder, so in the file path, we had to move one step (folder) up from the project folder and then to the **Debug** folder where the entire project outputs (`exe`) reside. If you would not like to start `exe` from VS, you would have to change the path in `CreateProcess` calls or add functionality to pass the file path through the command line or something similar.

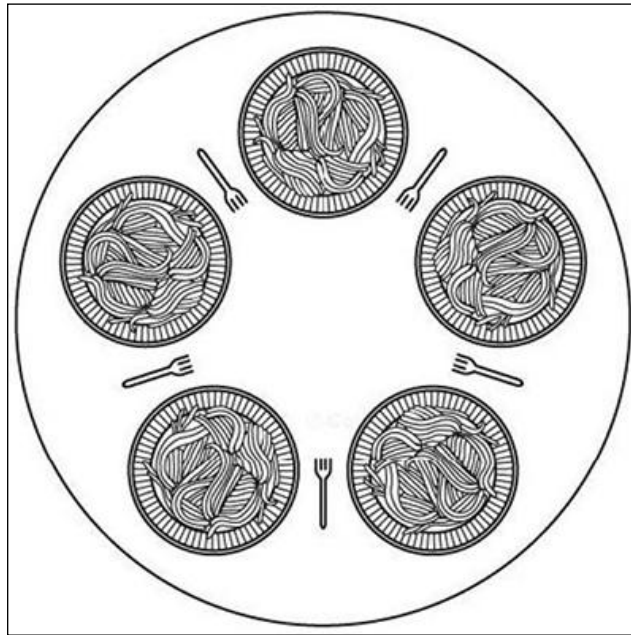
Solving classical IPC problems

Inter process communication is very important. Its implementation, on the other hand, is very complex. Operating system designers (and developers) may face various problems. Here, we will explain some of the most common problems.

The dining philosophers problem

The definition for the dining philosopher's problem was taken from Andrew Tanenbaum's book, *Modern Operating Systems Third Edition*. The authors' solution provided for this book is genuine.

In 1965, Dijkstra proposed and solved a synchronization problem that he called the dining philosophers' problem. The problem is quite simply as follows: five philosophers are seated around a circular table. Each philosopher has a plate of spaghetti. The spaghetti is so slippery that a philosopher needs two forks to eat it. Between each pair of plates is one fork. The layout of the table is illustrated in the following figure:



An assumption is that the life of a philosopher consists of alternate periods of eating and thinking. When a philosopher gets hungry, he tries to acquire his left and right forks, one at a time, in either order. If he is successful in acquiring two forks, he eats for a while, then puts down the forks, and continues to think. The key question is: can you write a program for each philosopher that does what it is supposed to do and never get stuck?

We could wait until the specified fork is available and then seize it. Unfortunately, the obvious solution is wrong. Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.

We could modify the program so that, after taking the left fork, the program checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process. This proposal too fails, although for a different reason. With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, and picking up their left forks again simultaneously, and so on, forever. A situation like this in which all the programs continue to run indefinitely but fail to make any progress is called starvation.

One improvement that has no deadlock and no starvation is to protect the five statements that follow the call to *think* by a mutex. Before starting to acquire forks, a philosopher would ask for a mutex. As we said earlier, mutex stands for mutual exclusion or the ability to provide exclusive access to some object. After replacing the forks, he would release the mutex. From a theoretical viewpoint, this solution is adequate. From a practical one, it has a performance issue: only one philosopher can eat at any instant of time. With five forks available, we should be able to allow two philosophers to eat at the same time.

The complete solution is given in the following example.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

1. Create a new Win32 project. Name it `PhilosophersDinner`.
2. Open `stdafx.h` and paste the following code:

```
#pragma once
#include "targetver.h"
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <commctrl.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <stdio.h>
#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \\  
name='Microsoft.Windows.Common-Controls' \\  
version='6.0.0.0' processorArchitecture='*' \\  
publicKeyToken='6595b64144ccf1df' language='*'\\" )
```

3. Open `PhilosophersDinner.cpp` and paste the following code:

```
#include "stdafx.h"

#define BUTTON_CLOSE      100
#define PHILOSOPHER_COUNT 5
#define WM_INVALIDATE     WM_USER + 1

typedef struct _tagCOMMUNICATIONOBJECT
{
    HWND  hWnd;
```

```

    bool    bExitApplication;
    int      iPhilosopherArray[PHILOSOPHER_COUNT];
    int      PhilosopherCount;
} COMMUNICATIONOBJECT, *PCOMMUNICATIONOBJECT;

HWND InitInstance(HINSTANCE hInstance, int nCmdShow);
ATOM MyRegisterClass(HINSTANCE hInstance);
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);
int PhilosopherPass(int iPhilosopher);
void FillEllipse(HWND hWnd, HDC hDC, int iLeft, int iTop, int
iRight, int iBottom, int iPass);

TCHAR* szTitle = TEXT("Philosophers Dinner Demo");
TCHAR* szWindowClass = TEXT("__PD_WND_CLASS__");
TCHAR* szSemaphoreName = TEXT("__PD_SEMAPHORE__");
TCHAR* szMappingName = TEXT("__SHARED_FILE_MAPPING__");
PCOMMUNICATIONOBJECT pCommObject = NULL;

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    HANDLE hMapping = CreateFileMapping((HANDLE)-1, NULL,
PAGE_READWRITE, 0, sizeof(COMMUNICATIONOBJECT), szMappingName);
    if (!hMapping)
    {
        MessageBox(NULL, TEXT("Cannot open file mapping"),
TEXT("Error!"), MB_OK);
        return 1;
    }
    pCommObject = (PCOMMUNICATIONOBJECT) MapViewOfFile(hMapping,
FILE_MAP_ALL_ACCESS, 0, 0, 0);
    if (!pCommObject)
    {
        MessageBox(NULL, TEXT("Cannot get access to file mapping! "),
TEXT("Error!"), MB_OK);
        CloseHandle(hMapping);
        return 1;
    }
    InitCommonControls();
    MyRegisterClass(hInstance);
    HWND hWnd = NULL;

```

```

    if (!(hWnd = InitInstance(hInstance, nCmdShow)))
    {
        return FALSE;
    }
    pCommObject->bExitApplication = false;
    pCommObject->hWnd = hWnd;
    memset(pCommObject->iPhilosopherArray, 0,
sizeof(*pCommObject->iPhilosopherArray));
    pCommObject->PhilosopherCount = PHILOSOPHER_COUNT;
    HANDLE hSemaphore = CreateSemaphore(NULL,
int(PHILOSOPHER_COUNT / 2), int(PHILOSOPHER_COUNT / 2),
szSemaphoreName);
    STARTUPINFO startupInfo[PHILOSOPHER_COUNT] =
{ { 0 }, { 0 }, { 0 }, { 0 }, { 0 } };
    PROCESS_INFORMATION processInformation[PHILOSOPHER_COUNT] =
{ { 0 }, { 0 }, { 0 }, { 0 }, { 0 } };

    HANDLE hProcesses[PHILOSOPHER_COUNT];
    TCHAR szBuffer[8];
    for (int iIndex = 0; iIndex < PHILOSOPHER_COUNT; iIndex++)
    {
#ifdef UNICODE
        wsprintf(szBuffer, L"%d", iIndex);
#else
        sprintf(szBuffer, "%d", iIndex);
#endif

        if (CreateProcess( TEXT("../Debug\\Philosopher.exe"),
szBuffer, NULL, NULL,
FALSE, 0, NULL, NULL, &startupInfo[iIndex],
&processInformation[iIndex]))
        {
            hProcesses[iIndex] = processInformation[iIndex].hProcess;
        }
    }
    MSG msg = { 0 };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    pCommObject->bExitApplication = true;
    UnmapViewOfFile(pCommObject);
    WaitForMultipleObjects(PHILOSOPHER_COUNT, hProcesses, TRUE,
INFINITE);
    for (int iIndex = 0; iIndex < PHILOSOPHER_COUNT; iIndex++)

```

```

    {
        CloseHandle(hProcesses[iIndex]);
    }
    CloseHandle(hSemaphore);
    CloseHandle(hMapping);
    return (int)msg.wParam;
}

ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEX wndEx;
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WndProc;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hInstance;
    wndEx.hIcon = LoadIcon(hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
    return RegisterClassEx(&wndEx);
}

HWND InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    HWND hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPED
| WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX, 200, 200, 540, 590,
NULL, NULL, hInstance, NULL);
    if (!hWnd)
    {
        return NULL;
    }

    HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE, FALSE,
FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH |
FF_MODERN, TEXT("Microsoft Sans Serif"));

    HWND hButton = CreateWindow(TEXT("BUTTON"), TEXT("Close"),
WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 410, 520, 100,
25, hWnd, (HMENU)BUTTON_CLOSE, hInstance, NULL);

```

```
SendMessage(hButton, WM_SETFONT, (WPARAM)hFont, TRUE);
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
return hWnd;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case BUTTON_CLOSE:
                {
                    DestroyWindow(hWnd);
                    break;
                }
            }
            break;
        }
        case WM_INVALIDATE:
        {
            InvalidateRect(hWnd, NULL, TRUE);
            break;
        }
        case WM_PAINT:
        {
            PAINTSTRUCT paintStruct;
            HDC hDC = BeginPaint(hWnd, &paintStruct);
            FillEllipse(hWnd, hDC, 210, 10, 310, 110,
PhilosopherPass(1));
            FillEllipse(hWnd, hDC, 410, 170, 510, 270,
PhilosopherPass(2));
            FillEllipse(hWnd, hDC, 335, 400, 435, 500,
PhilosopherPass(3));
            FillEllipse(hWnd, hDC, 80, 400, 180, 500,
PhilosopherPass(4));
            FillEllipse(hWnd, hDC, 10, 170, 110, 270,
PhilosopherPass(5));
            EndPaint(hWnd, &paintStruct);
            break;
        }
    }
}
```

```

        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        default:
        {
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
        }
    }
    return 0;
}

int PhilosopherPass(int iPhilosopher)
{
    return pCommObject->iPhilosopherArray[iPhilosopher - 1];
}

void FilleEllipse(HWND hWnd, HDC hDC, int iLeft, int iTop, int
iRight, int iBottom, int iPass)
{
    HBRUSH hBrush = NULL;
    if (iPass)
    {
        hBrush = CreateSolidBrush(RGB(255, 0, 0));
    }
    else
    {
        hBrush = CreateSolidBrush(RGB(255, 255, 255));
    }
    HBRUSH hOldBrush = (HBRUSH) SelectObject(hDC, hBrush);
    Ellipse(hDC, iLeft, iTop, iRight, iBottom);
    SelectObject(hDC, hOldBrush);
    DeleteObject(hBrush);
}

```

4. Right click on **Solution explorer** and add a new Win32 Console application project. Name it Philosopher.
5. Open stdafx.h and paste the following code:

```

#pragma once
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
#include <windows.h>.

```

6. Open `Philosopher.cpp` and paste the following code:

```

#include "stdafx.h"
#include <Windows.h>

#define EATING_TIME      1000
#define PHILOSOPHER_COUNT  5
#define WM_INVALIDATE     WM_USER + 1

typedef struct _tagCOMMUNICATIONOBJECT
{
    HWND  hWnd;
    bool  bExitApplication;
    int    iPhilosopherArray[PHILOSOPHER_COUNT];
    int    PhilosopherCount;
} COMMUNICATIONOBJECT, *PCOMMUNICATIONOBJECT;

void Eat();
TCHAR* szSemaphoreName = TEXT("__PD_SEMAPHORE__");
TCHAR* szMappingName = TEXT("__SHARED_FILE_MAPPING__");
bool bExitApplication = false;

int _tmain(int argc, _TCHAR* argv[])
{
    HWND hConsole = GetConsoleWindow();
    ShowWindow(hConsole, SW_HIDE);
    int iIndex = (int)_tcstol(argv[0], NULL, 10);
    HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE,
szMappingName);
    while (!bExitApplication)
    {
        HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
szSemaphoreName);
        WaitForSingleObject(hSemaphore, INFINITE);
        PCOMMUNICATIONOBJECT pCommObject = (PCOMMUNICATIONOBJECT)
MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0,
sizeof(COMMUNICATIONOBJECT));
        bExitApplication = pCommObject->bExitApplication;
        if (!pCommObject->iPhilosopherArray[
(iIndex + pCommObject->PhilosopherCount - 1)
% pCommObject->PhilosopherCount]
&& !pCommObject->iPhilosopherArray[
(iIndex + 1) % pCommObject->PhilosopherCount])
        {
            pCommObject->iPhilosopherArray[iIndex] = 1;
            Eat();

```



```
    }
    SendMessage(pCommObject->hWnd, WM_INVALIDATE, 0, 0);
    pCommObject->iPhilosopherArray[iIndex] = 0;
    UnmapViewOfFile(pCommObject);
    ReleaseSemaphore(hSemaphore, 1, NULL);
    CloseHandle(hSemaphore);
}
CloseHandle(hMapping);
return 0;
}

void Eat()
{
    Sleep(EATING_TIME);
}
```

How it works...

We have created five processes to mimic five philosophers. Each philosopher (process) has to think and eat. If a philosopher wants to eat, they need two forks, so they must ask for the left fork and, if it is available, then they ask for right fork. If both forks are available, it is possible to eat; if not, the philosopher leaves the left fork (if obtained) and waits for their next turn. We've set the eating time at 1 second.

PhilosopherDinner is our main app. By creating file mapping, we are able to communicate with other processes. Then it's important to synchronize processes by creating a `Semaphore` object. As we said earlier, using mutex would allow us to make only one philosopher dining at one instance of time. It would work, but it wouldn't be optimized enough. If we need two forks for each philosopher to eat, then we can have `FLOOR(NUMBER_OF_PHILOSOPHERS / 2)` eating simultaneously. That's why we would set a maximum of two objects that can pass semaphore at one instance of time, as shown in the following code:

```
HANDLE hSemaphore = CreateSemaphore( NULL,
    int( PHILOSOPHER_COUNT / 2 ),
    int( PHILOSOPHER_COUNT / 2 ), szSemaphoreName );
```

What is important for us to know is that semaphore can initially have a certain number of objects that can pass it, but this number can be increased with the third parameter of the `CreateSemaphore` API. In our example, we won't need this feature.

After initializing the semaphore object, the processes are created, and the application can enter its message loop. Let's review the other part of the application—`Philosopher` application. It is a console application and because we need no interface, we will hide its main window (in this case, console) as shown in the following code:

```
HWND hConsole = GetConsoleWindow( );
ShowWindow( hConsole, SW_HIDE );
```

The application then needs to obtain its index (philosopher name):

```
int iIndex = ( int ) _tcstol( argv[ 0 ], NULL, 10 );
```

After that, the philosopher has to obtain a handle to the file mapping object and enter the loop. In the loop, the philosopher waits for their turn by requesting the pass-through semaphore object. When the philosopher gets a semaphore pass, it can obtain both forks. Then, it sends a message to update the main app user interface with the following `SendMessage` API:

```
SendMessage( pCommObject->hWnd, WM_INVALIDATE, 0, 0 );
```

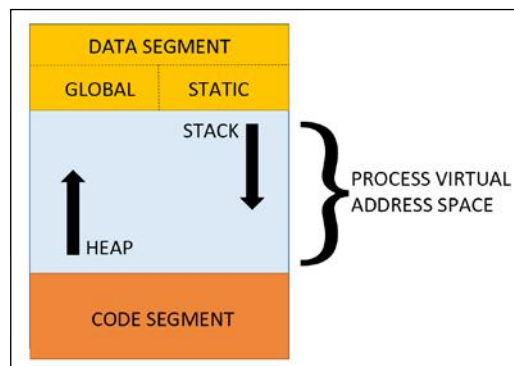
When all work is done, it can release the semaphore object and continue execution.

See also...

There are other classical IPC problems such as "Sleeping barber" and "Producer-Consumer problem". Producer-Consumer problem will be solved later in the chapter.

Implementation of the thread model

We can say that the process is an object whose task is to group related resources together. Every process has an address space with a representation, as shown in the following figure.

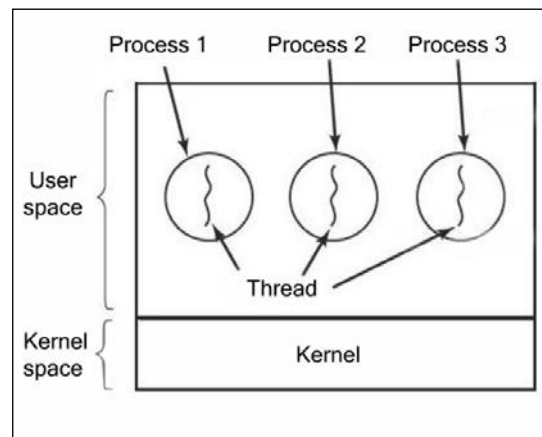


This so-called process image must be loaded in the physical memory on initialization of `CreateProcess`. All resources such as file handles, information about child processes, pending alarms, signaling handlers, and so on are stored. They can be managed more easily by grouping them together in the form of a process.

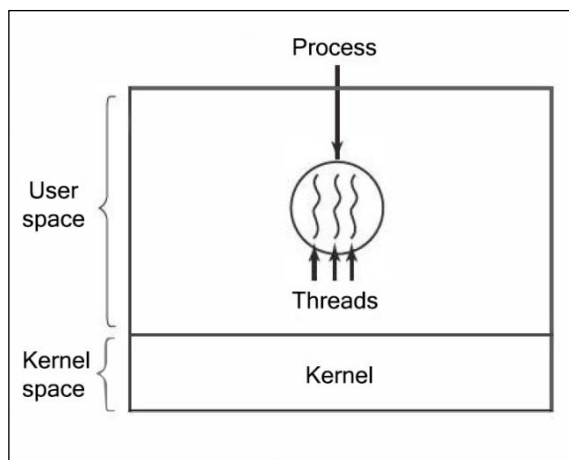
The other important concept besides process is a thread of execution, which is usually called **thread**. The thread is the lowest unit that can be scheduled for execution on the CPU. In another words, the process itself cannot get CPU time, only one of its threads can. The thread saves information about CPU registers with its working variables, along with the stack. The stack contains data related to functions call - with one frame for each function called but not yet returned from. A thread can be executed on the CPU, not a process. However, a process must have at least one thread, which is usually called main thread, so when we say that a process is executing on the CPU, we are referring to the process' main thread.

Processes are used to group resources together; threads are the entities scheduled for execution on the CPU. It is important to understand that a thread allows multiple executions to take place in the same process environment. Having multiple threads running in parallel in a single process context is the same as having multiple processes running in parallel on one computer. The term "multithreading" is used to describe the situation where multiple threads are running in the context of the single process.

In the following figure, we see three processes:



In the following figure, we see a single process with three threads of control. Although in both cases we have three threads, in the first figure, each of the threads operates in a different address space, whereas in the second figure, all the three threads share the same address space.



When a multithreaded process is run on a single CPU system, the threads take turns running. By switching between multiple processes, the system gives the illusion of parallelism. Multithreading works in the same way. With three threads in a process, the threads appear to run in parallel, each one on a CPU, with one-third the time that the CPU has scheduled for its process (approximately, because CPU time depends on OS, scheduling algorithm and more). On a multiprocessor system, the situation is similar, only that each CPU core executes threads in same manner as described here. The good side is that with more cores, more threads can run in parallel, giving us the power of native hardware parallelism and multithreaded execution.

In our next example, we will demonstrate basic thread usage by implementing a simple array sorting using two threads.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

1. Create a new empty Win32 Console application. Name it `MultithreadedArraySort`.
2. Open `MultithreadedArraySort.cpp` and paste the following code:

```
#include "stdafx.h"
#include <Windows.h>
#include <iostream>
#include <tchar.h>
```

```
using namespace std;

#define THREADS_NUMBER    2
#define ELEMENTS_NUMBER   200
#define BLOCK_SIZE        ELEMENTS_NUMBER / THREADS_NUMBER
#define MAX_VALUE         1000

typedef struct _tagARRAYOBJECT
{
    int* iArray;
    int iSize;
    int iThreadID;
} ARRAYOBJECT, *PARRAYOBJECT;

DWORD WINAPI ThreadStart(LPVOID lpParameter);
void PrintArray(int* iArray, int iSize);
void MergeArrays(int* leftArray, int leftArrayLenght, int*
rightArray, int rightArrayLenght, int* mergedArray);

int _tmain(int argc, TCHAR* argv[])
{
    int iArray1[BLOCK_SIZE];
    int iArray2[BLOCK_SIZE];
    int iArray[ELEMENTS_NUMBER];
    for (int iIndex = 0; iIndex < BLOCK_SIZE; iIndex++)
    {
        iArray1[iIndex] = rand() % MAX_VALUE;
        iArray2[iIndex] = rand() % MAX_VALUE;
    }

    HANDLE hThreads[THREADS_NUMBER];
    ARRAYOBJECT pObject1 = { &(iArray1[0]), BLOCK_SIZE, 0 };
    hThreads[0] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
ThreadStart, (LPVOID)&pObject1, 0, NULL);

    ARRAYOBJECT pObject2 = { &(iArray2[0]), BLOCK_SIZE, 1 };
    hThreads[1] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
ThreadStart, (LPVOID)&pObject2, 0, NULL);

    cout << "Waiting execution..." << endl;
    WaitForMultipleObjects(THREADS_NUMBER, hThreads, TRUE,
INFINITE);
```

```

    MergeArrays(&iArray1[0], BLOCK_SIZE, &iArray2[0], BLOCK_SIZE,
&iArray[0]);
    PrintArray(iArray, ELEMENTS_NUMBER);

    CloseHandle(hThreads[0]);
    CloseHandle(hThreads[1]);

    cout << "Array sorted..." << endl;
    return 0;
}

DWORD WINAPI ThreadStart(LPVOID lpParameter)
{
    PARRAYOBJECT pObject = (PARRAYOBJECT)lpParameter;
    int iTmp = 0;
    for (int iIndex = 0; iIndex < pObject->iSize; iIndex++)
    {
        for (int iEndIndex = pObject->iSize - 1; iEndIndex > iIndex;
iEndIndex--)
        {
            if (pObject->iArray[iEndIndex] < pObject->iArray[iIndex])
            {
                iTmp = pObject->iArray[iEndIndex];
                pObject->iArray[iEndIndex] = pObject->iArray[iIndex];
                pObject->iArray[iIndex] = iTmp;
            }
        }
    }
    return 0;
}

void PrintArray(int* iArray, int iSize)
{
    for (int iIndex = 0; iIndex < iSize; iIndex++)
    {
        cout << " " << iArray[iIndex];
    }
    cout << endl;
}

void MergeArrays(int* leftArray, int leftArrayLenght, int*
rightArray, int rightArrayLenght, int* mergedArray)
{
    int i = 0;
    int j = 0;

```

```
int k = 0;
while (i < leftArrayLenght && j < rightArrayLenght)
{
    if (leftArray[i] < rightArray[j])
    {
        mergedArray[k] = leftArray[i];
        i++;
    }
    else
    {
        mergedArray[k] = rightArray[j];
        j++;
    }
    k++;
}
if (i >= leftArrayLenght)
{
    while (j < rightArrayLenght)
    {
        mergedArray[k] = rightArray[j];
        j++;
        k++;
    }
}
if (j >= rightArrayLenght)
{
    while (i < leftArrayLenght)
    {
        mergedArray[k] = leftArray[i];
        i++;
        k++;
    }
}
}
```

How it works...

This is a very simple example that demonstrates basic thread usage. The idea was to somehow split the problem and distribute it among the threads in order to reduce the execution time by adding parallelism (divide and conquer). As we said earlier, the ability to divide the problem into smaller units (dependent or independent) makes it easier for us to create parallel logic in implementation, while at the same time, using system resources in parallel makes our application faster and more optimized.

There's more...

As we said earlier, every application has a main thread. Additional threads are created using the `CreateThread` Win32 API. A prototype of the same is as follows:

```
HANDLE CreateThread( LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize, LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter, DWORD dwFlags, LPDWORD lpThreadId );
```

It is important to set a thread start address or thread routine passed by the `lpStartAddress` parameter and a value for the provided routine passed by the `lpParameter` parameter. It is a predefined routine (function) pointer and is shown in the following code:

```
typedef DWORD ( WINAPI *PTHREAD_START_ROUTINE )
( LPVOID lpThreadParameter );
```

Our `ThreadStart` method matches the specified prototype, and this is where the thread starts its execution. The fourth parameter of the `CreateThread` API is passed to the thread routine. As it is a pointer, if you need to pass more than one parameter, you could create a structure or class and pass the object address. You can find more on the `CreateThread` API on MSDN (<http://msdn.microsoft.com/en-us/library/windows/desktop/ms682453%28v=vs.85%29.aspx>).

Thread usage

A majority of today's applications are using some database. In many cases, such applications usually ran from different PCs, with simultaneous read/write operations. The following example will demonstrate threads using the MySQL database.

Getting ready

For our next example, you will need to install the MySQL C connector. A detailed explanation can be found in the *Appendix*. After the successful installation of the MySQL C connector, start Visual Studio.

How to do it...

1. Create a new C++ Console application. Name it `MultithreadedDBTest`.
2. Open **Solution explorer** and add a new header file. Name it `CMySQL.h`. Open `CMySQL.h` and paste the following code:

```
#include "stdafx.h"
#include <stdio.h>
```



```
#include <stdlib.h>
#include <mysql.h>

class CMySQL
{
public:
    static CMySQL* CreateInstance(char* szHostName, char*
szDatabase, char* szUserId, char* szPassword);
    static void ReleaseInstance();
    bool ConnectInstance();
    bool DisconnectInstance();
    bool ReadData(char* szQuery, char* szResult, size_t
uBufferLenght);
    bool WriteData(char* szQuery, char* szResult, size_t
uBufferLenght);
private:
    CMySQL(char* szHostName, char* szDatabase, char* szUserId, char*
szPassword);
    ~CMySQL();
    char* szHostName;
    char* szDatabase;
    char* szUserId;
    char* szPassword;
    MYSQL* mysqlConnection;
    static CMySQL* mySqlInstance;
};
```

3. Open **Solution explorer** and add a new CMySQL.cpp file. Open CMySQL.cpp and paste the following code:

```
#include "stdafx.h"
#include "CMySQL.h"

CMySQL* CMySQL::mySqlInstance = NULL;

CMySQL* CMySQL::CreateInstance(char* szHostName, char* szDatabase,
char* szUserId, char* szPassword)
{
    if (mySqlInstance)
    {
        return mySqlInstance;
    }
    return new CMySQL(szHostName, szDatabase, szUserId, szPassword);
}
```

```
void CMySQL::ReleaseInstance()
{
    if (mySqlInstance)
    {
        delete mySqlInstance;
    }
}

CMySQL::CMySQL(char* szHostName, char* szDatabase, char* szUserId,
char* szPassword)
{
    size_t length = 0;

    this->szHostName = new char[length = strlen(szHostName) + 1];
    strcpy_s(this->szHostName, length, szHostName);

    this->szDatabase = new char[length = strlen(szDatabase) + 1];
    strcpy_s(this->szDatabase, length, szDatabase);

    this->szUserId = new char[length = strlen(szUserId) + 1];
    strcpy_s(this->szUserId, length, szUserId);

    this->szPassword = new char[length = strlen(szPassword) + 1];
    strcpy_s(this->szPassword, length, szPassword);
}

CMySQL::~CMySQL()
{
    delete szHostName;
    delete szDatabase;
    delete szUserId;
    delete szPassword;
}

bool CMySQL::ConnectInstance()
{
    MYSQL* mysqlLink = NULL;

    try
    {
        mysqlConnection = mysql_init(NULL);
        mysqlLink = mysql_real_connect(mysqlConnection, szHostName,
szUserId, szPassword, szDatabase, 3306, NULL, 0);
    }
}
```

```
        catch (...)
        {
            mysqlConnection = 0;
            return false;
        }
        return mysqlLink ? true : false;
    }

bool CMySQL::DisconnectInstance()
{
    try
    {
        mysql_close(mysqlConnection);
        return true;
    }
    catch (...)
    {
        return false;
    }
}

bool CMySQL::ReadData(char* szQuery, char* szResult, size_t
uBufferLenght)
{
    int mysqlStatus = 0;
    MYSQL_RES* mysqlResult = NULL;
    MYSQL_ROW mysqlRow = NULL;
    my_ulonglong numRows = 0;
    unsigned numFields = 0;

    try
    {
        mysqlStatus = mysql_query(mysqlConnection, szQuery);
        if (mysqlStatus)
        {
            return false;
        }
        else
        {
            mysqlResult = mysql_store_result(mysqlConnection);
        }
    }
```

```

        if (mysqlResult)
        {
            numRows = mysql_num_rows(mysqlResult);
            numFields = mysql_num_fields(mysqlResult);
        }

        mysqlRow = mysql_fetch_row(mysqlResult);
        if (mysqlRow)
        {
            if (!mysqlRow[0])
            {
                mysql_free_result(mysqlResult);
                return false;
            }
        }
        else
        {
            mysql_free_result(mysqlResult);
            return false;
        }

        size_t szResultLength = strlen(mysqlRow[0]) + 1;
        strcpy_s(szResult, szResultLength > uBufferLenght ?
uBufferLenght : szResultLength, mysqlRow[0]);
        if (mysqlResult)
        {
            mysql_free_result(mysqlResult);
            mysqlResult = NULL;
        }
    }
    catch (...)
    {
        return false;
    }
    return true;
}

bool CMySQL::WriteData(char* szQuery, char* szResult, size_t
uBufferLenght)
{
    try
    {

```

```

        int mysqlStatus = mysql_query(mysqlConnection, szQuery);
        if (mysqlStatus)
        {
            size_t szResultLength = strlen("Failed!") + 1;
            strcpy_s(szResult, szResultLength > uBufferLength ?
uBufferLength : szResultLength, "Failed!");
            return false;
        }
    }
    catch (...)
    {
        size_t szResultLength = strlen("Exception!") + 1;
        strcpy_s(szResult, szResultLength > uBufferLength ?
uBufferLength : szResultLength, "Exception!");
        return false;
    }

    size_t szResultLength = strlen("Success") + 1;
    strcpy_s(szResult, szResultLength > uBufferLength ?
uBufferLength : szResultLength, "Success");
    return true;
}

```

4. Open MultithreadedDBTest.cpp and paste the following code:

```

#include "stdafx.h"
#include "CMySQL.h"

#define BLOCK_SIZE 4096
#define THREADS_NUMBER 3

typedef struct
{
    char szQuery[BLOCK_SIZE];
    char szResult[BLOCK_SIZE];
    bool bIsRead;
} QUERYDATA, *PQUERYDATA;

CRITICAL_SECTION cs;
CMySQL* mySqlInstance = NULL;

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    PQUERYDATA pQueryData = (PQUERYDATA)lpParameter;

```

```

EnterCriticalSection(&cs);
if (mySqlInstance->ConnectInstance())
{
    if (pQueryData->bIsRead)
    {
        memset(pQueryData->szResult, 0, BLOCK_SIZE-1);
        mySqlInstance->ReadData(pQueryData->szQuery,
            pQueryData->szResult, BLOCK_SIZE - 1);
    }
    else
    {
        mySqlInstance->WriteData(pQueryData->szQuery,
            pQueryData->szResult, BLOCK_SIZE - 1);
    }
    mySqlInstance->DisconnectInstance();
}
LeaveCriticalSection(&cs);
return 0L;
}

int main()
{
    InitializeCriticalSection(&cs);
    mySqlInstance = CMySQL::CreateInstance(
        "mysql.services.expert.its.me", "expertit_97900S",
        "expertit_9790", "$dbpass_1342#");
    if (mySqlInstance)
    {
        HANDLE hThreads[THREADS_NUMBER];
        QUERYDATA queryData[THREADS_NUMBER] =
        {
            { "select address from clients where id = 3;",
              "", true },
            { "update clients set name='Merrill & Lynch' where id=2;",
              "", false },
            { "select name from clients where id = 2;",
              "", true }
        };
        for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
        {
            hThreads[iIndex] = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)StartAddress,
                &queryData[iIndex], 0, 0);
        }
    }
}

```

```
        WaitForMultipleObjects(THREADS_NUMBER, hThreads, TRUE,
INFINITE);
        for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
        {
            printf_s("%s\n", queryData[iIndex].szResult);
        }
        CMySQL::ReleaseInstance();
    }

    DeleteCriticalSection(&cs);
    return system("pause");
}
```

How it works...

The previous example demonstrates thread synchronization from the application that operates over the MySQL database. The example shows only one way to achieve this, while there was another way to use thread synchronization from **Database Management System (DBMS)** itself, using *table locks*. The important thing here was disabling the thread to read or write in the same time as another thread was performing either a *read* or *write* operation. We assumed that two out of our three threads have to read, while the third one has to write certain data. You could assume that after reading, the thread has to signal the main application or some other handler with the data read, to continue processing. The same thing stands for the thread that has to write data—it would obtain data from some handler and then perform the necessary operation. For thread synchronization, we were using a critical section object. A detailed explanation of a critical section object will be presented in *Chapter 3, Managing Threads*.

There's more...

As we said earlier, there was another, easier way to synchronize threads while they operate over the MySQL database. With today's modern MySQL DBMS, we can achieve this using table locking. Only to give a hint to the user, we will demonstrate one way of performing query from two threads but at the same time to retain synchronized read and write operations, without interference from one another:

1. Read thread:

```
lock table TABLE_NAME read;
select something from TABLE_NAME;
unlock tables;
```

2. Write thread:

```
lock table TABLE_NAME write;
insert into TABLE_NAME values( ... );
```

```
update TABLE_NAME set something;
unlock tables;
```

3. Both operations from a single thread—others must wait:

```
lock table TABLE_NAME TABLE_ALIAS read, TABLE_NAME write;
select something from TABLE_NAME as TABLE_ALIAS;
insert into TABLE_NAME values ...
unlock tables;
```

Pay attention that there are many MySQL engine models (InnoDB, MyISAM, Memory, and so on), and table locking and locks granularity varies between engines.

Implementing threads in user space

There are two ways to implement a thread's package: in the user space and in the kernel. The choice is moderately controversial and a hybrid implementation, where it is also possible to use a combination of user and kernel thread.

We will now describe these methods, along with their advantages and disadvantages. The first method is to put the threads package entirely in the user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first and most obvious advantage is that a user-level threads package can be implemented on an operating system that does not support threads.

Traditional operating systems used to fall into this category, and even now, some still do. With this approach, threads are implemented by a library. All of these implementations have the same general structure. The threads run on top of a runtime system, which is a collection of procedures that manage threads. We have seen a few of these already (`CreateThread`, `TerminateThread`, and others), but usually, there are more.

The following example will demonstrate thread usage in user space. We want to copy large files. We don't want to read it entirely first and write it after that. No, it would be more optimized to read it piece by piece, at the same time writing whatever piece was previously read. This scenario demonstrates the **Producer-Consumer** problem, mentioned earlier in the *Interprocess Communication* recipe.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

1. Create a new empty Win32 application project and name it ConcurrentFileCopy.
2. Open **Solution explorer** and add a new header file. Name it ConcurrentFileCopy.h. Open ConcurrentFileCopy.h and paste the following code:

```
#pragma once
```

```
#include <windows.h>
#include <commctrl.h>
#include <memory.h>
#include <tchar.h>
#include <math.h>
```

```
#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*'\\"" )
```

```
ATOM RegisterWndClass(HINSTANCE hInstance);
HWND InitializeInstance(HINSTANCE hInstance, int nCmdShow, HWND&
hWndPB);
LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);
DWORD WINAPI ReadRoutine(LPVOID lpParameter);
DWORD WINAPI WriteRoutine(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPTSTR szFileName, DWORD
dwFileOperation);
DWORD GetBlockSize(DWORD dwFileSize);
```

```
#define BUTTON_CLOSE 100
#define FILE_SAVE 0x0001
#define FILE_OPEN 0x0002
#define MUTEX_NAME _T("__RW_Mutex__")
```

```
typedef struct _tagCOPYDETAILS
{
    HINSTANCE hInstance;
    HWND hWndPB;
    LPTSTR szReadFileName;
    LPTSTR szWriteFileName;
} COPYDETAILS, *PCOPYDETAILS;
```

3. Now, open **Solution explorer** and add a new source file. Name it `ConcurrentFileCopy.cpp`. Open `ConcurrentFileCopy.cpp` and paste the following code:

```
#include "ConcurrentFileCopy.h"

TCHAR* szTitle = _T("Concurrent file copy");
TCHAR* szWindowClass = _T("__CFC_WND_CLASS__");

DWORD dwReadBytes = 0;
DWORD dwWriteBytes = 0;

DWORD dwBlockSize = 0;
DWORD dwFileSize = 0;

HLOCAL pMemory = NULL;

int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrev, LPTSTR
szCmdLine, int iCmdShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCmdLine);

    RegisterWndClass(hInstance);

    HWND hWnd = NULL;
    HWND hWndPB = NULL;
    if (!(hWnd = InitializeInstance(hInstance, iCmdShow, hWndPB)))
    {
        return 1;
    }

    MSG msg = { 0 };
    TCHAR szReadFile[MAX_PATH];
    TCHAR szWriteFile[MAX_PATH];

    if (FileDialog(hWnd, szReadFile, FILE_OPEN) && FileDialog(hWnd,
szWriteFile, FILE_SAVE))
    {
        COPYDETAILS copyDetails = { hInstance, hWndPB, szReadFile,
szWriteFile };

        HANDLE hMutex = CreateMutex(NULL, FALSE, MUTEX_NAME);
        HANDLE hReadThread = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)ReadRoutine, &copyDetails, 0, NULL);
```

```
        while (GetMessage(&msg, NULL, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        CloseHandle(hReadThread);
        CloseHandle(hMutex);
    }
    else
    {
        MessageBox(hWnd, _T("Cannot open file!"),
        _T("Error!"), MB_OK);
    }

    LocalFree(pMemory);
    UnregisterClass(szWindowClass, hInstance);
    return (int)msg.wParam;
}

ATOM RegisterWndClass(HINSTANCE hInstance)
{
    WNDCLASSEX wndEx;

    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WndProc;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hInstance;
    wndEx.hIcon = LoadIcon(hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

    return RegisterClassEx(&wndEx);
}
```

```

HWND InitializeInstance(HINSTANCE hInstance, int iCmdShow, HWND&
hWndPB)
{
    HWND hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPED
| WS_CAPTION | WS_SYSMENU | WS_MINIMIZEBOX, 200, 200, 440, 290,
NULL, NULL, hInstance, NULL);
    RECT rcClient = { 0 };
    int cyVScroll = 0;

    if (!hWnd)
    {
        return NULL;
    }

    HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE, FALSE,
FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
DEFAULT_QUALITY, DEFAULT_PITCH | FF_MODERN,
_T("Microsoft Sans Serif"));
    HWND hButton = CreateWindow(_T("BUTTON"), _T("Close"), WS_CHILD
| WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 310, 200, 100, 25,
hWnd, (HMENU)BUTTON_CLOSE, hInstance, NULL);
    SendMessage(hButton, WM_SETFONT, (WPARAM)hFont, TRUE);

    GetClientRect(hWnd, &rcClient);
    cyVScroll = GetSystemMetrics(SM_CYVSCROLL);

    hWndPB = CreateWindow(PROGRESS_CLASS, (LPTSTR)NULL, WS_CHILD |
WS_VISIBLE, rcClient.left, rcClient.bottom - cyVScroll,
rcClient.right, cyVScroll, hWnd, (HMENU)0, hInstance, NULL);
    SendMessage(hWndPB, PBM_SETSTEP, (WPARAM)1, 0);

    ShowWindow(hWnd, iCmdShow);
    UpdateWindow(hWnd);

    return hWnd;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {

```

```

        switch (LOWORD(wParam))
        {
            case BUTTON_CLOSE:
            {
                DestroyWindow(hWnd);
                break;
            }
        }
        break;
    }
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        break;
    }
    default:
    {
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}
return 0;
}

DWORD WINAPI ReadRoutine(LPVOID lpParameter)
{
    PCOPYDETAILS pCopyDetails = (PCOPYDETAILS)lpParameter;
    HANDLE hFile = CreateFile(pCopyDetails->szReadFileName,
        GENERIC_READ, FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == (HANDLE)INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }
    dwFileSize = GetFileSize(hFile, NULL);
    dwBlockSize = GetBlockSize(dwFileSize);
    HANDLE hWriteThread = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)WriteRoutine, pCopyDetails,
        0, NULL);
    size_t uBufferLength = (size_t)ceil((double)
        dwFileSize / (double)dwBlockSize);
    SendMessage(pCopyDetails->hWndPB, PBM_SETRANGE, 0,
        MAKELPARAM(0, uBufferLength));
}

```

```

pMemory = LocalAlloc(LPTR, dwFileSize);
void* pBuffer = LocalAlloc(LPTR, dwBlockSize);

int iOffset = 0;
DWORD dwBytesRed = 0;

do
{
    ReadFile(hFile, pBuffer, dwBlockSize, &dwBytesRed, NULL);
    if (!dwBytesRed)
    {
        break;
    }

    HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE,
MUTEX_NAME);
    WaitForSingleObject(hMutex, INFINITE);

    memcpy((char*)pMemory + iOffset, pBuffer, dwBytesRed);
    dwReadBytes += dwBytesRed;

    ReleaseMutex(hMutex);

    iOffset += (int)dwBlockSize;
} while (true);

LocalFree(pBuffer);
CloseHandle(hFile);
CloseHandle(hWriteThread);
return 0;
}

DWORD WINAPI WriteRoutine(LPVOID lpParameter)
{
    PCOPYDETAILS pCopyDetails = (PCOPYDETAILS)lpParameter;
    HANDLE hFile = CreateFile(pCopyDetails->szWriteFileName,
GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
    if (hFile == (HANDLE)INVALID_HANDLE_VALUE)
    {
        return FALSE;
    }

    DWORD dwBytesWritten = 0;
    int iOffset = 0;

```

```
do
{
    int iRemainingBytes = (int)dwFileSize - iOffset;
    if (iRemainingBytes <= 0)
    {
        break;
    }

    Sleep(10);
    if (dwWriteBytes < dwReadBytes)
    {
        DWORD dwBytesToWrite = dwBlockSize;
        if (!(dwFileSize / dwBlockSize))
        {
            dwBytesToWrite = (DWORD)iRemainingBytes;
        }

        HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE,
MUTEX_NAME);
        WaitForSingleObject(hMutex, INFINITE);

        WriteFile(hFile, (char*)pMemory + iOffset, dwBytesToWrite,
&dwBytesWritten, NULL);
        dwWriteBytes += dwBytesWritten;

        ReleaseMutex(hMutex);

        SendMessage(pCopyDetails->hWndPB, PBM_STEPIT, 0, 0);
        iOffset += (int)dwBlockSize;
    }
} while (true);

CloseHandle(hFile);
return 0;
}

BOOL FileDialog(HWND hWnd, LPTSTR szFileName, DWORD
dwFileOperation)
{
#ifdef _UNICODE
    OPENFILENAMEW ofn;
#else
    OPENFILENAMEA ofn;
```

```

#endif
    TCHAR szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));
    ofn.lStructSize = sizeof(ofn);
    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = _T("All\\0*.*\\0Text\\0*.TXT\\0");
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = dwFileOperation == FILE_OPEN ? OFN_PATHMUSTEXIST |
    OFN_FILEMUSTEXIST : OFN_SHOWHELP | OFN_OVERWRITEPROMPT;

    if (dwFileOperation == FILE_OPEN)
    {
        if (GetOpenFileName(&ofn) == TRUE)
        {
            _tcscpy_s(szFileName, MAX_PATH - 1, szFile);
            return TRUE;
        }
    }
    else
    {
        if (GetSaveFileName(&ofn) == TRUE)
        {
            _tcscpy_s(szFileName, MAX_PATH - 1, szFile);
            return TRUE;
        }
    }
    return FALSE;
}

DWORD GetBlockSize(DWORD dwFileSize)
{
    return dwFileSize > 4096 ? 4096 : 512;
}

```


How it works...

We created a UI pretty much similar to the *Philosophers Dinner* example. Routines `MyRegisterClass`, `InitInstance` and `WndProc` are pretty much the same. We added `FileDialog` to ask the user for the file path to read from and the file path to write to. We started two threads in order to read and write, respectively.

Scheduling in operating systems is very complex and a large challenge. As a result of either the scheduling algorithm or hardware interrupts, we could never know when the thread will be scheduled for execution on the CPU next! This means that the thread that writes could be scheduled for execution before the thread that reads. This situation would bring an exception, because the thread that needs to write has nothing to write.

That's why we added an extra `if` condition in the write operation, as shown in the following code:

```
if ( dwBytesWritten < dwBytesRead )
{
    WriteFile(hFile, pCharTmp, sizeof(TCHAR) * BLOCK_SIZE,
        &dwFileSize, NULL);
    dwBytesWritten += dwFileSize;
    SendMessage( hProgress, PBM_STEPIT, 0, 0 );
}
```

After a thread obtains a mutex, it can perform a write operation. However, there is a possibility that it was scheduled before the thread that is reading has, and the buffer might be empty. Every time the read thread gets something, it adds the number of bytes read to a `dwBytesRead` variable, and if the thread that needs to write has written less bytes than bytes read, it can write. If not, it will skip writing in this cycle and simply release mutex for the other thread.

There's more...

The *producer-consumer* problem is also known as the bounded-buffer problem. Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. It is also possible to generalize the problem to have m producers and n consumers, but we will only consider the case of one producer and one consumer because this assumption simplifies the solution. Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, and to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sustains that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up. This approach sounds simple enough, but it leads to the same kind of race conditions as we saw earlier with the spooler directory. It is left to the reader to try to solve similar situations using the knowledge obtained here.

Implementing threads in the kernel

The entire kernel is a process with a lot of system (kernel) threads running in its context. The kernel has a thread table that keeps track of all the threads in the system.

The kernel maintains the traditional process table to keep track of processes. Function calls that can block a thread are executed as system calls, at much greater cost than the executing system procedure. When a thread blocks, the kernel must run another thread. When a thread is destroyed, it is marked as not runnable—but its kernel data structures are not otherwise affected. Afterwards, when a new thread needs to be created, an old thread is reactivated, recycling resources for usage. Thread recycling is also possible for user-level threads, but as the thread management overhead is much smaller, there is less incentive to do this.

Getting ready

For our next example, you will need to install WinDDK—Driver Development Kit. The detailed explanation can be found in the *Appendix*. After the successful installation of WinDDK, start Visual Studio.

How to do it...

1. Create a new empty Win32 application project. Name it `KernelThread`.
2. Open **Solution explorer** and add a new header file. Name it `ThreadApp.h`. Open `ThreadApp.h` and paste the following code:

```
#include <windows.h>
#include <tchar.h>

#define DRIVER_NAME      TEXT( "TestDriver.sys" )
#define DRIVER_SERVICE_NAME TEXT( "TestDriver" )
#define Message(n) MessageBox(0, TEXT(n), \
    TEXT("Test Driver Info"), 0)

BOOL StartDriver(LPTSTR szCurrentDriver);
BOOL StopDriver(void);
```

3. Now, open **Solution explorer** and add a new source file. Name it `ThreadApp.cpp`. Open `ThreadApp.cpp` and paste the following code:

```
#include "ThreadApp.h"

int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR szCommandLine, int iCmdShow)
{
    StartDriver(DRIVER_NAME);
}
```

```

    ShellAbout(0, DRIVER_SERVICE_NAME, TEXT(""), NULL);
    StopDriver();
    return 0;
}
BOOL StartDriver(LPTSTR szCurrentDriver)
{
    HANDLE      hFile = 0;
    DWORD       dwReturn = 0;
    SC_HANDLE    hSCManager = { 0 };
    SC_HANDLE    hService = { 0 };
    SERVICE_STATUS ServiceStatus = { 0 };
    TCHAR        szDriverPath[MAX_PATH] = { 0 };
    GetSystemDirectory(szDriverPath, MAX_PATH);
    TCHAR szDriver[MAX_PATH + 1];
#ifdef _UNICODE
    wsprintf(szDriver, L"\\drivers\\%ws", DRIVER_NAME);
#else
    sprintf(szDriver, "\\drivers\\%s", DRIVER_NAME);
#endif
    _tcscat_s(szDriverPath, (_tcslen(szDriver) + 1) * sizeof(TCHAR),
szDriver);
    BOOL bSuccess = CopyFile(szCurrentDriver, szDriverPath, FALSE);
    if (bSuccess == FALSE)
    {
        Message("copy driver failed");
        return bSuccess;
    }
    hSCManager = OpenSCManager(NULL, NULL,
SC_MANAGER_CREATE_SERVICE);
    if (hSCManager == 0)
    {
        Message("open sc manager failed!");
        return FALSE;
    }
    hService = CreateService(hSCManager, DRIVER_SERVICE_NAME,
DRIVER_SERVICE_NAME, SERVICE_START | DELETE | SERVICE_STOP,
SERVICE_KERNEL_DRIVER, SERVICE_DEMAND_START, SERVICE_ERROR_IGNORE,
szDriverPath, NULL, NULL, NULL, NULL, NULL);
    if (hService == 0)
    {
        hService = OpenService(hSCManager, DRIVER_SERVICE_NAME,
SERVICE_START | DELETE | SERVICE_STOP);
        Message("create service failed!");
    }
    if (hService == 0)
    {

```

```

        Message("open service failed!");
        return FALSE;
    }
    BOOL startSuccess = StartService(hService, 0, NULL);
    if (startSuccess == FALSE)
    {
        Message("start service failed!");
        return startSuccess;
    }
    CloseHandle(hFile);
    return TRUE;
}
BOOL StopDriver(void)
{
    SC_HANDLE    hSCManager = { 0 };
    SC_HANDLE    hService = { 0 };
    SERVICE_STATUS ServiceStatus = { 0 };
    TCHAR        szDriverPath[MAX_PATH] = { 0 };
    GetSystemDirectory(szDriverPath, MAX_PATH);
    TCHAR szDriver[MAX_PATH + 1];
#ifdef _UNICODE
    wsprintf(szDriver, L"\\drivers\\%ws", DRIVER_NAME);
#else
    sprintf(szDriver, "\\drivers\\%s", DRIVER_NAME);
#endif
    _tcscat_s(szDriverPath, (_tcslen(szDriver) + 1) * sizeof(TCHAR),
szDriver);
    hSCManager = OpenSCManager(NULL, NULL,
SC_MANAGER_CREATE_SERVICE);
    if (hSCManager == 0)
    {
        return FALSE;
    }
    hService = OpenService(hSCManager, DRIVER_SERVICE_NAME,
SERVICE_START | DELETE | SERVICE_STOP);
    if (hService)
    {
        ControlService(hService, SERVICE_CONTROL_STOP,
&ServiceStatus);
        DeleteService(hService);
        CloseServiceHandle(hService);
        BOOL ifSuccess = DeleteFile(szDriverPath);
        return TRUE;
    }
    return FALSE;
}

```

4. Now, open **Solution explorer** and create a new empty Win32 Console project. Name it DriverApp.
5. Add a new header file. Name it DriverApp.h. Open DriverApp.h and paste the following code:

```
#include <ntddk.h>
```

```
DRIVER_INITIALIZE DriverEntry;  
DRIVER_UNLOAD OnUnload;
```

6. Now, open **Solution explorer**, and under the DriverApp project, add a new source file. Name it DriverApp.cpp. Open DriverApp.cpp and paste the following code:

```
#include "DriverApp.h"
```

```
VOID ThreadStart(PVOID lpStartContext)  
{  
    PKEVENT pEvent = (PKEVENT)lpStartContext;  
    DbgPrint("Hello! I am kernel thread. My ID is %u. Regards..",  
(ULONG)PsGetCurrentThreadId());  
    KeSetEvent(pEvent, 0, 0);  
    PsTerminateSystemThread(STATUS_SUCCESS);  
}  
NTSTATUS DriverEntry(PDRIVER_OBJECT theDriverObject,  
PUNICODE_STRING theRegistryPath)  
{  
    HANDLE hThread = NULL;  
    NTSTATUS ntStatus = 0;  
    OBJECT_ATTRIBUTES ThreadAttributes;  
    KEVENT kEvent = { 0 };  
    PETHREAD pThread = 0;  
    theDriverObject->DriverUnload = OnUnload;  
    DbgPrint("Entering KERNEL mode..");  
    InitializeObjectAttributes(&ThreadAttributes, NULL,  
OBJ_KERNEL_HANDLE, NULL, NULL);  
    __try  
    {  
        KeInitializeEvent(&kEvent, SynchronizationEvent, 0);  
        ntStatus = PsCreateSystemThread(&hThread, GENERIC_ALL,  
&ThreadAttributes, NULL, NULL, (PKSTART_ROUTINE) &ThreadStart,  
&kEvent);  
        if (NT_SUCCESS(ntStatus))  
        {  
            KeWaitForSingleObject(&kEvent, Executive, KernelMode, FALSE,  
NULL);  
        }  
    }  
    __except(EXCEPTION_EXECUTE_HANDLER)  
    {  
        ntStatus = GetExceptionCode();  
    }  
    return ntStatus;  
}
```

```

        ZwClose(hThread);
    }
    else
    {
        DbgPrint("Could not create system thread!");
    }
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    DbgPrint("Error while creating system thread!");
}
return STATUS_SUCCESS;
}

VOID OnUnload(PDRIVER_OBJECT DriverObject)
{
    DbgPrint("Leaving KERNEL mode..");
}

```

How it works...

First, we created a Win32 application but only for demonstration purposes. We won't have a UI, not even a message loop. We simply want to load a driver into the kernel; then, we will show the *ShellAbout* dialog only to give the user time to read the *DbgView* output (detailed explanation about *DbgView* can be found in the *Appendix*), and after the user closes the *ShellAbout* dialog, the driver is unloaded and the application ends.

The Win32 application that we created can only load and unload the driver, so no further explanation will occur. Now, let's review the *DriverApp* project. After the proper setting of the Visual Studio project for driver compilation (detailed explanation about compilation settings for Visual Studio can be found in the *Appendix*), we have declared the following two main routines that every driver must have in the *DriverApp.h* header file:

```

DRIVER_INITIALIZE DriverEntry;
DRIVER_UNLOAD     OnUnload;

```

These two routines are the driver entry point and driver unload routine. We will use driver entry point to initialize a thread object and start the kernel thread. A newly created thread will only write us a message that will display its unique identifier and will return immediately. Further, the kernel exploration and development is not part of this book. We only wanted to give a taste of the kernel thread. As the driver is compiled as */TC2*, we had to make sure that all variables are declared before the first command. This is shown in the following code:

```

HANDLE hThread = NULL;
NTSTATUS ntStatus = 0;

```

```
OBJECT_ATTRIBUTES ThreadAttributes;  
KEVENT kEvent = { 0 };  
PETHREAD pThread = 0;
```

Then, we had to set the unload routine:

```
theDriverObject->DriverUnload = OnUnload;
```

Now, before we can create the kernel thread, we need to initialize the `ThreadAttribute` object by calling the `InitializeObjectAttributes` routine first:

```
InitializeObjectAttributes(&ThreadAttributes, NULL,  
    OBJ_KERNEL_HANDLE, NULL, NULL);
```

The kernel development must be performed with extreme caution, because even the smallest of mistakes can lead to **Blue Screen Of Death (BSOD)** or a machine crash. That's why we are using the `__try - __except` block, which is slightly different from `try - catch` that we are used to.

Handles created in the kernel are not the same as handles created in the user space. The handle returned by `PsCreateSystemThread` can't be used for the `KeWaitForSingleObject` routine. We need to initialize an event that will be signaled (events will be explained in detail in *Chapter 3, Managing Threads*) from the thread itself when `KeWaitForSingleObject` returns. Every call to `PsCreateSystemThread` must have its matching call to the `ZwClose` routine. A call to `ZwClose` closes the kernel handle and prevents memory leaks.

Finally, we need to implement `PKSTART_ROUTINE` or a thread start address where instructions for the thread execution will start. An example of this is as follows:

```
VOID (__stdcall* KSTART_ROUTINE)( PVOID StartContext );
```

We have passed a pointer to `KEVENT` via the last parameter of `PsCreateSystemThread`. Now, using `DbgPrint`, we will simply write a message to the user with the corresponding thread ID. After that, we will set an event so that the corresponding `KeWaitForSingleObject` call can return and safely exit the driver. Make sure that `PsTerminateSystemThread` does not return. The kernel will clean the thread object on unloading the driver.

There's more...

While the kernel threads solve some problems, they do not solve all of them. For example, what happens when a multithreaded process creates another multithreaded process? Does the new process have as many threads as the old one did, or does it have just one? In many cases, the best choice depends on what the process is planning to do next. If it is going to start a new program, probably one thread is the correct choice, but if it continues to execute, reproducing all the threads is probably the right thing to do.

3

Managing Threads

In this chapter we will cover the following topics:

- ▶ Processes versus threads
- ▶ Permissive versus preemptive multitasking
- ▶ Explaining the Windows Thread object
- ▶ Basic thread management
- ▶ Implementing threads without synchronization
- ▶ Using synchronized threads
- ▶ Win32 synchronization objects and techniques

Introduction

To understand thread management better along with how to use threads, we need to understand three major thread operations: thread creation, termination, and joining to the parent thread, or simply join. We already said that every process must have at least one thread—the **main thread**. So when we refer to a thread, the context can be applied to a process as well.

Every thread can create another thread. A thread is created using **Win32 API** `CreateThread`. The following is the prototype of the `CreateThread` API:

```
HANDLE WINAPI CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,
```



```
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

By creating a separate thread, we are dividing the execution or operation that we need to perform on separate entities, which may (or may not) be executed concurrently. We intentionally mentioned *may or may not*, because we can run a separate thread asynchronously without blocking the caller (creator), or synchronously when the caller is blocked (waiting) for the calling thread to complete operation (we often say return). Each time we need to perform a concurrent task, we somehow need to split the execution in order to execute these independent operations concurrently. Well, we can achieve this by creating a separate thread.

As with any other task, threads are not meant to run forever. When a thread is created, its `StartAddress` must be provided. `StartAddress` is a pointer to the user-defined routine where the thread will start its execution. When the routine returns, the thread terminates. The `StartAddress` example is as follows:

```
DWORD WINAPI StartAddress(  
    LPVOID lpParameter  
);
```

A thread can be terminated by force, using Win32 API's `TerminateThread`. This is not the correct way to terminate a thread, and should be used only when it's absolutely necessary. This is because the operating system won't release the resources used by the thread properly.

When a thread is terminated, the resources held by the thread are released and the thread's exit code is set. The thread object is signaled due to its termination, and the process is terminated if it was the only thread in the process.

By forcefully terminating the thread, we may run into problems such as not freeing the memory stack, or not cleaning up the resources, and not informing the attached DLLs. The handles held by the thread are also not closed.

There are many situations when threads need to cooperate with each other. For example, if thread A needs some output data from thread B (user input or similar), we often say that thread A needs to wait for thread B. When thread A needs to wait for thread B, we say that thread B needs to join thread A. This term is used when one execution flow can't continue until another completes.

Processes versus threads

It is very important to decide properly when to use processes and when to use threads. One thing that we must be aware of is that a process must have a file loaded from a disk. We must understand that process is the complete entity which needs to be prepared and properly used. One more thing about a process is that its creation utilizes a lot of system resources, so you should use processes only when you need them.

On the other hand, threads are created much faster with less overhead, and loading from a disk does not occur. You only need to supply the `StartAddress` pointer, which is a user-defined routine (function). Do not forget that a process is still needed for stuff such as daemons, listeners (servers), and the like. However, when talking about smaller concurrent operations, threads should be your choice in most situations.

As we already used processes and threads in our previous examples, the following example will demonstrate a big difference when creating and using both.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure using the following steps:

1. Create a new C++ Console application named `tmpThread`.
2. Open `tmpThread.cpp`.
3. Add the following code to it:

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>

using namespace std;

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    cout << "Hello. I am a very simple thread."
        << endl
        << "I am used to demonstrate thread creation."
        << endl;

    return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
    // Thread creation. Pay attention that we only need
    // to define the StartAddress routine in the same code
    // segment as our main function to start thread and
    // perform concurrent operation.
    HANDLE hThread = CreateThread(NULL, 0, StartAddress, NULL, 0,
```

```

NULL);

    // Process creation. Pay attention that first argument
    // of CreateProcess points to the file which content
    // needs to be loaded from the disk.
    STARTUPINFO startupInfo = { 0 };
    PROCESS_INFORMATION processInformation = { 0 };

#ifdef _DEBUG
    BOOL bSuccess = CreateProcess(
        TEXT("../Debug/tmpProcess.exe"), NULL, NULL,
        NULL, FALSE, 0, NULL, NULL, &startupInfo,
        &processInformation);
#else
    BOOL bSuccess = CreateProcess(
        TEXT("../Release/tmpProcess.exe"), NULL, NULL,
        NULL, FALSE, 0, NULL, NULL, &startupInfo,
        &processInformation);
#endif

    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hThread);

    return system("pause");
}

```

4. Create a new C++ Console application named tmpProcess.
5. Open **Solution Explorer**. Right-click on the tmpThread project. Select **Project Dependencies**. In the drop-down list, select tmpThread. In the space below, check the tmpProcess project.
6. Open tmpProcess.cpp.
7. Add the following code to it:

```

#include "stdafx.h"
#include <iostream>

int _tmain(int argc, _TCHAR* argv[])
{
    std::cout << "Hello. I am a very simple process."
        << std::endl
        << "I am used to demonstrate process creation."
        << std::endl;

    return 0;
}

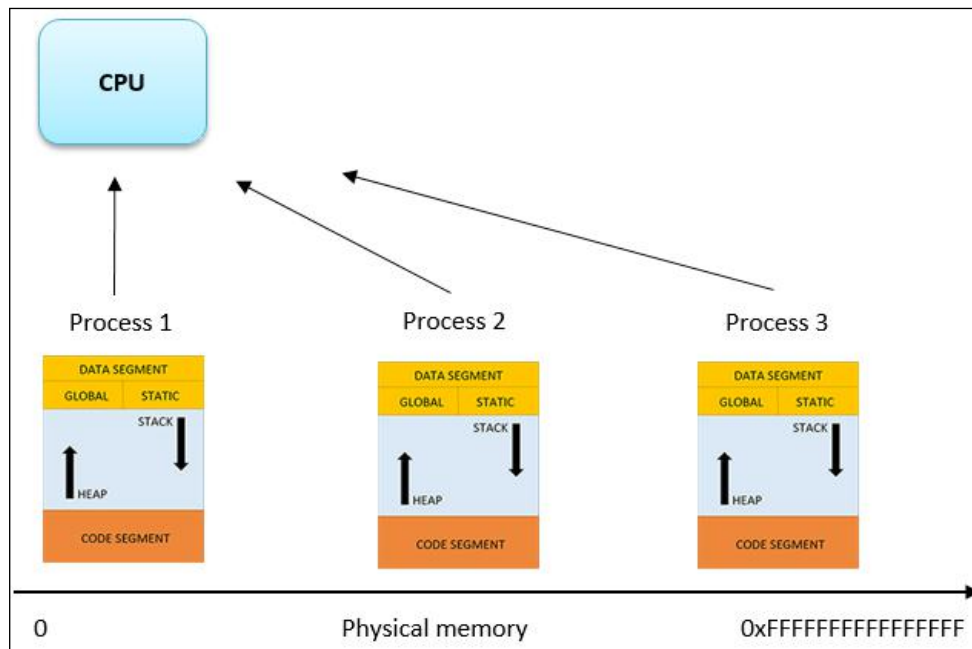
```

How it works...

The previous example is a very simple one. The idea was to show a big difference between the processes and the creation of threads. To create and use a thread, we need to implement and supply a routine within the same code segment (it can be implemented in a separate code segment or even in DLL). However, managing threads in parallel is a complex task. Creation and process usage is concurrent by definition. We could make it synchronous—by waiting for a primary thread of process—yet again it is performed on thread level not process.

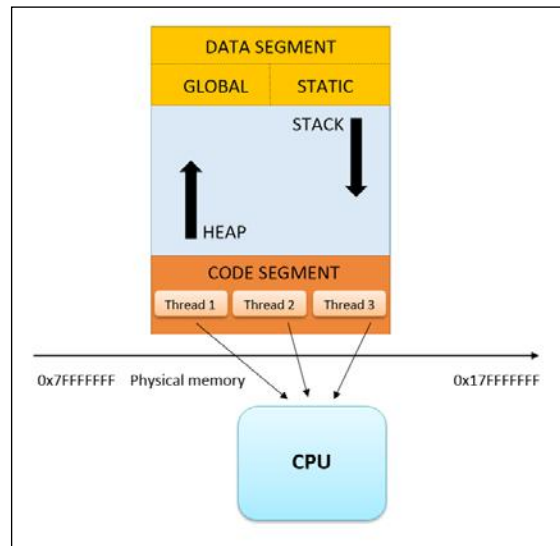
We'll use another opportunity to point out the biggest difference between processes and threads. When we are using processes, a good thing is that the process is an entire entity, along with address space, object contexts, file handles, and a lot more. However, the down side is that it's greedy—it wants all the resources for itself. Communication between processes is harder to implement as you can't use addresses (pointers) from one process to another, since addresses are private.

In the following figure, we see three processes loaded into the physical memory, which are either ready for execution or executing on the CPU:



With more work for process communication and synchronization—a thread is the most obvious choice. Compared to processes, threads are friendlier—if properly managed and synchronized, they can provide parallelism and much faster execution. The advantage for threads is that all the process resources belong to the common address space. Handles, pointers, and objects are all available along with a much smaller system overhead in transfers from RAM to the CPU.

In the following figure, we see one process with three threads loaded into the memory. Its threads are ready to execute on the CPU:



There's more...

You should consider the creation of threads each time the program performs an asynchronous operation. For example, programs with multiple windows have much to gain from creating a thread for each window. Another example would be to use a modern word processor, such as Microsoft Word, with built-in automatic grammar checking. Here, one thread (or several) is responsible for responding to the activities of the keyboard, another thread to update the text on the screen, and another one to govern the regular updating of the spare version of the work file, and so on.

Permissive versus preemptive multitasking

It is good for users to understand the current operating system's implementation regarding thread behavior, so we will give a short explanation regarding multitasking.

The earlier versions of the Windows operating system prior to Windows 2000 have worked on the permissive multitasking principle, while the newer versions of the Windows operating system are implemented on the preemptive multitasking principle.

In permissive multitasking, the operating system is relying on the fair share of resources. In other words, the applications return the control to the system, so that another process (application) could get processor time as well.

In preemptive multitasking, the operating system collectively suspends thread execution, so that the other threads get equal processor time.

Obviously, preemptive multitasking has the ability to provide a much easier and safer task execution, with the fact that it does not depend on the processes' *good will* or programmer's skill in handing over the control to the system.

Explaining the Windows Thread object

At the operating system level, a thread is an object created by the **Object Manager**. Like all the system objects, the thread has attributes (data) and methods (functions). The following table schematically shows the thread object and gives a list of its attributes and methods.

STANDARD HEADER OBJECT	
Object attributes Client ID Context Dynamic priority Base priority Processor affinity Execution time Alert status Suspension count Impersonation symbol Termination port Exit status	Object methods Create thread Open thread Query thread information Set thread information Current thread Terminate thread Get context Set context Suspend Resume Alert Test alert Register termination port

Most of the methods for threads have adequate Win32 API functions. For example, when we call `SuspendThread`, the Win32 subsystem calls the corresponding method `Suspend`. In other words, the Win32 API presents a method `Suspend` for the Win32 applications.

Windows has always protected some internal structures, such as windows or brushes of direct manipulation. Threads that are executed on a user level can't directly examine or modify the interior of the system object. It can only be done by calling the adequate Win32 API method that can do something with an object. Windows provides an identification code that identifies the object, while the programmers pass the identification code to the function that they need.

Threads have identification codes such as semaphores, events, files, and all objects in general. Only the Object Manager can change the state of an object. The `CreateThread` Win32 API routine that creates a thread returns an identification code for the new object. With the provided identification code, we can perform the following tasks:

- ▶ Raise or lower the thread priority
- ▶ Pause or resume the thread
- ▶ Terminate the thread
- ▶ Find out the thread exit code

Basic thread management

We are going to implement the thread class to give the user a sense of abstraction of the thread's implementation, which is useful for a specific problem. We will implement another helper class `CLock`, which we will use for synchronized execution if needed.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure using the following steps:

1. Create a new empty C++ Console application named `CThread`.
2. Add a new header file named `CThread.h`.
3. Add the following code to it:

```
#ifndef _CTHREAD_
#define _CTHREAD_

#include <windows.h>

#define STATE_RUNNING 0x0001
#define STATE_READY 0x0002
#define STATE_BLOCKED 0x0004
#define STATE_ALIVE 0x0008
#define STATE_ASYNC 0x0010
#define STATE_SYNC 0x0020
#define STATE_CONTINUOUS 0x0040

class CLock
```

```

{
public:
    CLock();
    ~CLock();
private:
    HANDLE hMutex;
};

class CThread
{
public:
    CThread() : hThread(0), dwThreadId(0), dwState(0),
lpUserData(0), lpParameter(0){ }
    HANDLE Create(LPVOID lpParameter,
        DWORD dwInitialState = STATE_ASYNC, DWORD dwCreationFlag = 0);
    void Join(DWORD dwWaitInterval = INFINITE);
    DWORD Suspend();
    DWORD Resume();
    void SetUserData(void* lpUserData);
    void* GetUserData() const;
    DWORD GetId() const;
    HANDLE GetHandle() const;
    DWORD GetAsyncState() const;
    DWORD GetState() const;
    void SetState(DWORD dwNewState);
    BOOL Alert();
protected:
    virtual void Run(LPVOID lpParameter = 0) = 0;
    LPVOID lpParameter;
private:
    static DWORD WINAPI StartAddress(LPVOID lpParameter);
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwState;
    void* lpUserData;
    HANDLE hEvent;
};

inline DWORD CThread::GetId() const
{
    return dwThreadId;
}

inline HANDLE CThread::GetHandle() const

```



```
{
    return hThread;
}
```

```
#endif
```

4. Add a new source file named CThread. Open the CThread.cpp file and add the following code to it:

```
#include "CThread.h"
```

```
Clock::Clock()
{
    hMutex = CreateMutex(NULL, FALSE, TEXT("_tmp_mutex_lock_"));
    WaitForSingleObject(hMutex, INFINITE);
}
```

```
Clock::~~Clock()
{
    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}
```

```
HANDLE CThread::Create(LPVOID lpParameter, DWORD dwInitialState,
DWORD dwCreationFlag)
```

```
{
    dwState |= dwInitialState;
    this->lpParameter = lpParameter;
    if (dwState & STATE_ALIVE)
    {
        return hThread;
    }
    hThread = CreateThread(NULL, 0, StartAddress, this,
dwCreationFlag, &dwThreadId);
    dwState |= STATE_ALIVE;
    if (dwState & STATE_CONTINUOUS)
    {
        hEvent = CreateEvent(NULL, TRUE, FALSE,
TEXT("__tmp_event__"));
    }
    return hThread;
}
```

```
void CThread::Join(DWORD dwWaitInterval)
{
    if (dwState & STATE_BLOCKED)
```

```
{
    return;
}
if (dwState & STATE_READY)
{
    return;
}
dwState |= STATE_READY;
WaitForSingleObject(hThread, dwWaitInterval);
dwState ^= STATE_READY;
}

DWORD CThread::Suspend()
{
    if (dwState & STATE_BLOCKED)
    {
        return DWORD(-1);
    }
    if (dwState & STATE_READY)
    {
        return DWORD(-1);
    }
    DWORD dwSuspendCount = SuspendThread(hThread);
    dwState |= STATE_BLOCKED;
    return dwSuspendCount;
}

DWORD CThread::Resume()
{
    if (dwState & STATE_RUNNING)
    {
        return DWORD(-1);
    }
    DWORD dwSuspendCount = ResumeThread(hThread);
    dwState ^= STATE_BLOCKED;
    return dwSuspendCount;
}

void CThread::SetUserData(void* lpUserData)
{
    this->lpUserData = lpUserData;
}

void* CThread::GetUserData() const
```

```
{
    return lpUserData;
}

DWORD CThread::GetAsyncState() const
{
    if (dwState & STATE_ASYNC)
    {
        return STATE_ASYNC;
    }
    return STATE_SYNC;
}

DWORD CThread::GetState() const
{
    return dwState;
}

void CThread::SetState(DWORD dwNewState)
{
    dwState = 0;
    dwState |= dwNewState;
}

BOOL CThread::Alert()
{
    return SetEvent(hEvent);
}

DWORD WINAPI CThread::StartAddress(LPVOID lpParameter)
{
    CThread* cThread = (CThread*)lpParameter;
    if (cThread->GetAsyncState() == STATE_SYNC)
    {
        if (cThread->dwState & STATE_CONTINUOUS)
        {
            DWORD dwWaitStatus = 0;
            while (TRUE)
            {
```

```
cThread->Run();

dwWaitStatus = WaitForSingleObject(cThread->hEvent, 10);

if (dwWaitStatus == WAIT_OBJECT_0)
{
    break;
}
}
return 0;
}

cThread->Run();
return 0;
}
if (cThread->dwState & STATE_CONTINUOUS)
{
    DWORD dwWaitStatus = 0;
    while (TRUE)
    {
        CLock lock;
        {
            cThread->Run();
        }

        dwWaitStatus = WaitForSingleObject(cThread->hEvent, 10);

        if (dwWaitStatus == WAIT_OBJECT_0)
        {
            break;
        }
    }
    return 0;
}
CLock lock;
{
    cThread->Run();
}
return 0;
}
```

5. Add a new source file `main.cpp`. Add the following code to it:

```
#include "CThread.h"

class Thread : public CThread
{
protected:
    virtual void Run(LPVOID lpParameter = 0);
};

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);

void Thread::Run(LPVOID lpParameter)
{
    WNDCLASSEX wndEx = { 0 };

    wndEx.cbClsExtra = 0;
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.cbWndExtra = 0;
    wndEx.hbrBackground = (HBRUSH)COLOR_BACKGROUND;
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hInstance = GetModuleHandle(NULL);
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.lpszClassName = (LPCTSTR) this->lpParameter;
    wndEx.lpszMenuName = NULL;
    wndEx.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;

    if (!RegisterClassEx(&wndEx))
    {
        return;
    }

    HWND hWnd = CreateWindow(wndEx.lpszClassName,
        TEXT("Basic Thread Management"), WS_OVERLAPPEDWINDOW, 200,
        200, 800, 600, HWND_DESKTOP, NULL, wndEx.hInstance, NULL);

    UpdateWindow(hWnd);
    ShowWindow(hWnd, SW_SHOW);

    MSG msg = { 0 };
    while (GetMessage(&msg, 0, 0, 0))
```

```
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);
}

int main()
{
    Thread thread;
    thread.Create(TEXT("WND_CLASS_1"));

    Thread pthread;
    pthread.Create(TEXT("WND_CLASS_2"));

    thread.Join();
    pthread.Join();

    return 0;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_CLOSE:
        {
            DestroyWindow(hWnd);
            break;
        }
        default:
        {
            return DefWindowProc(hWnd, uMsg, wParam, lParam);
        }
    }
    return 0;
}
```

How it works...

The `CThread` class is an abstract class with a pure virtual member function, `Run`. It's not meant to be used as is; it must be derived first. We will explain it thoroughly. First, we will declare the class `CLock`. This class is used for synchronized execution when needed. Implementation follows in the `CThread.cpp` file. In the `CLock` constructor, we try to create a mutex, or open an existing one. If a mutex is created, the thread that created it does not take the ownership. This is because `FALSE` is passed as the second parameter of a call to the `CreateMutex` API. A thread that wants a mutex ownership must request it by calling the `WaitForSingleObject` function. If a mutex is held by another thread, the thread that wants the mutex will be blocked, and will wait until the mutex is released. After the operation is completed, `CLock` is released and its destructor is called. In the `CLock` destructor, a mutex is released and the handle is closed.

For the `CThread` class, we have defined the following seven macros required for the thread states and execution methods:

- ▶ `STATE_RUNNING`: This sets the running flag. It means that the thread is running.
- ▶ `STATE_READY`: This sets the ready flag. It means that the thread is joining the caller.
- ▶ `STATE_BLOCKED`: This sets the blocked flag. It means that the thread is suspended.
- ▶ `STATE_ALIVE`: This sets the alive flag. It means that the thread is created.
- ▶ `STATE_ASYNC`: This sets the asynchronous flag. It means that the thread will perform execution concurrently—in other words, the caller won't block until the thread returns.
- ▶ `STATE_SYNC`: This sets the synchronic flag. It means that the thread will not perform the execution concurrently—in other words, the caller will block until the thread returns.
- ▶ `STATE_CONTINUOUS`: This sets the run forever flag. It means that the thread will have to do some continuous work that will never stop, or at least not until the process exit.

The `CThread` constructor implementation is quite simple; we only need to set the class attribute's value to zero. The `Create` method creates the actual thread. The first argument, `lpParameter`, must be provided and it will later be passed to the thread, `StartAddress`, where it will begin its execution. The second parameter, `dwAsyncState`, represents the thread execution method. The parameter is, by default, passed with the `STATE_ASYNC` value, but the user can change it to `STATE_SYNC`, `STATE_CONTINUOUS`, or both if necessary. If `STATE_SYNC` is set, the `CLock` object is used. The third parameter `dwCreationFlag` sets the thread state to zero by default. This means that the thread will start its execution immediately after creation. It is possible to create a thread in the suspended state by providing the `STATE_BLOCKED` flag, which is equivalent to `CREATE_SUSPENDED`. If the `STATE_CONTINUOUS` flag is set, the event object is created, or if it already exists, then the event object is opened. This flag is meant for stopping the thread that has to run some continuous task in a normal manner, without terminating it on process exit.

The `Join` method sets the `STATE_READY` flag and wait for the thread object to become signaled, or until the interval `dwWaitInterval` expires. The `dwWaitInterval` parameter is set to `INFINITE` by default.

The `Suspend` and `Resume` methods suspend and resume the thread respectively. The `STATE_BLOCKED` flag is set in `Suspend` and unset in `Resume`. The thread class has the `lpUserData` attribute designed for the future user-specific data to be set if necessary. The `SetUserData` and `GetUserData` methods set and get the user data value respectively. The `GetAsyncState` method returns the object asynchronous state.

The `GetState` and `SetState` methods get and set the `dwState` value or object state respectively. It is meant for additional object state change.

The `Alert` method changes the event's state to signaled. It is meant for a situation where `STATE_CONTINUOUS` is used. If a thread is meant to run a continuous task, there has to be some mechanism to stop the thread in a normal manner. So when we have one or more threads running continuously, we only have to call `Alert` from one thread instance. It will alert all the threads to stop continuous work and return.

As the class member functions (methods) have a first implicit argument (not shown) `this`, we must declare the `StartAddress` method as **static**. However, we can pass the object address (the `this` pointer) to the `StartAddress` method and make it behave like a non-static method using the pure virtual function, `Run`.

The `GetId` and `GetHandle` methods return a unique thread identifier and a thread handle respectively. Both the methods are implemented as inline routines.

There's more...

We were following the object-oriented approach very precisely in the previous thread implementation due to code reuse and simplicity of programming. The following topic will demonstrate the usage of `CThread` with very little difficulty.

Implementing threads without synchronization

In a lot of situations, parallel tasks need to occur without depending on each other. Such situations could happen in an application with more user-interface dialogs. For example, on a primary application window, there is a **list view** with customers shown. On another application window (or application tab), there is a list view with products shown. When we start such an application, it would be foolish to load the customers first and then the products. We can start two threads, each responsible for loading the object from the database (the first thread would load the customers, and the other one would load the products).

Now, let's review a simple example. We have to initialize the main application window and then need to create four child windows. For each window, we will create a separate thread with no constraints on system resources—in our example, a brush for painting each window respectively. Later, in our next example, we will assume that only one brush is available to demonstrate the synchronizing action.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure using the following steps:

1. Create a new empty C++ Windows application named `AsyncThreads`.
2. Add a new source file named `AsyncThreads`. Open `AsyncThreads.cpp`.
3. Add the following code to it:

```
#include "CThread.h"
#include <windowsx.h>
#include <tchar.h>
#include <time.h>

#define THREADS_NUMBER 4
#define WINDOWS_NUMBER 10
#define SLEEP_TIME 500

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);

class Thread : public CThread
{
protected:
    virtual void Run(LPVOID lpParameter = 0);
};

void Thread::Run(LPVOID lpParameter)
{
    // reseed random generator
    srand((unsigned)time(NULL));

    HWND hWnd = (HWND)GetUserData();
```

```
RECT rect;

// get window's dimensions
BOOL bError = GetClientRect(hWnd, &rect);
if (!bError)
{
    return;
}

int iClientX = rect.right - rect.left;
int iClientY = rect.bottom - rect.top;

//do not draw if the window does not have any dimensions
if ((!iClientX) || (!iClientY))
{
    return;
}

// get device context for drawing
HDC hDC = GetDC(hWnd);

if (hDC)
{
    // draw the ten random figures
    for (int iCount = 0; iCount < WINDOWS_NUMBER; iCount++)
    {
        // set coordinates
        int iStartX = (int)(rand() % iClientX);
        int iStopX = (int)(rand() % iClientX);
        int iStartY = (int)(rand() % iClientY);
        int iStopY = (int)(rand() % iClientY);
        // set the color
        int iRed = rand() & 255;
        int iGreen = rand() & 255;
        int iBlue = rand() & 255;

        // create a solid brush
        HANDLE hBrush = CreateSolidBrush(GetNearestColor(hDC,
        RGB(iRed, iGreen, iBlue)));
        HANDLE hbrOld = SelectBrush(hDC, hBrush);

        Rectangle(hDC, min(iStartX, iStopX), max(iStartX, iStopX),
        min(iStartY, iStopY), max(iStartY, iStopY));

        // delete the brush
    }
}
```

```

        DeleteBrush(SelectBrush(hDC, hbrOld));
    }

    // release the Device Context
    ReleaseDC(hWnd, hDC);
}

Sleep(SLEEP_TIME);

return;
}

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    WNDCLASSEX wndEx = { 0 };

    wndEx.cbClsExtra = 0;
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.cbWndExtra = 0;
    wndEx.hbrBackground = (HBRUSH)COLOR_BACKGROUND;
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hInstance = hThis;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.lpszClassName = _T("async_thread");
    wndEx.lpszMenuName = NULL;
    wndEx.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;

    if (!RegisterClassEx(&wndEx))
    {
        return 1;
    }

    HWND hWnd = CreateWindow(wndEx.lpszClassName,
        TEXT("Basic Thread Management"), WS_OVERLAPPEDWINDOW, 200,
        200, 840, 440, HWND_DESKTOP, NULL, wndEx.hInstance, NULL);

    HWND hRects[THREADS_NUMBER];

```

```

hRects[0] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
    WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 20, 20, 180,
    350, hWnd, NULL, hThis, NULL);
hRects[1] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
    WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 220, 20, 180,
    350, hWnd, NULL, hThis, NULL);
hRects[2] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
    WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 420, 20, 180,
    350, hWnd, NULL, hThis, NULL);
hRects[3] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
    WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 620, 20, 180,
    350, hWnd, NULL, hThis, NULL);

UpdateWindow(hWnd);
ShowWindow(hWnd, SW_SHOW);

Thread threads[THREADS_NUMBER];
for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
{
    threads[iIndex].Create(NULL, STATE_ASYNC | STATE_CONTINUOUS);
    threads[iIndex].SetUserData(hRects[iIndex]);
}

SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR)threads);

MSG msg = { 0 };
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);
return 0;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:

```

```

    {
        PostQuitMessage(0);
        break;
    }
case WM_CLOSE:
{
    Thread* pThread = (Thread*)GetWindowLongPtr(hWnd,
        GWLP_USERDATA);
    pThread->Alert();
    for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
    {
        pThread[iIndex].Join();
    }
    DestroyWindow(hWnd);
    break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}

```

4. Now, copy `CThread.h` and `CThread.cpp` to the project directory.
5. Open **Solution Explorer** and right-click on **Header files**. Select **Add Existing** and add the `CThread.h` file.
6. Open **Solution Explorer** and right-click on **Source files**. Select **Add Existing** and add the `CThread.cpp` file.

How it works...

We are using the `CThread` class that we've implemented in our previous example. As we said before, we need to subclass it in order to use it. So, we've created the `Thread` class that is derived from `CThread`. There is a routine, `Run`, which we must define (implement) as it was declared abstract.

Let's explain the `Run` routine. First, we will calculate the window rectangle size. We need these coordinates in order to sustain what window area to paint. For that purpose, we will use the `GetClientRect` API. Then, we will determine the color so that we can create a proper brush. Now, we need the device context in order to paint. The `GetDC` API is used. We will use `CreateSolidBrush` in addition to the `GetNearestColor` call to get a known color. After using the resources, we will free them by calling proper methods such as `DeleteBrush` and `ReleaseDC` respectively.

Let's review the main routine now. First, the `WNDCLASSEX` object that is needed for the application window is created, initialized, and registered with the `RegisterClassEx` call. After we've created a window with the `CreateWindow` API, we have to create four child windows where the threads will perform the paint action. Calls to `UpdateWindow` and `ShowWindow` follow. We will create four thread objects. Next, we'll call its method, `Create`, with the `STATE_ASYNC` and `STATE_CONTINUOUS` flags merged using the operator, bitwise **OR** (`|`), and then we'll set the user data pointer to the child window handle, which will later be used to identify the portion of the window that it needs to paint.

The call to `SetWindowLongPtr` follows. This routine associates certain user data (with the provided flag `GWLP_USERDATA`) with the window identified by the first parameter—in our case, the main application window. The third parameter is meant for a specific user data that needs to be associated with the window. We will pass the `Thread` array address, so that we can use it later in `WindowProcedure`, after that application enters the message loop.

In `WindowProcedure`, we are processing only two cases: when the user closes the application and when the window needs to be destroyed. The case `WM_CLOSE` is where we need to wait for threads to safely join. First, we'll obtain the address of the thread array by calling `GetWindowLongPtr`, where we saved its address before. As we said before, it is enough that only one thread calls the method, `Alert`. All the threads that are listening for an event signal will receive it. Then, we'll call `Join` for each thread and exit the program.

There's more...

As you will notice, it is a very simple example with great complements to properly design the `CThread` class previously defined. Once again, the object-oriented approach is useful when you need to take advantage of code reuse.

Using synchronized threads

Contrary to the previous recipe, there are a lot of situations where parallel tasks need to occur with dependency on each other. Such situations could occur in an application with a user interface dialog for displaying customers and for the selected customer payment history. For example, on the application window, there is a list view with customers. Below, there is a list view with the payment history for a selected customer. When loading the initial set of customers, we need to display the first customer's (selected on load) payment history. We'll start one thread for loading the customer list. At the same time, another thread that is responsible for loading a specific customer payment history will also start. Until the first customer is loaded, the payment thread must wait for the customer ID.

Now, let's go back to the example from the previous recipe. We'll assume the same situation, but this time, with limited system resources, we have to initialize the main application window, and then we need to create four child windows. For each window, we will create a separate thread; this time, we will have only one brush for painting each window respectively.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure using the following steps:

1. Create a new empty C++ Windows application named `SyncThreads`.
2. Add a new source file named `SyncThreads`. Open `SyncThreads.cpp`.
3. Add the following code to it:

```
#include "CThread.h"
#include <windowsx.h>
#include <tchar.h>
#include <time.h>

#define THREADS_NUMBER    4
#define WINDOWS_NUMBER    10
#define SLEEP_TIME        500

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);

class Thread : public CThread
{
protected:
    virtual void Run(LPVOID lpParameter = 0);
};

void Thread::Run(LPVOID lpParameter)
{
    // reseed random generator
    srand((unsigned)time(NULL));

    HWND hWnd = (HWND)GetUserData();
    RECT rect;

    // get window's dimensions
    BOOL bError = GetClientRect(hWnd, &rect);
    if (!bError)
    {
```

```
        return;
    }
    int iClientX = rect.right - rect.left;
    int iClientY = rect.bottom - rect.top;

    //do not draw if the window does not have any dimensions
    if ((!iClientX) || (!iClientY))
    {
        return;
    }

    // get device context for drawing
    HDC hDC = GetDC(hWnd);

    if (hDC)
    {
        // draw the ten random figures
        for (int iCount = 0; iCount < WINDOWS_NUMBER; iCount++)
        {
            // set coordinates
            int iStartX = (int)(rand() % iClientX);
            int iStopX = (int)(rand() % iClientX);
            int iStartY = (int)(rand() % iClientY);
            int iStopY = (int)(rand() % iClientY);
            // set the color
            int iRed = rand() & 255;
            int iGreen = rand() & 255;
            int iBlue = rand() & 255;

            // create a solid brush
            HANDLE hBrush = CreateSolidBrush(GetNearestColor(hDC,
            RGB(iRed, iGreen, iBlue)));
            HANDLE hbrOld = SelectBrush(hDC, hBrush);

            Rectangle(hDC, min(iStartX, iStopX), max(iStartX, iStopX),
            min(iStartY, iStopY), max(iStartY, iStopY));

            // delete the brush
            DeleteBrush(SelectBrush(hDC, hbrOld));
        }

        // release the Device Context
        ReleaseDC(hWnd, hDC);
    }
}
```



```

    }

    Sleep(SLEEP_TIME);

    return;
}

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    WNDCLASSEX wndEx = { 0 };

    wndEx.cbClsExtra = 0;
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.cbWndExtra = 0;
    wndEx.hbrBackground = (HBRUSH)COLOR_BACKGROUND;
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
    wndEx.hInstance = hThis;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.lpszClassName = _T("async_thread");
    wndEx.lpszMenuName = NULL;
    wndEx.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;

    if (!RegisterClassEx(&wndEx))
    {
        return 1;
    }

    HWND hWnd = CreateWindow(wndEx.lpszClassName,
        TEXT("Basic Thread Management"), WS_OVERLAPPEDWINDOW, 200,
        200, 840, 440, HWND_DESKTOP, NULL, wndEx.hInstance, NULL);

    HWND hRects[THREADS_NUMBER];
    hRects[0] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
        WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 20, 20, 180, 350,
        hWnd, NULL, hThis, NULL);
    hRects[1] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
        WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 220, 20, 180, 350,
        hWnd, NULL, hThis, NULL);
    hRects[2] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
        WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 420, 20, 180, 350,
        hWnd, NULL, hThis, NULL);

```

```

hRects[3] = CreateWindow(_T("STATIC"), _T(""), WS_BORDER |
    WS_CHILD | WS_VISIBLE | WS_CLIPCHILDREN, 620, 20, 180, 350,
    hWnd, NULL, hThis, NULL);

UpdateWindow(hWnd);
ShowWindow(hWnd, SW_SHOW);

Thread threads[THREADS_NUMBER];
for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
{
    threads[iIndex].Create(NULL, STATE_SYNC | STATE_CONTINUOUS);
    threads[iIndex].SetUserData(hRects[iIndex]);
}

SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR)threads);

MSG msg = { 0 };
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);
return 0;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_CLOSE:
        {
            Thread* pThread = (Thread*)GetWindowLongPtr(hWnd,
                GWLP_USERDATA);
            pThread->Alert();
            for (int iIndex = 0; iIndex < THREADS_NUMBER; iIndex++)
            {

```

```
        pThread[iIndex].Join();
    }
    DestroyWindow(hWnd);
    break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}
```

4. Now, copy `CThread.h` and `CThread.cpp` to the project directory.
5. Open **Solution Explorer** and right-click on **Header files**. Select **Add Existing** and add the `CThread.h` file.
6. Open **Solution Explorer** and right-click on **Source files**. Select **Add Existing** and add the `CThread.cpp` file.

How it works...

Well, this is almost the same code as before. The only difference being that this code differs by a single letter. Instead of initializing the threads on creation with `STATE_ASYNC`, we've used the `STATE_SYNC` flag. That's all.

Everything that we need for both our examples was done much before with proper design of the `CThread` class.

There's more...

We would like to highlight the power of the object-oriented approach here, as well as the simplicity of code reuse with proper design of the problem itself. It is important to properly decide on the major points in your problem before you actually start coding. Sometimes, it is better to spend a lot of time thinking and designing the solution itself before any line of code is made. With such an approach, programming becomes a lot easier, especially with the usage of object-oriented techniques.

Win32 synchronization objects and techniques

In order for the threads to be executed, they must be scheduled for execution. In order for them to be executed without interfering with each other, they must be synchronized.

Suppose that one thread creates a brush and then creates several threads that share the same brush and draw with it. The first thread must not destroy the brush until other threads have finished drawing. Or, for example, one thread accepts input from the user and writes it to a file, while another thread reads from the file and processes the text. The thread that reads can't read if a thread that writes is writing that file. Both situations require a means of coordination among several threads.

One solution would be to create a global *Boolean* variable, `bDone`, which one thread uses to signal another thread. The thread that writes could set `bDone` to `TRUE`, while the thread that needs to read executes a loop until it finds the indicator `bDone` changed. Such a mechanism would work but the thread that executes the loop consumes a lot of CPU time. Instead, the Windows operating system supports a number of synchronization objects as follows:

- ▶ The **mutex** object (mutual exclusion) works as a narrow gate where it can take only one thread at a time
- ▶ The **semaphore** object works as a gateway where certain number of threads can pass at one instance of time
- ▶ The **event** object that broadcasts a public signal, which any thread that listens can receive (hear)
- ▶ The **critical section** object works just like a mutex but only within a single process

All of these objects are system objects created by the **Object Manager**. Although each synchronization object coordinates with the various interactions, they all work in the same way. The thread that wants to execute a coordinated operation waits for an answer from one of these objects and resumes only when it receives it. The dispatcher removes the objects from a queue, which are in the wait state, to avoid wasting CPU time. When the signal arrives, the dispatcher allows the thread to continue.

Even the unrelated processes can share mutexes, semaphores, and events. Processes can coordinate with the shared object's activities as threads can. There are three mechanisms for distribution. One is inheritance, when a process creates another and the new process gets a copy of the parent identification codes. Only these identification codes, which are marked to be inherited when they are created, will be passed (inherited).

Other methods include calling a function to create another ID code for an existing object. Which routine we call depends on the information that we already have. If we have the identification codes for the source and the destination processes, we will use the `DuplicateHandle` API. If we have the name of the object, we will use one of the `Open` functions. The two programs can agree in advance when it comes to the names of objects that they share, or one may refer to another object name via shared memory or pipes.

Mutexes, semaphores, and events are held in memory until all the processes that own them are completed, or until all the objects' identification codes are closed with the `CloseHandle` API.

Synchronization object – mutex

How and when the signal arrives depends on the object. For example, one important feature of a mutex is that only one thread can own it. A mutex does nothing, except that it allows ownership for only one thread at a time. If you need a few threads to work with one file, you can create a mutex to protect the file. When any thread starts an operation on the file, it must first ask for the ownership of the mutex, and if no other thread is holding the mutex, the thread gets an ownership. If, on the other hand, some other thread just *grabbed* the mutex for itself, the request fails; the thread is blocked and becomes suspended while waiting to take the ownership of the mutex. When a thread finishes writing, it releases the mutex, and the waiting thread becomes *alive* again on receiving the mutex and performs the requested operations on the file.

A mutex does not protect anything active. It works in the following manner: the threads that will use it agree to not use a shared object without holding the mutex ownership first. There is nothing to prevent the threads from attempting to enroll at once. A mutex is a signal, like the *Boolean* variable `bDone` from the previous example. A mutex can be used to protect a global variable, a hardware port, the identification code for the pipe, or the client area of the window. Every time multiple threads share a system resource, you should consider using a mutex for the synchronization, in order to avoid the mutual conflicts.

The following code shows the routines for mutex usage. To create or open an existing mutex object, you can use `CreateMutex`:

```
HANDLE WINAPI CreateMutex(  
    LPSECURITY_ATTRIBUTES lpMutexAttributes,  
    BOOL                  bInitialOwner,  
    LPCTSTR                szName  
);
```

You can set the security attributes or leave them empty for default values. If `TRUE` is set for `bInitialOwner`, the caller gets immediate ownership on the mutex. The mutex name must be supplied.

It is possible to open an existing mutex using `OpenMutex`:

```
HANDLE WINAPI OpenMutex(
    DWORD    dwDesiredAccess,
    BOOL     bInheritHandle,
    LPCTSTR  szName
);
```

If the mutex does not exist, the routine fails and returns `ERROR_FILE_NOT_FOUND`. After the thread finishes its work, it must release the mutex in order for other threads to get it. The `ReleaseMutex` API is used:

```
BOOL WINAPI ReleaseMutex(
    HANDLE  hMutex
);
```

It has one argument—handle to mutex, which was previously returned by `CreateMutex` or `OpenMutex`.

Synchronization object – semaphore

As we said before, a semaphore works like a mutex, with the difference that multiple objects can hold its ownership. Assume that we need a full logical CPU core for a task, for example, a complex mathematical calculation. If we use only one thread per core, we can get the proper result, and if we use more than one thread per core, we won't get proper results. Assume that we also need more threads than the number of logical cores to work simultaneously.

In such a situation, the semaphore object would be a great choice, because we would set its maximum value to be the number of logical cores on the machine. While the number of threads does not exceed the number of cores, they can work concurrently, optimized for the maximum result. When the number of threads exceeds the number of cores, some of the threads will be suspended and will wait for another thread to complete.

While only one thread at a time can get the mutex, the semaphore remains signaled until its acquisition number reaches the maximum number of objects that can hold the ownership of the semaphore. If a thread tries to wait for the semaphore, it will be suspended until another thread does not release it.

The following are the routines for semaphore usage. To create or open an existing semaphore object, you can use `CreateSemaphore`:

```
HANDLE WINAPI CreateSemaphore(
    LPSECURITY_ATTRIBUTES  lpSemaphoreAttributes,
    LONG                   lInitialCount,
    LONG                   lMaximumCount,
    LPCTSTR                 szName
);
```

You can set the security attributes or leave it empty for the default values. The initial count is set to determine the initial signaled state of the semaphore. If it is zero, its state is nonsignaled. It also must be smaller than the `lMaximumCount` value. The maximum count represents the maximum number of objects (threads) that can hold ownership at the same time. The semaphore name must be supplied.

It is possible to open an existing semaphore using `OpenSemaphore`:

```
HANDLE WINAPI OpenSemaphore(  
    DWORD    dwDesiredAccess,  
    BOOL     bInheritHandle,  
    LPCTSTR  szName  
);
```

If a semaphore does not exist, the routine fails and returns `NULL`. After the thread finishes its work, it must release the semaphore in order for its counter to be decremented, so that other threads can get it. The `ReleaseSemaphore` API is used:

```
BOOL WINAPI ReleaseSemaphore(  
    HANDLE    hSemaphore,  
    LONG     lReleaseCount  
    LPLONG    lpPreviousCount  
);
```

It has three arguments. The first one is the handle to the semaphore, which was previously returned by `CreateSemaphore` or `OpenSemaphore`. The second one is how many objects the semaphore needs to decrement. Well, usually one, because the thread that releases it needs to decrement it by one. In a situation where a thread takes the semaphore and then creates another thread that will also request for the semaphore, and the thread creator finishes its task before the thread that was created, the newly created thread won't know that it will forcibly be terminated, so the thread that created it can terminate the child thread and decrement the semaphore for two.

Synchronization object – event

The event is an object that a program creates when it needs some kind of mechanism for alerting the thread when some action occurs. In its simplest form—the manual reset event—the object sets its signal on and off as a response to the two commands `SetEvent` (signal on) and `ResetEvent` (signal off). When the signal is turned on, all the threads that are waiting for the event will receive it. When the signal is switched off, all the threads that are waiting for an event are blocked. Unlike a mutex and semaphores, events change their state only when a thread explicitly sets or resets it.

For example, the manual reset event can be used to allow certain threads to run only when the program is not painting its window, or only after the user enters certain information.

The automatic reset event always releases only one thread at each signal before resetting. It could be useful for a program where the main thread prepares data for other working threads. Each time a new set of data is ready, the main thread sets the event and the working threads are released. Other working threads remain waiting in a queue to get a new task.

In addition to setting and resetting, the event can be pulsed. A pulse sets a signal for a very short time and then turns it off. When we pulse a manual reset event, we allow all the threads that are waiting to continue, and then the event is reset. Pulsing an automatic event allows only one thread that is waiting to pass, and then resets the event. If no thread is waiting, neither will pass. Setting an automatic event, on the other hand, will cause the event to leave its signal while a thread is waiting. As soon as one thread passes, the event is reset.

The following are the routines for event usage. To create or open an existing event object, you can use `CreateEvent` or `CreateEventEx`:

```
HANDLE WINAPI CreateEvent (
    LPSECURITY_ATTRIBUTES  lpEventAttributes,
    BOOL                   bManualReset,
    BOOL                   bInitialState,
    LPCTSTR                szName
);

HANDLE WINAPI CreateEventEx(
    LPSECURITY_ATTRIBUTES  lpEventAttributes,
    LPCTSTR                szName,
    DWORD                  dwFlags,
    DWORD                  dwDesiredAccess
);
```

For the `CreateEvent` API, you can set the security attributes or leave it empty for the default values. The `bManualReset` value determines the event object. If the `bManualReset` parameter is set to `TRUE`, `ResetEvent` must be used to set the object state to nonsignaled. The `bInitialState` value sets the state of an event. If the value is set to `TRUE`, the object is set to the signaled state immediately. The event name must be supplied.

On the other hand, `CreateEventEx` works pretty much the same way, only its terminology is different. Apart from the security attributes, which are the same, you need to supply the name and value for `dwFlags`, which have predefined macros for the initial set and manual reset event types: `CREATE_EVENT_INITIAL_SET` and `CREATE_EVENT_MANUAL_RESET`. The only real difference is that `dwDesiredAccess` can set the access mask for the event object. For a complete list of *Synchronization Object Security and Access Rights*, refer to MSDN <http://msdn.microsoft.com/en-us/library/windows/desktop/ms686670%28v=vs.85%29.aspx>.

It is possible to open an existing event object using `OpenEvent`:

```
HANDLE WINAPI OpenEvent (
    DWORD    dwDesiredAccess,
    BOOL     bInheritHandle,
    LPCTSTR  szName
);
```

If the event does not exist, the routine fails and returns `NULL`. Calling `OpenEvent` enables multiple threads to hold the same event, but only if the event was previously created by calling `CreateEvent` or `CreateEventEx`.

For setting an event object, `SetEvent` is used:

```
BOOL WINAPI SetEvent (
    HANDLE    hEvent
);
```

It has one argument—handle to the event object previously created by `CreateEvent` and `CreateEventEx`, or opened by `OpenEvent`. The manual reset event state remains signaled until it is explicitly set to nonsignaled, by calling the `ResetEvent` API.

To reset an event object, `ResetEvent` is used:

```
BOOL WINAPI ResetEvent (
    HANDLE    hEvent
);
```

It has one argument—handle to the event object previously created by `CreateEvent` and `CreateEventEx`, or opened by `OpenEvent`. The manual reset event state remains nonsignaled until it is explicitly set to signaled, by calling the `SetEvent` API.

To pulse an event object, `PulseEvent` is used:

```
BOOL WINAPI PulseEvent (
    HANDLE    hEvent
);
```

This method sets the event object to the signaled state and then resets it to the nonsignaled state, releasing a specific number of waiting threads.

Synchronization object – critical section

A critical section performs the same function as a mutex, except that a critical section can't be shared. It is visible only within a single process. Both the critical section and the mutex allow only one thread to own it, but critical sections work much faster and with less overhead.

The routines that operate with critical sections do not use the same terminology as the ones that work with a mutex, but they do mostly the same thing.

Unlike other objects that have been waiting for one of the wait functions, the critical section object works in a different way. The critical section object enters its unsignaled state.

After it is released, the critical section object enters its signaled state.

In order to use the critical section object, first we need to declare it:

```
CRITICAL_SECTION cs;
```

After that, we need to initialize it:

```
void WINAPI InitializeCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;
```

Then, we can use it by asking a thread to enter the critical region by calling the `EnterCriticalSection` or `TryEnterCriticalSection` API:

```
void WINAPI EnterCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;  
BOOL WINAPI TryEnterCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;
```

After the thread finishes its task, it must leave the critical region by calling the `LeaveCriticalSection` API:

```
void WINAPI LeaveCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;
```

After that, it needs to free the resources by calling the `DeleteCriticalSection` API:

```
void WINAPI DeleteCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;
```


4

Message Passing

In this chapter, we will cover the following topics:

- ▶ Explaining Message Passing Interface
- ▶ Understanding a message queue
- ▶ Using the thread message queue
- ▶ Communicating through the pipe object

Introduction

To explain the **Message Passing Interface (MPI)**, we need to understand the operating system well. As we mentioned in *Chapter 1, Introduction to C++ Concepts and Features*, modern Windows operating systems are designed as *event-driven* systems.

An event-driven system is the one where some action occurs as a response to some signal sent to the operating system. For example, when you move a mouse, `cursor X-coordinate change` and `cursor Y-coordinate change` events are raised (we say an event is raised, not occurred). When you press any keyboard key, another event is raised—`key pressed` and so on.

At the operating system level, a large numbers of events are raised every few milliseconds, hidden from the user. Programming without events would be very difficult, especially for synchronization tasks.

One form of an event is the *message* at the operating system level. When you press your left mouse button, the operating system calls the window procedure (*event handler*) for the window in which region (area) the click occurred. Parameters that define a message, along with event-specific details, such as click coordinates, are passed to the window procedure that's being called.

Every application that has a UI must have some form of a window. Top-level windows must implement top-level window handlers or window procedures, which will handle application events (messages), for example, button click, window paint, and others.

To be able to process messages for the window, an application must implement the event handler that the OS will call each time an event is raised. If you remember that every application must have its primary thread, then you can use the primary thread to process messages from other threads. In other words, you can use the window procedure from the main application window to receive messages from other threads to communicate with each other.

There are other ways to pass a message to the thread, such as the `PostThreadMessage` API, but the thread to which the message is posted must have created a message queue. We will explain this approach too, later in this chapter. However, using the window message queue is much easier, in cases when your application has a UI.

This chapter will explain in detail how to use message passing for various tasks, especially for synchronization between separate threads.

Explaining the Message Passing Interface

In order to explain message passing in detail we'll discuss the button click. Like any other control, it belongs to some top-level window, whose procedure is called when a button click event is raised as a response to the mouse button press.

The following example will demonstrate the basic usage of message handling, responsible for the mouse move and button-click.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Win32 project application named `basic_MPI`.
2. Open **Solution Explorer** and right-click on **Source file**. Add the new source file named `basic_MPI`. Open `basic_MPI.cpp` and add the following code to it:

```
#include <windows.h>
#include <windowsx.h>
```

```

#include <tchar.h>

#include <commctrl.h>

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

#define BUTTON_MSG      100
#define BUTTON_CLOSE    101
#define LABEL_TEXT      102

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__basic_MPI_wnd_class__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));

    if (!RegisterClassEx(&wndEx))

```

```

    {
        return 1;
    }

    InitCommonControls();

    HWND hWnd = CreateWindow(szWindowClass, _T("Basic Message
    Passing Interface"), WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX, 200, 200, 440, 340, NULL, NULL, wndEx.hInstance,
    NULL);

    if (!hWnd)
    {
        return NULL;
    }

    HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
        FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
        | FF_MODERN, _T("Microsoft Sans Serif"));

    HWND hButtonMsg = CreateWindow(_T("BUTTON"), _T("Show msg"),
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 190,
        260, 100, 25, hWnd, (HMENU)BUTTON_MSG, wndEx.hInstance, NULL);

    HWND hButtonClose = CreateWindow(_T("BUTTON"), _T("Close me"),
        WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 310, 260,
        100, 25, hWnd, (HMENU)BUTTON_CLOSE, wndEx.hInstance, NULL);

    HWND hText = CreateWindow(_T("STATIC"), NULL, WS_CHILD |
        WS_VISIBLE | SS_LEFT | WS_BORDER, 15, 20, 390, 220, hWnd,
        (HMENU)LABEL_TEXT, wndEx.hInstance, NULL);

    SendMessage(hButtonMsg, WM_SETFONT, (WPARAM)hFont, TRUE);
    SendMessage(hButtonClose, WM_SETFONT, (WPARAM)hFont, TRUE);
    SendMessage(hText, WM_SETFONT, (WPARAM)hFont, TRUE);

    ShowWindow(hWnd, iWndShow);
    UpdateWindow(hWnd);

    MSG msg;
    while (GetMessage(&msg, 0, 0, 0))
    {
        TranslateMessage(&msg);
    }

```

```

        DispatchMessage(&msg);
    }

    UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

    return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case BUTTON_MSG:
                {
                    MessageBox(hWnd, _T("Hello!"), _T("Basic MPI"), MB_OK |
MB_TOPMOST);
                    break;
                }
                case BUTTON_CLOSE:
                {
                    PostMessage(hWnd, WM_CLOSE, 0, 0);
                    break;
                }
            }
            break;
        }

        case WM_MOUSEMOVE:
        {
            int xPos = GET_X_LPARAM(lParam);
            int yPos = GET_Y_LPARAM(lParam);

            TCHAR szBuffer[4096];
            wsprintf(szBuffer,
                _T("\n\n\t%ws\t%d\n\t%ws\t%d"),
                _T("Cursor X position:"), xPos,
                _T("Cursor Y position:"), yPos);

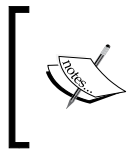
            HWND hText = GetDlgItem(hWnd, LABEL_TEXT);
            SetWindowText(hText, szBuffer);
            break;
        }
    }
}

```

```
    }  
    case WM_DESTROY:  
    {  
        PostQuitMessage(0);  
        break;  
    }  
    default:  
    {  
        return DefWindowProc(hWnd, uMsg, wParam, lParam);  
    }  
}  
  
return 0;  
}
```

How it works...

The purpose of this example was to demonstrate the basic message passing. You need to pay close attention to the `BUTTON_CLOSE` and `LABEL_TEXT` macros that we defined earlier. We need them later as child-control identifiers. For a child window, `hMenu` specifies the child-window identifier.



The child-control identifier is an integer value used by the window to notify its parent about events. The application determines the child-window identifier; it must be unique for all child windows for the same parent window (from MSDN).

Let's examine the `winMain` entry point now. We've created the `WNDCLASSEX` instance first and initialized it. After a successful window class registration with subsequent call to `RegisterClassEx`, we create a main window, followed by a call to `CreateWindow`.

Then, we need to create a text label and button control. Button and label are also windows but with predefined system class (`BUTTON`, `STATIC`, and so on). For the complete system class reference, refer to MSDN at [http://msdn.microsoft.com/en-us/library/windows/desktop/ms632679\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms632679(v=vs.85).aspx).

After a call to `ShowWindow`, the application enters its message loop by calling the Win32 API, `GetMessage`, in the `while` loop. When the application closes the top-level window, a call to `GetMessage` returns zero (0), and there are no more messages for the application to process.

As opposed to MS-DOS-based applications, Win32 applications are event-driven. They don't perform explicit function calls to obtain input. Instead, they wait for the system to pass input to them. The operating system passes all the messages designated for the application to the application window. The window has a function called a window procedure that the system calls whenever it has messages for the window. The window procedure processes the message and returns the control to the system.

Let's focus on `WindowProcedure` now. We decided to process only two messages: `WM_MOUSEMOVE`, which is responsible for mouse move, and `WM_COMMAND`, which is responsible for button click. For mouse move, we've used predefined macros, `GET_X_LPARAM`, to obtain the x-coordinate, and `GET_Y_LPARAM`, to obtain the y-coordinate. For button click, we are using the lower 16 bits from the `wParam` argument to determine which child control raised the click. After we determine which control raised an event, we can perform any processing that we need for that control. In our case, we are using the **Show msg** button to display a message box and the **Close me** button to close the application.

There's more...

Even though there are operating system specialized synchronization objects and routines, it is very handy and much easier to use window messages for synchronization tasks whenever possible. For example, if you run a separate thread asynchronously to do some task, you can use message passing to notify the main thread when the working thread finishes its task. In the examples to come, we will use message passing for communication between threads.

Understanding a message queue

To understand message passing better, let's review the **message queue**. Every top-level window has its own message queue that is created by the operating system. In a multitasking environment, there are many applications executed at the same time with a lot of windows displayed on the user's screen. The operating system is processing a very large number of messages, handling various events raised from the system itself, and running various applications. At the same time, the user is pressing keyboard keys, clicking the mouse, or swiping the touch screen.

To demonstrate message passing along with the queue operations, let's review the following example. Let's assume that we need to perform complex drawings represented with a graphical plot with four axes. Every axis represents a certain mathematical calculation that needs graphical representation.

We will create an application where one thread is responsible for a drawing based on the calculation performed by another thread, while the other thread is responsible for the calculation. Let's assume that the thread calculation time is smaller than the drawing time that is needed for a portion of the plot that has to be drawn.

We need to implement a message queue, where the threads that perform calculation store their messages. These threads (producers) are performing the calculations faster than the draw thread (consumer), which is processing the messages taken from the queue. Each time the producer finishes its task, it places a message in the queue, which is later processed by the consumer.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ Win32 empty project application named `queue_MPI`.
2. Open **Solution Explorer** and right-click on **Header files**. Add an existing header file `CQueue.h` from *Chapter 1, Introduction to C++ Concepts and Features*.
3. Open **Solution Explorer** and right-click on **Source files**. Add a new source file named `queue_MPI`. Open `queue_MPI.cpp`. Add the following code to it:

```
#include <windows.h>
#include <windowsx.h>
#include <tchar.h>
#include "CQueue.h"
#include <commctrl.h>

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \\  
name='Microsoft.Windows.Common-Controls' \\  
version='6.0.0.0' processorArchitecture='*' \\  
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

#define BUFFER_SIZE      4096

#define LABEL_TEXT       100

#define WM_PLOTDATA      WM_USER      + 1
#define WM_ENDTASK       WM_PLOTDATA + 1

#define THREAD_NUMBER    4
#define MAX_MSGCOUNT    10

#define CALCULATION_TIME 1000
```

```

#define DRAWING_TIME    2300

#define EVENT_NAME      _T( "__t_event__" )

typedef struct tagPLOTDATA
{
    int value;
    DWORD dwThreadId;
    int iMsgID;
} PLOTDATA, *PPLOTDATA;

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
DWORD WINAPI DrawPlot(LPVOID lpParameter);

int iMessageID = 0;
HANDLE gEvent = NULL;
HANDLE hThreads[THREAD_NUMBER];
HANDLE hThread = NULL;

CRITICAL_SECTION cs;
CQueue<PLOTDATA> queue;

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    InitializeCriticalSection(&cs);

    TCHAR* szWindowClass = _T("__basic_MPI_wnd_class__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);

```

```

wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndEx.lpszMenuName = NULL;
wndEx.lpszClassName = szWindowClass;
wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wndEx))
{
    return 1;
}

InitCommonControls();

HWND hWnd = CreateWindow(szWindowClass, _T("Basic Message
    Passing Interface"), WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX, 200, 200, 440, 300, NULL, NULL, wndEx.hInstance,
    NULL);

if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH |
    FF_MODERN, _T("Microsoft Sans Serif"));

HWND hText = CreateWindow(_T("STATIC"), NULL, WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 15, 20, 390, 220, hWnd,
    (HMENU)LABEL_TEXT, wndEx.hInstance, NULL);

SendMessage(hText, WM_SETFONT, (LPARAM)hFont, TRUE);

ShowWindow(hWnd, iWndShow);

MSG msg;
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

```

```

    }

    UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

    WaitForMultipleObjects(THREAD_NUMBER, hThreads, TRUE, INFINITE);

    for (int iIndex = 0; iIndex < THREAD_NUMBER; iIndex++)
    {
        CloseHandle(hThreads[iIndex]);
    }

    WaitForSingleObject(hThread, INFINITE);

    CloseHandle(hThread);
    CloseHandle(gEvent);

    DeleteCriticalSection(&cs);

    return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CREATE:
        {
            gEvent = CreateEvent(NULL, TRUE, FALSE, EVENT_NAME);

            for (int iIndex = 0; iIndex < THREAD_NUMBER; iIndex++)
            {
                hThreads[iIndex] = CreateThread(NULL, 0,
                    (LPTHREAD_START_ROUTINE)StartAddress, hWnd, 0, NULL);
            }

            hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
DrawPlot, hWnd, 0, NULL);
            break;
        }
        case WM_PLOTDATA:

```

```

    {
        PLOTDATA pData = (PLOTDATA)lParam;

        HWND hLabel = GetDlgItem(hWnd, LABEL_TEXT);

        TCHAR szBuffer[BUFFER_SIZE];
        GetWindowText(hLabel, szBuffer, BUFFER_SIZE);

        wsprintf(szBuffer,
            _T("%ws\n\n\tMessage has been received. Msg ID:\t%d"),
            szBuffer, pData->iMsgID);
        SetWindowText(hLabel, szBuffer);

        break;
    }
    case WM_ENDTASK:
    {
        HWND hLabel = GetDlgItem(hWnd, LABEL_TEXT);

        TCHAR szBuffer[BUFFER_SIZE];

        wsprintf(szBuffer,
            _T("\n\n\tPlot is drawn. You can close the window now.));
        SetWindowText(hLabel, szBuffer);
        break;
    }
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        break;
    }
    default:
    {
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}

return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    HWND hWnd = (HWND)lpParameter;

```

```

HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, EVENT_NAME);

if (hEvent != NULL)
{
    SetEvent(hEvent);
}

CloseHandle(hEvent);

int iCount = 0;
while (iCount++ < MAX_MSGCOUNT)
{
    // perform calculation
    Sleep(CALCULATION_TIME);

    // assembly result into PLOTDATA structure
    PLOTDATA pData = new PLOTDATA();
    pData->value = (rand() % 0xFFFF) - iMessageID;
    pData->dwThreadId = GetCurrentThreadId();
    pData->iMsgID = ++iMessageID;

    EnterCriticalSection(&cs);
    queue.Enqueue(pData);
    LeaveCriticalSection(&cs);

    PostMessage(hWnd, WM_PLOTDATA, 0, (LPARAM)pData);
}

return 0L;
}

DWORD WINAPI DrawPlot(LPVOID lpParameter)
{
    HWND hWnd = (HWND)lpParameter;

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, EVENT_NAME);

    WaitForSingleObject(hEvent, INFINITE);
    CloseHandle(hEvent);

    int iCount = 0;
    while (iCount++ < MAX_MSGCOUNT * THREAD_NUMBER)
    {
        EnterCriticalSection(&cs);

```



```

        PLOTDATA pData = queue.Dequeue();
        LeaveCriticalSection(&cs);

        if (!pData)
        {
            break;
        }

        // perform drawing
        Sleep(DRAWING_TIME);

        HWND hLabel = GetDlgItem(hWnd, LABEL_TEXT);

        TCHAR szBuffer[BUFFER_SIZE];

        wsprintf(szBuffer,
#ifdef _UNICODE
            _T("\n\n\t%ws\t%u\n\t%ws\t%d\n\t%ws\t%d"),
#else
            _T("\n\n\t%s\t%u\n\t%s\t%d\n\t%s\t%d"),
#endif
            _T("Thread ID:"), (DWORD)pData->dwThreadId,
            _T("Current value:"), (int)pData->value,
            _T("Message ID:"), pData->iMsgID);
        SetWindowText(hLabel, szBuffer);

        delete pData;
    }

    PostMessage(hWnd, WM_ENDTASK, 0, 0);

    return 0L;
}

```

How it works...

First, we need to define certain message values. A message is represented as a unique integer, larger than the Windows predefined `WM_USER` value. All the messages larger than the `WM_USER` (1024) value are considered application specific (user defined). The first 1024 messages are used (reserved) by Windows. We decided to define the two messages as follows:

- ▶ `WM_PLOTDATA` for the calculation thread to inform the main thread that calculation has been completed
- ▶ `WM_ENDTASK` to inform the main thread that all the drawing has been completed

In an application entry point, we have created a queue whose elements are references to the `PLOTDATA` type or some specific structure that the user has defined. After creating a window, we will associate the following queue address to the window section for user-specific data that belongs to the window:

```
SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR) &queue);
```

Using the `GWLP_USERDATA` macro as a parameter to set the `SetWindowLongPtr` API call, the function associates the address of the queue object in a manner that if you have the window handle (`hWnd`), you can always obtain the queue address. After creating the necessary controls, we need to create four threads for calculation purposes and one thread to draw. Whenever we need the queue object address, we can obtain it using `GetWindowLongPtr` as follows:

```
CQueue<PLOTDATA>* plotPtr = (CQueue<PLOTDATA>*)
GetWindowLongPtr( hWnd, GWLP_USERDATA );
```

In order to avoid the complexity of drawing (which is not the topic here), we will set the working threads to sleep for 1 second, while the draw thread will sleep for 2.3 seconds, only to create an environment where the consumer is slower than the producer. All the threads have fixed number of calculations to perform as well as the draw thread to draw.

After each calculation is completed, the `PLOTDATA` object is created and put into the queue. The thread then calls the `PostMessage` API to send (post) a message to the window procedure specified with the first parameter—the handle to the window. The second parameter specifies a message to send (post), while the third and fourth parameters are message specific. If it is a Windows message (less than `WM_USER`), then its meaning is known; if it is a user-defined message then both the parameters are left to the user to use. We are using the `lParam` parameter to pass the address of a newly created `PLOTDATA` object so that the window procedure can display the message ID or any value from the `PLOTDATA` object that the user needs.

After the producer creates a message, it places the message into the queue. The draw thread *consumer*, on the other hand, is taking the messages from the queue to perform drawing. If a pointer to the `PLOTDATA` object is obtained (the queue wasn't empty), the consumer performs drawing. After a successful drawing, the `PLOTDATA` object is destroyed in order to avoid memory leaks. When all the points are drawn and all the messages are processed, the draw thread uses `PostMessage` to notify the window procedure about completion.

There's more...

Even though we have used the `PostMessage` API, we are using the term *message is sent* and not the term *posted*. Why? Well, both the routines send messages; the only difference is that `PostMessage` returns without waiting for the callee to process the message. The `SendMessage` API does the same thing; it sends the messages, except that it waits while the callee processes the message, leaving the caller blocked till then. The messages in a message queue are retrieved using the `GetMessage` or `PeekMessage` API.

Using the thread message queue

As we said earlier, we can send a message to the thread without creating the window (UI). In our previous example, we were simply using the message queue created for the thread because it owns a window. Here, we will explain the ability to pass messages without using the window message queue.

Due to the simplicity of the example, we will send a message to the thread each time the user clicks on a button.

An analogy to such a task would be if we want to create an application that will display information about the weather forecast, we will have two threads: one is responsible for physical devices such as pressure meter, wind speed meter, and similar, while the other thread needs to wait for the data collected from the first thread. After it obtains the data, it can process it and display it to the user.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Windows application named `threadMPI`.
2. Open **Solution Explorer** and right-click on **Header files**. Add an existing header file `CQueue.h` from the *Chapter 1, Introduction to C++ Concepts and Features*.
3. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `threadMPI`. Open `threadMPI.cpp`.
4. Add the following code to it:

```
#include <windows.h>
#include <windowsx.h>
#include <tchar.h>
#include "CQueue.h"
#include <commctrl.h>

#pragma comment ( lib, "comctl32.lib" )
```

```

#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*'\\" )

#define BUFFER_SIZE      4096

#define BUTTON_CLOSE  100

#define T_MESSAGE      WM_USER      + 1
#define T_ENDTASK      T_MESSAGE  + 1

#define EVENT_NAME      _T( "__t_event__" )

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__basic_MPI_wnd_class__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;

```

```

    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));

    if (!RegisterClassEx(&wndEx))
    {
        return 1;
    }

    InitCommonControls();

    HWND hWnd = CreateWindow(szWindowClass, _T("Basic Message
    Passing Interface"), WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU |
    WS_MINIMIZEBOX, 200, 200, 440, 300, NULL, NULL, wndEx.hInstance,
    NULL);

    if (!hWnd)
    {
        return NULL;
    }

    HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
        FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH |
        FF_MODERN, _T("Microsoft Sans Serif"));

    HWND hButton = CreateWindow(_T("BUTTON"), _T("Send message"),
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | WS_TABSTOP, 310, 210, 100,
    25, hWnd, (HMENU)BUTTON_CLOSE, wndEx.hInstance, NULL);

    SendMessage(hButton, WM_SETFONT, (LPARAM)hFont, TRUE);

    ShowWindow(hWnd, iWndShow);

    HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, EVENT_NAME);

    DWORD dwThreadId = 0;

    HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
    StartAddress, hWnd, 0, &dwThreadId);

    SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR)&dwThreadId);

    WaitForSingleObject(hEvent, INFINITE);

```

```
CloseHandle(hEvent);

MSG msg;
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

PostThreadMessage(dwThreadId, T_ENDTASK, 0, 0);
WaitForSingleObject(hThread, INFINITE);

CloseHandle(hThread);
return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case BUTTON_CLOSE:
                {
                    DWORD* pdwThreadId = (DWORD*)GetWindowLongPtr(hWnd,
GWLP_USERDATA);
                    PostThreadMessage(*pdwThreadId, T_MESSAGE, 0, 0);
                    break;
                }
            }
            break;
        }
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
    }
}
```

```

    }
    default:
    {
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}
return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    MSG msg;
    PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, EVENT_NAME);

    if (hEvent != NULL)
    {
        SetEvent(hEvent);
    }
    CloseHandle(hEvent);

    while (GetMessage(&msg, NULL, 0, 0))
    {
        switch (msg.message)
        {
            case T_MESSAGE:
            {
                // TO DO
                // add some message processing here
                MessageBox(NULL, _T("Hi.\nI've just received a message."),
                _T("Thread"), MB_OK);
                break;
            }
            case T_ENDTASK:
            {
                // main thread is exiting
                // return immediately
                return 0L;
            }
        }
    }
    return 0L;
}

```

How it works...

As in our previous example, and every time we want to use custom messages, we need to define them first. We will define the following two messages:

- ▶ `T_MESSAGE` to notify the thread to start its specific task
- ▶ `T_ENDTASK` to notify the thread that the application is closing, and it needs to return immediately

As the message queue for the new thread is not created, due to fact that the thread does not own a window, we need to create it manually. First, we create an event from the thread that will send messages—`producer`. The event won't be set until the newly created thread creates a message queue. We associate the thread ID with the window so that, later, we'll be able to use it from the window procedure, which we'll use for a simple button-click handling.

Now, we wait for the thread to create a message queue. By calling the `PeekMessage` API in the following manner, we are forcing the system to create the message queue:

```
PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);
```

After the message queue is created, we can set the event by calling the `SetEvent` API and close its handle. The thread then enters its message loop. Subsequent calls to `GetMessage` will block the thread until the message is received. When our thread receives a message, it examines the message identifier and displays a message box. In case of `T_MESSAGE`, it displays a message; else, it returns in the case of the `T_ENDTASK` message.

There's more...

We've mentioned `producer` and `consumer` so many times here. Our goal was to point out the widespread occurrence of consumer/producer problems in everyday tasks. Once again, a smart designing of the application is more than half the work done. Remember to perform the application design very carefully, because it is the most important step.

Communicating through the pipe object

Besides message passing, Windows offers other ways to communicate and share data between processes and threads. One of these ways is the **pipe** object. Pipe is the object in memory that the processes use for their communication. The process that creates a pipe is the **pipe server**. A process that connects to a pipe is a **pipe client**. One process writes information to the pipe, and then another process reads the information from the pipe.

We will implement a small client-server example. One process will be a server; another one will be a client. The server will run for an infinite period of time. We will use the pipe object for process communication. Every time the client connects, the server needs to accept the connection and send a welcome message, while the client needs to receive the server's welcome message and make a request to the server again by sending a simple message.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Console application named `pipe_server`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `pipe_server.cpp`.
3. Add the following code to it:

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define BUFFER_SIZE 4096

DWORD WINAPI StartAddress(LPVOID lpParameter);

int _tmain(void)
{
    LPTSTR szPipename = _T("\\\\.\\pipe\\basicMPI");

    while (TRUE)
    {
        _tprintf(_T("basicMPI: waiting client connection\n"));

        HANDLE hPipe = CreateNamedPipe(szPipename, PIPE_ACCESS_DUPLEX,
            PIPE_WAIT | PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE,
            PIPE_UNLIMITED_INSTANCES, BUFFER_SIZE, BUFFER_SIZE, 0,
            NULL);

        if (hPipe == INVALID_HANDLE_VALUE)
        {
            _tprintf(_T("CreateNamedPipe failed! Error code: [%u]\n"),
                GetLastError());
            return 2;
        }

        if (ConnectNamedPipe(hPipe, NULL))
        {
```

```

        _tprintf(_T("Connection succeeded.\n"));

        HANDLE hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)StartAddress, hPipe, 0, NULL);

        CloseHandle(hThread);
    }
}

return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    HANDLE hPipe = (HANDLE)lpParameter;

    PTCHAR szRequest = (PTCHAR)HeapAlloc(GetProcessHeap(), 0,
        BUFFER_SIZE * sizeof(TCHAR));
    PTCHAR szReply = (PTCHAR)HeapAlloc(GetProcessHeap(), 0,
        BUFFER_SIZE * sizeof(TCHAR));
    DWORD dwBytesRead = 0;
    DWORD dwReplyBytes = 0;
    DWORD dwBytesWritten = 0;

    _tprintf(_T("Waiting messages.\n"));

    if (!ReadFile(hPipe, szRequest, BUFFER_SIZE * sizeof(TCHAR),
        &dwBytesRead, NULL))
    {
        _tprintf(_T("ReadFile failed! Error code: [%u]\n"),
            GetLastError());
        return 2L;
    }

    // do some processing

    _tprintf(_T("Client request: \"%s\"\n"), szRequest);

    _tcscpy_s(szReply, BUFFER_SIZE,
        _T("default answer from server"));

    dwReplyBytes = (_tcslen(szReply) + 1) * sizeof(TCHAR);

    if (!WriteFile(hPipe, szReply, dwReplyBytes, &dwBytesWritten,
        NULL))
    {

```

```

        _tprintf(_T("WriteFile failed! Error code: [%u]"),
GetLastError());
        return 2L;
    }

    FlushFileBuffers(hPipe);
    DisconnectNamedPipe(hPipe);
    CloseHandle(hPipe);

    HeapFree(GetProcessHeap(), 0, szRequest);
    HeapFree(GetProcessHeap(), 0, szReply);

    _tprintf(_T("Success.\n"));

    return 0L;
}

```

4. Open **Solution Explorer** and right-click on **Add New Project**. Create a new empty C++ Console application named `pipe_client`.
5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `pipe_client`. Open `pipe_client.cpp`.
6. Add the following code to it:

```

#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define BUFFER_SIZE 4096

int _tmain(void)
{
    TCHAR szBuffer[BUFFER_SIZE];
    DWORD dwRead = 0;
    DWORD dwWritten = 0;
    DWORD dwMode = PIPE_READMODE_MESSAGE;
    LPTSTR szPipename = _T("\\\\.\\pipe\\basicMPI");
    LPTSTR szMessage = _T("Message from client.");
    DWORD dwToWrite = _tcslen(szMessage) * sizeof(TCHAR);

    HANDLE hPipe = CreateFile(szPipename, GENERIC_READ |
        GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

    WaitNamedPipe(szPipename, INFINITE);

    SetNamedPipeHandleState(hPipe, &dwMode, NULL, NULL);

    if (!WriteFile(hPipe, szMessage, dwToWrite, &dwWritten, NULL))

```

```

    {
        _tprintf(_T("WriteFile failed! Error code: [%u]\n"),
            GetLastError());
        return -1;
    }

    _tprintf(
        _T("Message sent to server, receiving reply as follows:\n"));

    while (ReadFile(hPipe, szBuffer, BUFFER_SIZE * sizeof(TCHAR),
        &dwRead, NULL) && GetLastError() != ERROR_MORE_DATA)
    {
        _tprintf(_T("%s"), szBuffer);
    }

    _tprintf(_T("\nSuccess!\nPress ENTER to exit."));
    scanf_s("%c", szBuffer);

    CloseHandle(hPipe);

    return 0;
}

```

How it works...

Let's explain the server first. In order to use the pipe object, we need to create one first using the `CreateNamedPipe` API. The pipe name must be unique. The name must have the `\\.\pipe\pipename` format. Following is an example:

```

HANDLE hPipe = CreateNamedPipe( szPipename, PIPE_ACCESS_DUPLEX, PIPE_
WAIT | PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE, PIPE_UNLIMITED_
INSTANCES, BUFFER_SIZE, BUFFER_SIZE, 0, NULL);

```

For a complete reference of the `CreateNamedPipe` API, refer to MSDN at <http://msdn.microsoft.com/en-us/library/windows/desktop/aa365150%28v=vs.85%29.aspx>.

Our server must run forever in order to process the requests, so we will execute it in an infinite loop followed by a call to `ConnectNamedPipe`. This call enables the pipe server to wait for a client to connect to the pipe. In other words, if the handle to the pipe object was obtained without specifying `FILE_FLAG_OVERLAPPED` in `CreateNamedPipe`, the function does not return until the client is connected. When the client connects, the execution continues. We will create a thread that will process the client request, while the main thread will be ready for the next request concurrently.

Now, let's review request processing. After we've obtained a handle to the pipe, we need to allocate enough heap memory for the request as well as for the response. We will use `ReadFile` to read from the other end of the pipe and then do some processing. The response is sent using `WriteFile` in the following manner:

```
WriteFile(hPipe, szReply, dwReplyBytes, &dwBytesWritten, NULL);
```

The first parameter is the pipe handle. Next goes the address of the buffer where the message is stored. Then, we need to specify the number of bytes we need to write. The address of the buffer for the bytes written comes next. The final parameter is the address of the `OVERLAPPED` object, which we could use for asynchronous operations with `WriteFile`. After we have written to the other end of the pipe, we need to flush the buffers, disconnect from the pipe, and close the pipe handle from here and not from the thread that created the pipe. We do this because the main thread opens the pipe for one client. If another request arrives, while the first isn't processed yet, the main thread will create another thread to process the new request. In such implementations, the thread responsible to process requests will take care of open handles.

Now, let's review the client: We can open the pipe in two ways: using `CreateFile` or the `CallNamedPipe` API. The client needs to wait for the server until the pipe is available. The execution will continue when the server calls `ConnectNamedPipe`, and then the thread gets suspended. We are using `WriteFile` and `ReadFile` in the same manner like we did in the server process. When all the processing is done, we must close the pipe handle.

There's more...

All the techniques shown in this chapter have their advantages and disadvantages. Proper understanding of each of these techniques will give the user the ability to use them in conjunction with each other and create the best possible solutions. The following chapters will demonstrate practical usage of these techniques in order to gain the best results.

5

Thread Synchronization and Concurrent Operations

In this chapter, we will cover the following topics:

- ▶ Pseudoparallelism
- ▶ Process and thread priority
- ▶ The Windows dispatcher object and scheduling
- ▶ Using mutex
- ▶ Using semaphore
- ▶ Using events
- ▶ Using critical section
- ▶ Using pipes

Introduction

In order to understand concurrent execution on different processors, we need to distinguish hardware parallelism and pseudoparallelism. Modern processors have more than one physical core for processing and these cores can process and compute in parallel, while older processors have only one core for processing and it is not always possible to get proper parallel execution.

Furthermore, we will take a short look at Windows scheduling priority, which is necessary when we need to perform tasks with different priorities.

By the end of the chapter, we will use and compare all previously explained synchronization techniques in order to conclude what we have learned so far.

Pseudoparallelism

As we said before, modern processors have more than one physical core, and parallel processing is their native feature. What about older ones? Is it possible to create parallel execution in a single core environment?

Let's give a simple example: suppose we need to calculate a certain mathematical equation. We will calculate the sum that will be implemented as loop with x iterations. In every iteration, certain expressions are calculated.

When we create a program, where one thread will execute a certain task, the calculation time necessary for the task execution is finite. Now let's assume that its value is t . If we split the execution between two threads, where the first thread will calculate the first half of the sum of the iterations and the second thread will calculate the other half of the sum of the iterations, will we get faster execution and the time approximate to $t/2$?

Well, on a single core processor, the answer is *no*, because the operating system schedules both threads for execution, but there is no hardware to support such executions. Since there is only one core, only one thread can be executed on the CPU at any given instant of time.

It is important to understand that creating threads won't always provide benefits. What if we split some task into eight threads and tried to execute the program on a single core CPU? We would only slow down the process because unnecessary thread creation occurs, so division of the task and separate executions will only waste time. If we add synchronization to such a program, we will waste a lot of time on waiting between threads when only one thread can run.

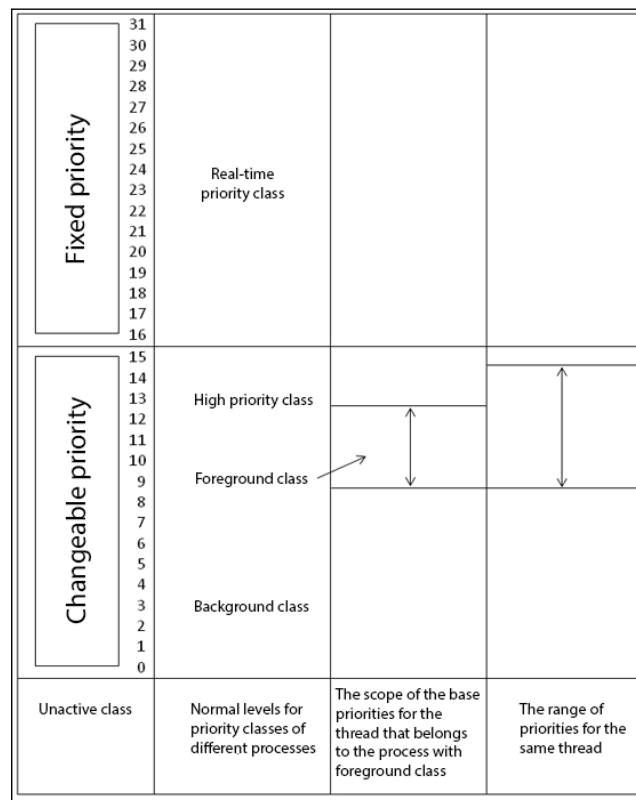
Pseudoparallelism is a very important operating system feature. It gives you an illusion that all tasks (programs) are executed in parallel. For example, while you type in Microsoft Word, you can play Winamp, print a file, and burn DVD at the same time. Either on a single core or on multiple cores, all these tasks are divided into separate threads, which are scheduled by the operating system for execution. Switching quickly among threads, the operating system creates an illusion that all these threads are executed concurrently.

Understanding process and thread priority

To be able to work with threads, you need to know more than just how to create or terminate a thread. It's required that threads interact effectively, which demands proper time control. Time control has two forms: priority and synchronization. Priority controls how often a thread gets the CPU for execution, while synchronization controls thread competition for shared resources, and gives the sequence where threads must perform tasks in a specific order.

When one thread finishes, the scheduler looks for the next thread that will be executed. While selecting the next thread, the threads with higher priority have an advantage. Some activities such as response to sudden power loss are always executed with higher priority. Elements that perform system interrupts have a higher priority than user processes. The bottom line is that every process has a priority rating list, and thread base priority comes from the owning process.

As we've seen before, thread object attributes include the base and dynamic priority. When we use commands for changing priority, we can only change the base priority. Still, it is not possible to change the thread priority by more than two levels, either two levels above or two levels below. That is because the thread can't be more important than its parent process. Though the process can't change its thread priority by more than two steps, the system can. The system allows some kind of improvement—*dynamic priority*—to the threads that need to execute some important operations. When the user performs some action in the application window such as a key stroke or a mouse click, the system always increases the window owning thread priority. When the thread that waits for data from hard drive gets the data, the system increases the priority for that thread too. These temporary priority improvements, when added to base priority, form the dynamic priority. The dispatcher, or the element responsible for scheduling, chooses the next thread based on the dynamic priority. The base and dynamic priority are shown in the following figure:



Decreasing the dynamic priority starts immediately after completion of the task. Dynamic priority decreases one step each time when the thread receives requested time on the CPU, and finally stabilizes on the base priority.

The following example will demonstrate the basic use of the process and thread priority. We will create an application where the main thread will own the main window, while another thread will be responsible for drawing random figures. It is important that the main thread has at least one step higher priority than the drawing thread. This is required because the main thread must respond to any user input, such as moving or minimizing the window or closing the application.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure.

1. Create a new C++ Win32 empty project application named `basic_priority`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `basic_priority`. Open `basic_priority.cpp`.
3. Add the following code:

```
#include <windows.h>
#include <windowsx.h>
#include <time.h>
#include <tchar.h>
#include <math.h>
#include <commctrl.h>

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

#define RESET_EVENT    _T( "__tmp_reset_event__" )
#define WINDOW_WIDTH  800
#define WINDOW_HEIGHT 600

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
void DrawProc(HWND hWnd);
```

```

unsigned GetNextSeed(void);

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__priority_wnd_class__");

    WNDCLASSEX wndEx = { 0 };

    wndEx.cbSize = sizeof(WNDCLASSEX);

    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;

    if (!RegisterClassEx(&wndEx))
    {
        return 1;
    }

    HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, RESET_EVENT);

    InitCommonControls();

    HWND hWnd = CreateWindow(szWindowClass,
        _T("Basic thread priority"), WS_OVERLAPPED | WS_CAPTION
        | WS_SYSMENU | WS_MINIMIZEBOX, 50, 50, WINDOW_WIDTH,
        WINDOW_HEIGHT, NULL, NULL, wndEx.hInstance, NULL);

    if (!hWnd)
    {

```

```
        return NULL;
    }

    HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
        FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
        | FF_MODERN, _T("Microsoft Sans Serif"));

    ShowWindow(hWnd, iWndShow);

    MSG msg;
    while (GetMessage(&msg, 0, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    CloseHandle(hEvent);

    UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

    return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_CREATE:
        {
            DWORD dwThreadId = 0;
            SetThreadPriority(GetCurrentThread(),
                THREAD_PRIORITY_NORMAL);
            HANDLE hThread = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)StartAddress, hWnd,
                0, &dwThreadId); (LONG_PTR)dwThreadId);

            Sleep(100);
            CloseHandle(hThread);
            break;
        }
        case WM_CLOSE:
        {
```

```

        HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE,
            RESET_EVENT);
        SetEvent(hEvent);
        DWORD dwThreadId = (DWORD) GetWindowLongPtr(hWnd,
            GWLP_USERDATA);
        HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE,
            dwThreadId);

        WaitForSingleObject(hThread, INFINITE);
        CloseHandle(hThread);

        DestroyWindow(hWnd);
        break;
    }
    case WM_DESTROY:
    {
        PostQuitMessage(0);
        break;
    }
    default:
    {
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}
return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    HWND hWnd = (HWND)lpParameter;

    SetThreadPriority(GetCurrentThread(),
        THREAD_PRIORITY_BELOW_NORMAL);
    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, RESET_EVENT);

    DWORD dwWaitResult(1);
    while (dwWaitResult != WAIT_OBJECT_0)
    {
        dwWaitResult = WaitForSingleObject(hEvent, 100);
        DrawProc(hWnd);
    }

    CloseHandle(hEvent);
}

```

```
        return 0L;
    }

    void DrawProc(HWND hWnd)
    {
        int iTotal = 10;

        srand(GetNextSeed());

        HDC hDC = GetDC(hWnd);
        if (hDC)
        {
            for (int iCount = 0; iCount < iTotal; iCount++)
            {
                int iStartX = (int)(rand() % WINDOW_WIDTH);
                int iStopX = (int)(rand() % WINDOW_WIDTH);
                int iStartY = (int)(rand() % WINDOW_HEIGHT);
                int iStopY = (int)(rand() % WINDOW_HEIGHT);

                int iRed = rand() & 255;
                int iGreen = rand() & 255;
                int iBlue = rand() & 255;

                HANDLE hBrush = CreateSolidBrush(GetNearestColor(hDC,
RGB(iRed, iGreen, iBlue)));
                HANDLE hbrOld = SelectBrush(hDC, hBrush);

                Rectangle(hDC, min(iStartX, iStopX), max(iStartX, iStopX),
min(iStartY, iStopY), max(iStartY, iStopY));

                DeleteBrush(SelectBrush(hDC, hbrOld));
            }
            ReleaseDC(hWnd, hDC);
        }
    }

    unsigned GetNextSeed(void)
    {
        static unsigned seed = (unsigned)time(NULL);
        return ++seed;
    }
}
```

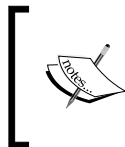
How it works...

When we call the `CreateWindow` or `CreateWindowEx` function, the system sends a `WM_CREATE` message to the window message queue before the function returns. The window procedure of the created window receives a message only after the window is created and also before the window becomes visible. That's why we need to create an event before a call to `CreateWindow` is made.

When `WindowProcedure` receives a `WM_CREATE` message, we will set the main thread priority to `NORMAL` (which is default) and then we'll create another thread, responsible for drawing. After that thread is created, we will associate its identifier into the window section for user-specific data that belongs to the window by using the following code:

```
SetWindowLongPtr(hWnd, GWLP_USERDATA, (LONG_PTR) &dwThreadId);
```

Then main thread waits for 0.1 second in order for a newly created thread to start execution, and after that its handle can be closed. Now, let's examine the `StartAddress` thread routine. The main window handle is passed as an argument, so we need to cast the `LPVOID` type to the `HWND` type. Now, we will set the thread priority to one step below normal (`THREAD_PRIORITY_BELOW_NORMAL`) because it is necessary that the main thread has at least one step higher priority than the drawing thread. Let's explain this: suppose a user expects an application to be responsive all the time. In this example, our working thread has a very simple job to execute: it draws rectangles, with a 0.1 second pause between each 10 rectangles. In another situation, where the working thread should calculate a **Fourier series** with large number of iterations for example, the application won't be responsive if the thread that calculates has equal or higher priority than the main thread. This happens because the dispatcher selects the next thread for execution, based on its priority and scheduling algorithm. If the main thread has less priority than the working thread, then the working thread has more priority and the application won't be responsive to the user input immediately. It will still process the user input, but after the working thread gets its time on CPU, which will result in slow application behavior. After setting its priority, the thread enters an infinite loop where it draws rectangles until the user closes the application.



A Fourier series is a representation of a periodic function usually presented as an infinite sum of sines and cosines. It has many applications in cryptography, speech recognition, handwriting, and so on.

Let's now focus on the `DrawProc` drawing routine. After calling the `srand` function to reseed the random generator, we need to obtain a device context to draw in. Then we will take random coordinates, which are within the boundaries of the application window and a random color to draw an actual rectangle. The `GetNextSeed` routine sets the seed value to the current timestamp, when called for the first time, as static variables are initialized only once, and after that, we'll simply increment its value by one ensuring a unique value each time.

When the user wants to close the application, we will open an event and set its state to `signaled`. In that way, the thread will exit on the next call to `WaitForSingleObject`. Then we will obtain the working thread identifier that is associated with the main window and wait for its exit using the `WaitForSingleObject` wait function.

There's more...

We may conclude that if we set high priority for a thread, the application will execute faster. Well, this is incorrect. Pay attention that if you increase the priority for various applications (whether it is process or thread priority) a lot, the system will devote a lot of time for those threads, but it won't be able to perform its own tasks, resulting in a slow or even unresponsive system. If you open the **Task Manager**, you will see that only few windows' processes are running on a higher priority than `NORMAL`. Usually those are `dwm.exe` (Desktop Window Manager) and `taskmgr.exe` (Windows Task Manager). Windows too sets desktop processes with higher priority to be able to respond to user commands, just like our application that prefers user input over drawing (calculating). You must use priority with caution, and our advice is to increase priority only when you must.

The Windows dispatcher object and scheduling

As we've previously learned, in order to choose the next thread for execution, the dispatcher selects threads from the queue, starting from a higher priority down to lower priority. However, the queue does not contain all the threads. Some of them are suspended or blocked. We previously said that there are three states, **Running**, **Ready**, and **Blocked**. These were basic condition states. Now, let's expand these states and explain them all.

At any instant, a thread can be in one of the following states:

- ▶ **Ready**: It resides in the queue waiting for execution
- ▶ **Standby**: It is next for execution
- ▶ **Running**: It is executing on the CPU
- ▶ **Waiting**: It isn't executing—it is waiting for a resume signal
- ▶ **Transition**: It is to be executed just as soon as the system loads its context
- ▶ **Terminated**: It is done executing, but the object hasn't been deleted yet

When dispatcher selects the next thread for execution, the system loads its context into memory. The context includes a series of values such as machine register values, kernel stack, environment block, and user stack residing in the process address space, even though a part of the context was swapped to the disk, thread enters transition state, while the system collects parts. Switching threads means saving all parts of the context and loading all context parts of the next thread.

The newly loaded thread executes some part of the time, which is called a **quantum**. The system maintains a counter that measures the current time. For each clock tick the system decrements the counter by a certain value. When the counter reaches zero, the dispatcher performs context switching, and sets up the next thread for execution.

Using mutex

One of the Windows synchronization objects is *mutex*. We use mutex when we want to acquire exclusive access to some shared object. We also use mutex when we want to protect some part of the code, to be executed by one thread at a time.

In our following example, we will ask the user to provide a text file with six integer coordinates on each line in order to calculate a system of two linear equations:

$$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases}$$

We will also use the `CQueue` class, previously implemented in *Chapter 1, Introduction to C++ Concepts and Features*. Our task is to load all the values from the file, and then each group of coordinates will be added to the queue. When each group is loaded, we will start some number of threads to calculate concurrently. Important decisions to make here are how many threads we should start, and what areas of code we should protect and synchronize.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Windows application named `concurrent_operations`.
2. Open **Solution Explorer** and right-click on **Header file**. Add the existing header file, used in *Chapter 1, Introduction to C++ Concepts and Features*, named `CQueue.h`.
3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `main`. Open `main.h`.

4. Add the following code:

```
#pragma once

#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <psapi.h>
#include <strsafe.h>
#include <string.h>
#include <sstream>
#include <fstream>
#include <commdlg.h>
#include "CQueue.h"

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \\  
name='Microsoft.Windows.Common-Controls' \\  
version='6.0.0.0' processorArchitecture='*' \\  
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

using namespace std;

#define CONTROL_BROWSE 100
#define CONTROL_START 101
#define CONTROL_RESULT 103
#define CONTROL_TEXT 104
#define CONTROL_PROGRESS 105

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM  
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPSTR szFileName);

typedef struct
{
    int iA1;
    int iB1;
    int iC1;
    int iA2;
    int iB2;
    int iC2;
    HWND hWndProgress;
    HWND hWndResult;
} QueueElement, *PQueueElement;
```

5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.
6. Add the following code:

```
#pragma once

/*****
 * You need to specify text file with coordinates.      *
 * Our example will solve system of 2 linear equations  *
 * a1x + b1y = c1;                                     *
 * a2x + b2y = c2;                                     *
 *                                                     *
 * Each line should have exactly six coordinates.      *
 * File format example:                               *
 *                                                     *
 * 4 -3 2 -7 5 6                                       *
 * 12 1 -7 9 -5 8                                     *
 *                                                     *
 * Avoid new line at the end of the file.              *
 *****/
#include "main.h"

char szFilePath[MAX_PATH] = { 0 };
char szResult[4096];
CQueue<QueueElement> queue;
TCHAR* szMutex = _T("__mutex_132__");

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev,
    LPTSTR szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__concurrent_operations__");
    WNDCLASSEX wndEx = { 0 };

    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
```

```
wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndEx.lpszMenuName = NULL;
wndEx.lpszClassName = szWindowClass;
wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wndEx))
{
    return 1;
}

InitCommonControls();

HWND hWnd = CreateWindow(szWindowClass,
    _T("Concurrent operations"), WS_OVERLAPPED
    | WS_CAPTION | WS_SYSMENU, 50, 50, 305, 250, NULL, NULL,
    wndEx.hInstance, NULL);

if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
    | FF_MODERN, _T("Microsoft Sans Serif"));

HWND hLabel = CreateWindow(_T("STATIC"),
    _T(" Press \"Browse\" and choose file with coordinates: "),
    WS_CHILD | WS_VISIBLE | SS_CENTERIMAGE | SS_LEFT | WS_BORDER,
    20, 20, 250, 25, hWnd, (HMENU)CONTROL_TEXT, wndEx.hInstance,
    NULL);
SendMessage(hLabel, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hResult = CreateWindow(_T("STATIC"), _T(""), WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 20, 65, 150, 75, hWnd,
    (HMENU)CONTROL_RESULT, wndEx.hInstance, NULL);
SendMessage(hResult, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hBrowse = CreateWindow(_T("BUTTON"), _T("Browse"), WS_CHILD
    | WS_VISIBLE
```

```

        | BS_PUSHBUTTON, 190, 65, 80, 25, hWnd, (HMENU)CONTROL_BROWSE,
wndEx.hInstance, NULL);
        SendMessage(hBrowse, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hStart = CreateWindow(_T("BUTTON"), _T("Start"), WS_CHILD |
WS_VISIBLE
        | BS_PUSHBUTTON, 190, 115, 80, 25, hWnd, (HMENU)CONTROL_START,
wndEx.hInstance, NULL);
        SendMessage(hStart, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hProgress = CreateWindow(PROGRESS_CLASS, _T(""), WS_CHILD |
WS_VISIBLE
        | WS_BORDER, 20, 165, 250, 25, hWnd, (HMENU)CONTROL_PROGRESS,
wndEx.hInstance, NULL);
        SendMessage(hProgress, PBM_SETSTEP, (WPARAM)1, 0);
        SendMessage(hProgress, PBM_SETPOS, (WPARAM)0, 0);

        ShowWindow(hWnd, iWndShow);

        HANDLE hMutex = CreateMutex(NULL, FALSE, szMutex);

        MSG msg;
        while (GetMessage(&msg, 0, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        CloseHandle(hMutex);
        UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

        return (int)msg.wParam;
    }

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_COMMAND:

```

```
{
    switch (LOWORD(wParam))
    {
        case CONTROL_BROWSE:
        {
            if (!FileDialog(hWnd, szFilePath))
            {
                MessageBox(hWnd,
                    _T("You must choose valid file path!"),
                    _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
            }
            else
            {
                char szBuffer[MAX_PATH];
                wsprintfA(szBuffer,
                    "\n File: %s Press \"Start\" now.",
                    strrchr(szFilePath, '\\') + 1);

                SetWindowTextA(GetDlgItem(hWnd, CONTROL_TEXT),
szBuffer);
            }
            break;
        }
        case CONTROL_START:
        {
            if (!*szFilePath)
            {
                MessageBox(hWnd,
                    _T("You must choose valid file path first!"),
                    _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
                break;
            }

            ifstream infile(szFilePath);

            if (infile.is_open())
            {
                string line;
                while (std::getline(infile, line))
                {
                    QueueElement* pQElement = new QueueElement();
                    istringstream iss(line);

                    if (!(iss >> pQElement->iA1 >> pQElement->iB1 >>
pQElement->iC1 >> pQElement->iA2 >> pQElement->iB2 >> pQElement-
>iC2))

```

```

        {
            break;
        }

        pQElement->hWndProgress = GetDlgItem(hWnd,
            CONTROL_PROGRESS);
        pQElement->hWndResult = GetDlgItem(hWnd,
            CONTROL_RESULT);

        queue.Enqueue(pQElement);
    }
    infile.close();

    SendMessage(GetDlgItem(hWnd, CONTROL_PROGRESS),
        PBM_SETRANGE, 0, MAKELPARAM(0, queue.Count()));

    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);
    for (DWORD dwIndex = 0; dwIndex <
        sysInfo.dwNumberOfProcessors; dwIndex++)
    {
        HANDLE hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)StartAddress, &queue,
            0, NULL);
        SetThreadIdealProcessor(hThread, dwIndex);

        Sleep(100);
        CloseHandle(hThread);
    }
}
else
{
    MessageBox(hWnd, _T("Cannot open file!"),
        _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
}
break;
}
}
break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}

```

```
        return 0;
    }

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    // We'll solve the linear system using Cramer's rule:
    CQueue<QueueElement>* pQueue = (CQueue<QueueElement>*)
lpParameter;

    HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, szMutex);
    QueueElement* pQElement = NULL;

    while (true)
    {
        WaitForSingleObject(hMutex, INFINITE);

        pQElement = pQueue->Dequeue();

        ReleaseMutex(hMutex);

        if (pQElement == NULL)
        {
            break;
        }

        char szBuffer[1024];

        double dDeterminant = (pQElement->iA1 * pQElement->iB2) -
(pQElement->iB1 * pQElement->iA2);

        if (dDeterminant != 0)
        {
            double dX = ((pQElement->iC1 * pQElement->iB2) -
                (pQElement->iB1 * pQElement->iC2)) / dDeterminant;
            double dY = ((pQElement->iA1 * pQElement->iC2) -
                (pQElement->iC1 * pQElement->iA2)) / dDeterminant;

            sprintf_s(szBuffer, "  x = %8.4lf,\nty = %8.4lf\n", dX, dY);
        }
        else
        {
            sprintf_s(szBuffer, "  Determinant is zero.\n");
        }
    }
}
```

```

    }

    strcat_s(szResult, 4096, szBuffer);
    SetWindowTextA(pQElement->hWndResult, szResult);

    SendMessage(pQElement->hWndProgress, PBM_STEPIT, 0, 0);
    delete pQElement;

    Sleep(1000);
}

CloseHandle(hMutex);
return 0L;
}

BOOL FileDialog(HWND hWnd, LPSTR szFileName)
{
    OPENFILENAMEA ofn;

    char szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));

    ofn.lStructSize = sizeof(ofn);
    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = "All\\0*.*\\0Text\\0*.TXT\\0";
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        strcpy_s(szFileName, MAX_PATH - 1, szFile);
        return TRUE;
    }

    return FALSE;
}

```


How it works...

In the file `main.h`, we have defined the `QueueElement` structure in order to load each group of coordinates to the queue more easily. In `main.cpp`, we have declared a few global variables: `szFilePath` for the file that the user will select; `szResult` for displaying messages upon calculation; `queue` for the actual queue; and `szMutex` that represents the mutex name for later synchronization. At the application entry point (`_tWinMain`), we initialize and create the window, with the `hLabel` control, which we will use for displaying the messages about user interaction; the `hResult` control, which threads will use for displaying the calculation results; the `hBrowse` control, which the user will use in order to select the file; the `hStart` control, which the user will use to start the calculation; and the `hProgress` control, which the application will use to display the calculation progress.

Before the main thread enters the message loop, it will create a mutex object. When the user clicks on **Browse**, we will display the **Open file** dialog box for the user to choose the actual file. When the user clicks on the **Start** button, we will check if the file was previously selected. If so, we will read one line at a time from the file and initialize the `QueueElement` object. The `QueueElement` object expects coordinates, and provides the handles to the `progress` control and the `result` control. These handles will be used by the threads in order to display the progress of the calculation and result respectively.

As we said so many times before, proper application design leads to getting adequate results and reduces the possibility of errors and bugs. The first question that we need to take care of is how many threads can we start? The next question would be what areas of the code do we need to protect from simultaneous access? Or for which statements do we need to acquire exclusive access? The answer to first question would be: if we have less concurrent tasks to execute than the number of processors available, we could start as many threads as there are tasks. However, the user can provide a file with an unlimited number of lines with coordinates. If more tasks than the number of available processors are provided, the system won't perform so well, and our application won't have the best performance. So when we load all the objects into the queue, we will query the system information for the available number of CPUs. Then, we will limit the number of threads in a way not to exceed the number of available processors, as shown in the following code:

```
SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);

for (DWORD dwIndex = 0; dwIndex < sysInfo.dwNumberOfProcessors;
     dwIndex++)
{
    HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
    StartAddress, &queue, 0, NULL);
    SetThreadIdealProcessor(hThread, dwIndex);
}
```

```
Sleep( 100 );  
CloseHandle( hThread );  
}
```

Well, we could also query the queue count so as to not start more threads than necessary, but you'll get the point. We are also using the `SetThreadIdealProcessor` API in order to tell the system to schedule the thread to the selected processor. This routine specifies a thread ideal processor and provides a hint to the scheduler about the preferred processor for a thread. The scheduler runs the thread on the thread's ideal processor when possible, but without guaranteeing about a given processor number. In a situation where the selected processor is busy and our thread is next for execution, the system will schedule it on the next available core.

The answer to the second question is as follows: what is the shared object that all threads use? Well that would certainly be the `queue` object. The `CQueue::Dequeue` operation is performed from each thread in parallel. That place would be an application bottleneck without proper synchronization, so we must take care of it. We use a mutex object when we want to gain exclusive access to one thread. That is perfect for our `Dequeue` operation because at any instant of time, only one thread should obtain an object from the queue. By using the mutex object, we are providing exclusive access to the shared object, in this example `queue`, and its operation `Dequeue`.

There's more...

This was a simple example only to demonstrate how to properly approach the application from design to the implementation itself. Even though *Chapter 7, Understanding Concurrent Code Design*, will explain concurrent code design in detail, it is important for the user to be familiar with parallel execution and get the habit to think properly before starting coding.

Using semaphore

Another Windows synchronization object is *semaphore*. Semaphore works as a narrow gate, which a certain number of threads can pass through at any instant of time. We will use our previous example again with slight changes to demonstrate various approaches with different synchronization objects and techniques while achieving the same results.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure.

1. Create a new empty C++ Windows application named `concurrent_operations2`.
2. Open **Solution Explorer** and right-click on **Header file**. Add the existing header file used in *Chapter 1, Introduction to C++ Concepts and Features*, named `CQueue.h`.
3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `main.h`. Open `main.h`.
4. Add the following code:

```
#pragma once

#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <psapi.h>
#include <strsafe.h>
#include <string.h>
#include <sstream>
#include <fstream>
#include <commdlg.h>
#include "CQueue.h"

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \\  
name='Microsoft.Windows.Common-Controls' \\  
version='6.0.0.0' processorArchitecture='*' \\  
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

using namespace std;

#define CONTROL_BROWSE 100
#define CONTROL_START 101
#define CONTROL_RESULT 103
#define CONTROL_TEXT 104
#define CONTROL_PROGRESS 105

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM  
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPSTR szFileName);

typedef struct
```

```

{
    int iA1;
    int iB1;
    int iC1;
    int iA2;
    int iB2;
    int iC2;
    HWND hWndProgress;
    HWND hWndResult;
} QueueElement, *PQueueElement;

```

5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.
6. Add the following code:

```

#pragma once

/*****
 * You need to specify text file with coordinates.
 * Our example will solve system of 2 linear equations
 * a1x + b1y = c1;
 * a2x + b2y = c2;
 *
 * Each line should have exactly six coordinates.
 * File format example:
 *
 * 4 -3 2 -7 5 6
 * 12 1 -7 9 -5 8
 *
 * Avoid new line at the end of the file.
 *****/
#include "main.h"

char szFilePath[MAX_PATH] = { 0 };
char szResult[4096];
CQueue<QueueElement> queue;
TCHAR* szMutex = _T("__mutex_165__");
TCHAR* szSemaphore = _T("__semaphore_456__");

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);

```

```
UNREFERENCED_PARAMETER(szCommandLine);

TCHAR* szWindowClass = _T("__concurrent_operations__");

WNDCLASSEX wndEx = { 0 };
wndEx.cbSize = sizeof(WNDCLASSEX);
wndEx.style = CS_HREDRAW | CS_VREDRAW;
wndEx.lpfnWndProc = WindowProcedure;
wndEx.cbClsExtra = 0;
wndEx.cbWndExtra = 0;
wndEx.hInstance = hThis;
wndEx.hIcon = LoadIcon(wndEx.hInstance,
MAKEINTRESOURCE(IDI_APPLICATION));
wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndEx.lpszMenuName = NULL;
wndEx.lpszClassName = szWindowClass;
wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wndEx))
{
    return 1;
}

InitCommonControls();

HWND hWnd = CreateWindow(szWindowClass,
    _T("Concurrent operations"), WS_OVERLAPPED
    | WS_CAPTION | WS_SYSMENU, 50, 50, 305, 250, NULL, NULL,
    wndEx.hInstance, NULL);

if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
    | FF_MODERN, _T("Microsoft Sans Serif"));

HWND hLabel = CreateWindow(_T("STATIC"),
    _T(" Press \"Browse\" and choose file with coordinates: "),
```

```

        WS_CHILD | WS_VISIBLE | SS_CENTERIMAGE | SS_LEFT | WS_BORDER,
        20, 20, 250, 25, hWnd, (HMENU)CONTROL_TEXT, wndEx.hInstance,
        NULL);

SendMessage(hLabel, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hResult = CreateWindow(_T("STATIC"), _T(""), WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 20, 65, 150, 75, hWnd,
    (HMENU)CONTROL_RESULT, wndEx.hInstance, NULL);
SendMessage(hResult, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hBrowse = CreateWindow(_T("BUTTON"), _T("Browse"), WS_CHILD
| WS_VISIBLE
    | BS_PUSHBUTTON, 190, 65, 80, 25, hWnd, (HMENU)CONTROL_BROWSE,
wndEx.hInstance, NULL);
SendMessage(hBrowse, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hStart = CreateWindow(_T("BUTTON"), _T("Start"), WS_CHILD |
WS_VISIBLE
    | BS_PUSHBUTTON, 190, 115, 80, 25, hWnd, (HMENU)CONTROL_START,
wndEx.hInstance, NULL);
SendMessage(hStart, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hProgress = CreateWindow(PROGRESS_CLASS, _T(""), WS_CHILD |
WS_VISIBLE
    | WS_BORDER, 20, 165, 250, 25, hWnd, (HMENU)CONTROL_PROGRESS,
wndEx.hInstance, NULL);
SendMessage(hProgress, PBM_SETSTEP, (WPARAM)1, 0);
SendMessage(hProgress, PBM_SETPOS, (WPARAM)0, 0);

ShowWindow(hWnd, iWndShow);

HANDLE hMutex = CreateMutex(NULL, FALSE, szMutex);

SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);
LONG lProcessorCount = (LONG)sysInfo.dwNumberOfProcessors;
HANDLE hSemaphore = CreateSemaphore(NULL, lProcessorCount,
lProcessorCount, szSemaphore);

MSG msg;
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);

```

```
        DispatchMessage(&msg);
    }

    CloseHandle(hSemaphore);
    CloseHandle(hMutex);
    UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

    return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case CONTROL_BROWSE:
                {
                    if (!FileDialog(hWnd, szFilePath))
                    {
                        MessageBox(hWnd,
                            _T("You must choose valid file path!"),
                            _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
                    }
                    else
                    {
                        char szBuffer[MAX_PATH];
                        wsprintfA(szBuffer,
                            "\n File: %s Press \"Start\" now.",
                            strrchr(szFilePath, '\\') + 1);

                        SetWindowTextA(GetDlgItem(hWnd, CONTROL_TEXT),
szBuffer);
                    }
                    break;
                }
                case CONTROL_START:
```

```

{
    if (!*szFilePath)
    {
        MessageBox(hWnd,
            _T("You must choose valid file path first!"),
            _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
        break;
    }

    ifstream infile(szFilePath);

    if (infile.is_open())
    {
        string line;
        while (std::getline(infile, line))
        {
            QueueElement* pQElement = new QueueElement();

            istringstream iss(line);

            if (!(iss >> pQElement->iA1 >> pQElement->iB1 >>
pQElement->iC1 >> pQElement->iA2 >> pQElement->iB2 >> pQElement-
>iC2))
            {
                break;
            }

            pQElement->hWndProgress = GetDlgItem(hWnd,
                CONTROL_PROGRESS);
            pQElement->hWndResult = GetDlgItem(hWnd,
                CONTROL_RESULT);

            queue.Enqueue(pQElement);
        }

        infile.close();

        SendMessage(GetDlgItem(hWnd, CONTROL_PROGRESS),
            PBM_SETRANGE, 0, MAKELPARAM(0, queue.Count()));

        int iCount = queue.Count();

        for (int iIndex = 0; iIndex < iCount; iIndex++)
        {
            HANDLE hThread = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)StartAddress, &queue,
                0, NULL);

```



```
        Sleep(100);
        CloseHandle(hThread);
    }
}
else
{
    MessageBox(hWnd, _T("Cannot open file!"),
_T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
}
break;
}
}
break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    // We'll solve the linear system using Cramer's rule:
    CQueue<QueueElement>* pQueue = (CQueue<QueueElement>*)
lpParameter;

    HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, szMutex);
    HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
szSemaphore);
    QueueElement* pQElement = NULL;

    while (true)
    {
        WaitForSingleObject(hSemaphore, INFINITE);
        ReleaseSemaphore(hSemaphore, 1, NULL);

        WaitForSingleObject(hMutex, INFINITE);
        pQElement = pQueue->Dequeue();
        ReleaseMutex(hMutex);

        if (pQElement == NULL)
        {
            break;
        }
    }
}
```

```

    }

    char szBuffer[1024];

    double dDeterminant = (pQElement->iA1 * pQElement->iB2) -
        (pQElement->iB1 * pQElement->iA2);

    if (dDeterminant != 0)
    {
        double dX = ((pQElement->iC1 * pQElement->iB2) -
            (pQElement->iB1 * pQElement->iC2)) / dDeterminant;
        double dY = ((pQElement->iA1 * pQElement->iC2) -
            (pQElement->iC1 * pQElement->iA2)) / dDeterminant;

        sprintf_s(szBuffer, " x = %8.4lf, \ty = %8.4lf\n", dX, dY);
    }
    else
    {
        sprintf_s(szBuffer, " Determinant is zero.\n");
    }

    strcat_s(szResult, 4096, szBuffer);
    SetWindowTextA(pQElement->hWndResult, szResult);

    SendMessage(pQElement->hWndProgress, PBM_STEPIT, 0, 0);

    delete pQElement;

    Sleep(1000);
}

CloseHandle(hSemaphore);
CloseHandle(hMutex);

return 0L;
}

BOOL FileDialog(HWND hWnd, LPSTR szFileName)
{
    OPENFILENAMEA ofn;

    char szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));

    ofn.lStructSize = sizeof(ofn);

```

```
    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = "All\\0*.*\\0Text\\0*.TXT\\0";
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        strcpy_s(szFileName, MAX_PATH - 1, szFile);
        return TRUE;
    }

    return FALSE;
}
```

How it works...

This is a slightly different approach than the one used in our previous example. Instead of creating as many threads as there are processors available, we instantiate a semaphore object where the initial as well as the maximum count of parallel operations will not exceed the number of CPUs. So it is similar; the only difference is the technique used and approach selected. We have asked the system for the semaphore object inside the `StartAddress` routine. After successful operation, we are releasing the semaphore by decrementing it by one:

```
WaitForSingleObject( hSemaphore, INFINITE );
ReleaseSemaphore( hSemaphore, 1, NULL );
```

There's more...

Again, the most important part was to properly select the synchronization spots as well as proper synchronization object selection. We've intentionally selected semaphore in order to demonstrate to the user that the first solution isn't always the only solution. Fortunately, Windows provides a variety of synchronization objects, and developers should be aware of each one of them in order to select the most appropriate one in their development.

Using event

The event synchronization object works as a signal for alerting the thread in a certain manner. It can be used to start or stop an execution, or when a user input is performed, and so on.

We will use the same example as in our previous two topics, with improvement to application behavior by adding an event that will signal the threads that the user wants to abort the execution and end the program. In this way, we will make our application even more user friendly and more responsive to user input.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure.

1. Create a new empty C++ Windows application named `concurrent_operations3`.
2. Open **Solution Explorer** and right-click on **Header file**. Add an existing header file, used in *Chapter 1, Introduction to C++ Concepts and Features*, named `CQueue.h`.
3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `main`. Open `main.h`.
4. Add the following code:

```
#pragma once

#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <psapi.h>
#include <strsafe.h>
#include <string.h>
#include <sstream>
#include <fstream>
#include <commdlg.h>
#include "CQueue.h"

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \\  
name='Microsoft.Windows.Common-Controls' \\  
version='6.0.0.0' processorArchitecture='*' \\  
publicKeyToken='6595b64144ccf1df' language='*'\\"" )
```

```
using namespace std;

#define CONTROL_BROWSE 100
#define CONTROL_START 101
#define CONTROL_RESULT 103
#define CONTROL_TEXT 104
#define CONTROL_PROGRESS 105

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPSTR szFileName);

typedef struct
{
    int iA1;
    int iB1;
    int iC1;
    int iA2;
    int iB2;
    int iC2;
    HWND hWndProgress;
    HWND hWndResult;
} QueueElement, *PQueueElement;
```

5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named main. Open main.cpp.
6. Add the following code:

```
#pragma once

/*****
 * You need to specify text file with coordinates.      *
 * Our example will solve system of 2 linear equations  *
 * a1x + b1y = c1;                                       *
 * a2x + b2y = c2;                                       *
 *                                                       *
 * Each line should have exactly six coordinates.      *
 * File format example:                                  *
 *                                                       *
 * 4 -3 2 -7 5 6                                         *
 * 12 1 -7 9 -5 8                                       *
 *                                                       *
 * Avoid new line at the end of the file.              *
 *****/
```

```

#include "main.h"

char szFilePath[MAX_PATH] = { 0 };
char szResult[4096];
CQueue<QueueElement> queue;
TCHAR* szEvent = _T("__event_879__");
TCHAR* szMutex = _T("__mutex_132__");

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__concurrent_operations__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
    wndEx.lpszMenuName = NULL;
    wndEx.lpszClassName = szWindowClass;
    wndEx.hIconSm = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));

    if (!RegisterClassEx(&wndEx))
    {
        return 1;
    }

    InitCommonControls();

    HWND hWnd = CreateWindow(szWindowClass,
        _T("Concurrent operations"), WS_OVERLAPPED
        | WS_CAPTION | WS_SYSMENU, 50, 50, 305, 250, NULL, NULL,
        wndEx.hInstance, NULL);

```

```
if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
    | FF_MODERN, _T("Microsoft Sans Serif"));

HWND hLabel = CreateWindow(_T("STATIC"),
    _T(" Press \"Browse\" and choose file with coordinates: "),
    WS_CHILD | WS_VISIBLE | SS_CENTERIMAGE | SS_LEFT | WS_BORDER,
    20, 20, 250, 25, hWnd, (HMENU)CONTROL_TEXT, wndEx.hInstance,
    NULL);

SendMessage(hLabel, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hResult = CreateWindow(_T("STATIC"), _T(""), WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 20, 65, 150, 75, hWnd,
    (HMENU)CONTROL_RESULT, wndEx.hInstance, NULL);
SendMessage(hResult, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hBrowse = CreateWindow(_T("BUTTON"), _T("Browse"), WS_CHILD
| WS_VISIBLE
    | BS_PUSHBUTTON, 190, 65, 80, 25, hWnd, (HMENU)CONTROL_BROWSE,
wndEx.hInstance, NULL);
SendMessage(hBrowse, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hStart = CreateWindow(_T("BUTTON"), _T("Start"), WS_CHILD |
WS_VISIBLE
    | BS_PUSHBUTTON, 190, 115, 80, 25, hWnd, (HMENU)CONTROL_START,
wndEx.hInstance, NULL);
SendMessage(hStart, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hProgress = CreateWindow(PROGRESS_CLASS, _T(""), WS_CHILD
    | WS_VISIBLE | WS_BORDER, 20, 165, 250, 25, hWnd,
    (HMENU)CONTROL_PROGRESS, wndEx.hInstance, NULL);
SendMessage(hProgress, PBM_SETSTEP, (WPARAM)1, 0);
SendMessage(hProgress, PBM_SETPOS, (WPARAM)0, 0);

ShowWindow(hWnd, iWndShow);

HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, szEvent);
HANDLE hMutex = CreateMutex(NULL, FALSE, szMutex);
```

```
MSG msg;
while (GetMessage(&msg, 0, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

CloseHandle(hMutex);
CloseHandle(hEvent);
UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

return (int)msg.wParam;
}

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
        {
            PostQuitMessage(0);
            break;
        }
        case WM_CLOSE:
        {
            HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, szEvent);
            SetEvent(hEvent);
            CloseHandle(hEvent);

            DestroyWindow(hWnd);
            break;
        }
        case WM_COMMAND:
        {
            switch (LOWORD(wParam))
            {
                case CONTROL_BROWSE:
                {
                    if (!FileDialog(hWnd, szFilePath))
                    {
                        MessageBox(hWnd,
```



```
        _T("You must choose valid file path!"),
        _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
    }
    else
    {
        char szBuffer[MAX_PATH];
        wsprintfA(szBuffer,
            "\n File: %s Press \"Start\" now.",
            strrchr(szFilePath, '\\') + 1);

        SetWindowTextA(GetDlgItem(hWnd, CONTROL_TEXT),
szBuffer);
    }
    break;
}
case CONTROL_START:
{
    if (!*szFilePath)
    {
        MessageBox(hWnd,
            _T("You must choose valid file path first!"),
            _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
        break;
    }

    ifstream infile(szFilePath);

    if (infile.is_open())
    {
        string line;

        while (std::getline(infile, line))
        {
            QueueElement* pQElement = new QueueElement();

            istringstream iss(line);

            if (!(iss >> pQElement->iA1 >> pQElement->iB1 >>
pQElement->iC1 >> pQElement->iA2 >> pQElement->iB2 >> pQElement-
>iC2))
            {
                break;
            }
        }
    }
}
```

```

        pQElement->hWndProgress = GetDlgItem(hWnd,
            CONTROL_PROGRESS);
        pQElement->hWndResult = GetDlgItem(hWnd,
            CONTROL_RESULT);

        queue.Enqueue(pQElement);
    }

    infile.close();

    SendMessage(GetDlgItem(hWnd, CONTROL_PROGRESS),
        PBM_SETRANGE, 0, MAKELPARAM(0, queue.Count()));

    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);

    for (DWORD dwIndex = 0; dwIndex <
        sysInfo.dwNumberOfProcessors; dwIndex++)
    {
        HANDLE hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)StartAddress, &queue,
            0, NULL);
        SetThreadIdealProcessor(hThread, dwIndex);
        Sleep(100);
        CloseHandle(hThread);
    }
}
else
{
    MessageBox(hWnd, _T("Cannot open file!"),
        _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
}
break;
}
}
break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

```

```
    }
    return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    // We'll solve the linear system using Cramer's rule:
    CQueue<QueueElement>* pQueue = (CQueue<QueueElement>*)
lpParameter;

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, szEvent);
    HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, szMutex);
    QueueElement* pQElement = NULL;

    while (true)
    {
        DWORD dwStatus = WaitForSingleObject(hEvent, 10);
        if (dwStatus == WAIT_OBJECT_0)
        {
            break;
        }

        WaitForSingleObject(hMutex, INFINITE);
        pQElement = pQueue->Dequeue();
        ReleaseMutex(hMutex);

        if (pQElement == NULL)
        {
            break;
        }

        char szBuffer[1024];
        double dDeterminant = (pQElement->iA1 * pQElement->iB2) -
(pQElement->iB1 * pQElement->iA2);
        if (dDeterminant != 0)
        {
            double dX = ((pQElement->iC1 * pQElement->iB2) -
                (pQElement->iB1 * pQElement->iC2)) / dDeterminant;
            double dY = ((pQElement->iA1 * pQElement->iC2) -
                (pQElement->iC1 * pQElement->iA2)) / dDeterminant;
            sprintf_s(szBuffer, "  x = %8.4lf,\nty = %8.4lf\n", dX, dY);
        }
        else
        {

```

```

        sprintf_s(szBuffer, " Determinant is zero.\n");
    }

    strcat_s(szResult, 4096, szBuffer);
    SetWindowTextA(pQElement->hWndResult, szResult);

    SendMessage(pQElement->hWndProgress, PBM_STEPIT, 0, 0);

    delete pQElement;

    Sleep(1000);
}

CloseHandle(hMutex);
CloseHandle(hEvent);
return 0L;
}

BOOL FileDialog(HWND hWnd, LPSTR szFileName)
{
    OPENFILENAMEA ofn;

    char szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));

    ofn.lStructSize = sizeof(ofn);
    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = "All\0*.*\0Text\0*.TXT\0";
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        strcpy_s(szFileName, MAX_PATH - 1, szFile);
        return TRUE;
    }

    return FALSE;
}

```

How it works...

Just as before, we are using a mutex object for synchronization. Only this time we are adding an event in order to be able to signal to the thread that some action has occurred. That action is simple: the user wants to end the application. However, this will give the user a wider picture of how to use the event object in a multithreading environment—very important for its future development.

So, before the main thread enters the message loop, we will create an event. At the beginning of the `StartAddress` thread, we will call `WaitForSingleObject` with a wait time of 10 milliseconds, only to give the thread the space to query the object signaled state: if it is signaled, the thread must exit; if not, it will continue execution.

There's more...

Events are very important for a multithreaded environment. Besides synchronization, objects that take care of exclusive access are almost always required to have some signaling technique that the threads will use between them to communicate about actions that occurred. Our example showed basic event usage, while it is left to the user to use and explore the demonstrated techniques further.

Using critical section

Another Windows synchronization object is *critical section*. Critical section behaves in the same manner as mutex. However, it can be used only in a single process context, as opposed to mutex, which can be shared among multiple processes. However, a critical section object is allocated faster by the system, with much smaller overhead.

Our following example will use the same task as before, only this time, we will use critical section instead of mutex. We can use critical section here because our entire application executes in a single process context.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now let's create our program and explain its structure.

1. Create a new empty C++ Windows application named `concurrent_operations4`.
2. Open **Solution Explorer** and right-click on **Header file**. Add an existing header file, used in *Chapter 1, Introduction to C++ Concepts and Features*, named `CQueue.h`.

3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `main.h`. Open `main.h`.
4. Add the following code:

```
#pragma once

#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <psapi.h>
#include <strsafe.h>
#include <string.h>
#include <sstream>
#include <fstream>
#include <commdlg.h>
#include "CQueue.h"

#pragma comment ( lib, "comctl32.lib" )
#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*'\\"" )

using namespace std;

#define CONTROL_BROWSE 100
#define CONTROL_START 101
#define CONTROL_RESULT 103
#define CONTROL_TEXT 104
#define CONTROL_PROGRESS 105

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
DWORD WINAPI StartAddress(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPSTR szFileName);

typedef struct
{
    int iA1;
    int iB1;
    int iC1;
    int iA2;
    int iB2;
    int iC2;
    HWND hWndProgress;
    HWND hWndResult;
} QueueElement, *PQueueElement;
```

5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.
6. Add the following code:

```
#pragma once

/*****
 * You need to specify text file with coordinates.      *
 * Our example will solve system of 2 linear equations  *
 * a1x + b1y = c1;                                     *
 * a2x + b2y = c2;                                     *
 *                                                     *
 * Each line should have exactly six coordinates.      *
 * File format example:                               *
 *                                                     *
 * 4 -3 2 -7 5 6                                       *
 * 12 1 -7 9 -5 8                                     *
 *                                                     *
 * Avoid new line at the end of the file.              *
 *****/
#include "main.h"

char szFilePath[MAX_PATH] = { 0 };
char szResult[4096];
CQueue<QueueElement> queue;
TCHAR* szEvent = _T("__event_374__");
CRITICAL_SECTION criticalSection;

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__concurrent_operations__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
```

```

wndEx.hIcon = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));
wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndEx.lpszMenuName = NULL;
wndEx.lpszClassName = szWindowClass;
wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wndEx))
{
    return 1;
}

InitCommonControls();

HWND hWnd = CreateWindow(szWindowClass,
    _T("Concurrent operations"), WS_OVERLAPPED
    | WS_CAPTION | WS_SYSMENU, 50, 50, 305, 250, NULL, NULL,
    wndEx.hInstance, NULL);

if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
    | FF_MODERN, _T("Microsoft Sans Serif"));

HWND hLabel = CreateWindow(_T("STATIC"),
    _T(" Press \"Browse\" and choose file with coordinates: "),
    WS_CHILD | WS_VISIBLE | SS_CENTERIMAGE | SS_LEFT | WS_BORDER,
    20, 20, 250, 25, hWnd, (HMENU)CONTROL_TEXT, wndEx.hInstance,
    NULL);

SendMessage(hLabel, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hResult = CreateWindow(_T("STATIC"), _T(""), WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 20, 65, 150, 75, hWnd,
    (HMENU)CONTROL_RESULT, wndEx.hInstance, NULL);

```



```

        SendMessage(hResult, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hBrowse = CreateWindow(_T("BUTTON"), _T("Browse"), WS_CHILD
| WS_VISIBLE
| BS_PUSHBUTTON, 190, 65, 80, 25, hWnd, (HMENU)CONTROL_BROWSE,
wndEx.hInstance, NULL);
        SendMessage(hBrowse, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hStart = CreateWindow(_T("BUTTON"), _T("Start"), WS_CHILD |
WS_VISIBLE
| BS_PUSHBUTTON, 190, 115, 80, 25, hWnd, (HMENU)CONTROL_START,
wndEx.hInstance, NULL);
        SendMessage(hStart, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hProgress = CreateWindow(PROGRESS_CLASS, _T(""), WS_CHILD |
WS_VISIBLE | WS_BORDER,
        20, 165, 250, 25, hWnd, (HMENU)CONTROL_PROGRESS, wndEx.
hInstance, NULL);
        SendMessage(hProgress, PBM_SETSTEP, (WPARAM)1, 0);
        SendMessage(hProgress, PBM_SETPOS, (WPARAM)0, 0);

        ShowWindow(hWnd, iWndShow);

        HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, szEvent);
        InitializeCriticalSection(&criticalSection);

        MSG msg;
        while (GetMessage(&msg, 0, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        DeleteCriticalSection(&criticalSection);
        CloseHandle(hEvent);
        UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

        return (int)msg.wParam;
    }

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)
{
    switch (uMsg)
    {

```

```
case WM_DESTROY:
{
    PostQuitMessage(0);
    break;
}
case WM_CLOSE:
{
    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, szEvent);
    SetEvent(hEvent);
    CloseHandle(hEvent);

    DestroyWindow(hWnd);
    break;
}
case WM_COMMAND:
{
    switch (LOWORD(wParam))
    {
        case CONTROL_BROWSE:
        {
            if (!FileDialog(hWnd, szFilePath))
            {
                MessageBox(hWnd,
                    _T("You must choose valid file path!"),
                    _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
            }
            else
            {
                char szBuffer[MAX_PATH];
                wsprintfA(szBuffer,
                    "\n File: %s Press \"Start\" now.",
                    strrchr(szFilePath, '\\') + 1);

                SetWindowTextA(GetDlgItem(hWnd, CONTROL_TEXT),
szBuffer);
            }
            break;
        }
        case CONTROL_START:
        {
            if (!*szFilePath)
            {
                MessageBox(hWnd,
                    _T("You must choose valid file path first!"),
                    _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
            }
        }
    }
}
```

```
        break;
    }

    ifstream infile(szFilePath);

    if (infile.is_open())
    {
        string line;
        while (std::getline(infile, line))
        {
            QueueElement* pQElement = new QueueElement();
            istringstream iss(line);

            if (!(iss >> pQElement->iA1 >> pQElement->iB1 >>
pQElement->iC1 >> pQElement->iA2 >> pQElement->iB2 >> pQElement->
iC2))
            {
                break;
            }

            pQElement->hWndProgress = GetDlgItem(hWnd,
                CONTROL_PROGRESS);
            pQElement->hWndResult = GetDlgItem(hWnd,
                CONTROL_RESULT);

            queue.Enqueue(pQElement);
        }

        infile.close();

        SendMessage(GetDlgItem(hWnd, CONTROL_PROGRESS),
            PBM_SETRANGE, 0, MAKELPARAM(0, queue.Count()));

        SYSTEM_INFO sysInfo;
        GetSystemInfo(&sysInfo);

        for (DWORD dwIndex = 0; dwIndex <
            sysInfo.dwNumberOfProcessors; dwIndex++)
        {
            HANDLE hThread = CreateThread(NULL, 0,
                (LPTHREAD_START_ROUTINE)StartAddress, &queue,
                0, NULL);
            SetThreadIdealProcessor(hThread, dwIndex);

            Sleep(100);
            CloseHandle(hThread);
        }
    }
```

```

        }
        else
        {
            MessageBox(hWnd, _T("Cannot open file!"),
_T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
        }
        break;
    }
}
break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    // We'll solve the linear system using Cramer's rule:
    CQueue<QueueElement>* pQueue = (CQueue<QueueElement>*)
lpParameter;

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, szEvent);
    QueueElement* pQElement = NULL;

    while (true)
    {
        DWORD dwStatus = WaitForSingleObject(hEvent, 10);

        if (dwStatus == WAIT_OBJECT_0)
        {
            break;
        }

        EnterCriticalSection(&criticalSection);

        pQElement = pQueue->Dequeue();

        LeaveCriticalSection(&criticalSection);

        if (pQElement == NULL)

```

```
{
    break;
}

char szBuffer[1024];

double dDeterminant = (pQElement->iA1 * pQElement->iB2) -
(pQElement->iB1 * pQElement->iA2);

if (dDeterminant != 0)
{
    double dX = ((pQElement->iC1 * pQElement->iB2) -
        (pQElement->iB1 * pQElement->iC2)) / dDeterminant;
    double dY = ((pQElement->iA1 * pQElement->iC2) -
        (pQElement->iC1 * pQElement->iA2)) / dDeterminant;

    sprintf_s(szBuffer, "  x = %8.4lf,\nty = %8.4lf\n", dX, dY);
}
else
{
    sprintf_s(szBuffer, "  Determinant is zero.\n");
}

strcat_s(szResult, 4096, szBuffer);
SetWindowTextA(pQElement->hWndResult, szResult);

SendMessage(pQElement->hWndProgress, PBM_STEPIT, 0, 0);
delete pQElement;

Sleep(1000);
}

CloseHandle(hEvent);

return 0L;
}

BOOL FileDialog(HWND hWnd, LPSTR szFileName)
{
    OPENFILENAMEA ofn;

    char szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));

    ofn.lStructSize = sizeof(ofn);
```

```

    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = "All\\0*.*\\0Text\\0*.TXT\\0";
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        strcpy_s(szFileName, MAX_PATH - 1, szFile);
        return TRUE;
    }

    return FALSE;
}

```

How it works...

The preceding code is pretty much the same, with differences in the synchronization object used. Using critical section demands a slightly different approach, because its handle cannot be inherited by a child process. What's important for us is that the critical section object allows only one thread to perform exclusive access, the same as mutex, only faster.

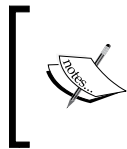
There's more...

Selection between mutex and critical section must be performed carefully. Even though the main difference resides in accessing synchronization objects between processes and system speed in providing an appropriate action and overhead for creation and manipulation, the user should always try to predict all possible scenarios (use cases) for the application before choosing the proper synchronization object and technique.

Using pipes

When we are explaining synchronization, it is absolutely necessary to give an example with processes. Unlike working in single address space, as with threads, working with multiple processes and their address spaces has its upsides and downsides.

The upsides are generally, in lower probability of errors. The context and address space of a process is private for the process and all its resources, including threads. If you allocate some memory location in one process and pass a pointer to another process, it will be meaningless! The operating system creates separate address spaces for all processes. The running process does not know anything about other processes, and as such, it is greedy: it wants all resources for itself. So, when working with multiple processes, you need a shared memory location if you want to share some data, and/or a so called *pipe* for communication. There are other communication objects such as *socket*, but we would need another book to explain everything here. We have chosen *pipe* because it is faster and easier to work with.



Windows Sockets enables programmers to create advanced Internet, intranet, and other network-capable applications to transmit application data across the wire, independent of the network protocol being used.

The downside would be a lot harder work because anything that you want to share among processes you have to write to shared memory from which other process can then read. Our following example will use the same task as in the previous three examples, only this time, there are no working threads; we will have working processes. This approach is good for *daemon* tasks such as servers and printers.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now let's create our program and explain its structure.

1. Create a new empty C++ Windows application named `concurrent_operations5`.
2. Open **Solution Explorer** and right-click on **Header file**. Add an existing header file, used in *Chapter 1, Introduction to C++ Concepts and Features*, named `CQueue.h`.
3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `main`. Open `main.h`.
4. Add the following code:

```
#pragma once

#include <windows.h>
#include <commctrl.h>
#include <tchar.h>
#include <psapi.h>
#include <strsafe.h>
```

```

#include <string.h>
#include <sstream>
#include <fstream>
#include <commdlg.h>
#include "CQueue.h"

#pragma comment ( linker, "\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' \
version='6.0.0.0' processorArchitecture='*' \
publicKeyToken='6595b64144ccf1df' language='*\\" )

using namespace std;

#define CONTROL_BROWSE 100
#define CONTROL_START 101
#define CONTROL_RESULT 103
#define CONTROL_TEXT 104
#define CONTROL_PROGRESS 105

#define BUFFER_SIZE 1024

#define PIPE_NAME _T( "\\\\.\\pipe\\__pipe_636__" )
#define EVENT_NAME _T( "__event_879__" )
#define MUTEX_NAME _T( "__mutex_132__" )
#define MAPPING_NAME _T( "__mapping_514__" )

typedef struct
{
    int iA1;
    int iB1;
    int iC1;
    int iA2;
    int iB2;
    int iC2;
    HWND hWndProgress;
    HWND hWndResult;
} QueueElement, *PQueueElement;

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam);
DWORD WINAPI ListenerRoutine(LPVOID lpParameter);
DWORD WINAPI StartAddress(LPVOID lpParameter);
BOOL FileDialog(HWND hWnd, LPSTR szFileName);
bool CalculateCramer(QueueElement* pQElement, char* szResult);
BOOL StartProcess(HWND hWndResult);

```


5. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.
6. Add the following code:

```
#pragma once

/*****
 * You need to specify text file with coordinates.      *
 * Our example will solve system of 2 linear equations  *
 * a1x + b1y = c1;                                     *
 * a2x + b2y = c2;                                     *
 *                                                     *
 * Each line should have exactly six coordinates.      *
 * File format example:                               *
 *                                                     *
 * 4 -3 2 -7 5 6                                       *
 * 12 1 -7 9 -5 8                                     *
 *                                                     *
 * Avoid new line at the end of the file.              *
 *****/
#include "main.h"

char szFilePath[MAX_PATH] = { 0 };
char szResult[4096];
CQueue<QueueElement> queue;

int WINAPI _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iWndShow)
{
    UNREFERENCED_PARAMETER(hPrev);
    UNREFERENCED_PARAMETER(szCommandLine);

    TCHAR* szWindowClass = _T("__concurrent_operations__");

    WNDCLASSEX wndEx = { 0 };
    wndEx.cbSize = sizeof(WNDCLASSEX);
    wndEx.style = CS_HREDRAW | CS_VREDRAW;
    wndEx.lpfnWndProc = WindowProcedure;
    wndEx.cbClsExtra = 0;
    wndEx.cbWndExtra = 0;
    wndEx.hInstance = hThis;
    wndEx.hIcon = LoadIcon(wndEx.hInstance,
        MAKEINTRESOURCE(IDI_APPLICATION));
    wndEx.hCursor = LoadCursor(NULL, IDC_ARROW);
```

```

wndEx.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wndEx.lpszMenuName = NULL;
wndEx.lpszClassName = szWindowClass;
wndEx.hIconSm = LoadIcon(wndEx.hInstance,
    MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wndEx))
{
    return 1;
}

InitCommonControls();

HWND hWnd = CreateWindow(szWindowClass,
    _T("Concurrent operations"), WS_OVERLAPPED
    | WS_CAPTION | WS_SYSMENU, 50, 50, 305, 250, NULL, NULL,
    wndEx.hInstance, NULL);

if (!hWnd)
{
    return NULL;
}

HFONT hFont = CreateFont(14, 0, 0, 0, FW_NORMAL, FALSE,
    FALSE, FALSE, BALTIC_CHARSET, OUT_DEFAULT_PRECIS,
    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH
    | FF_MODERN, _T("Microsoft Sans Serif"));

HWND hLabel = CreateWindow(_T("STATIC"),
    _T(" Press \"Browse\" and choose file with coordinates: "),
    WS_CHILD | WS_VISIBLE | SS_CENTERIMAGE | SS_LEFT | WS_BORDER,
    20, 20, 250, 25, hWnd, (HMENU)CONTROL_TEXT, wndEx.hInstance,
    NULL);

SendMessage(hLabel, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hResult = CreateWindow(_T("STATIC"), _T(""), WS_CHILD |
    WS_VISIBLE | SS_LEFT | WS_BORDER, 20, 65, 150, 75, hWnd,
    (HMENU)CONTROL_RESULT, wndEx.hInstance, NULL);
SendMessage(hResult, WM_SETFONT, (WPARAM)hFont, TRUE);

HWND hBrowse = CreateWindow(_T("BUTTON"), _T("Browse"), WS_CHILD
    | WS_VISIBLE

```

```

        | BS_PUSHTHUTTON, 190, 65, 80, 25, hWnd, (HMENU)CONTROL_BROWSE,
wndEx.hInstance, NULL);
        SendMessage(hBrowse, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hStart = CreateWindow(_T("BUTTON"), _T("Start"), WS_CHILD |
WS_VISIBLE
        | BS_PUSHTHUTTON, 190, 115, 80, 25, hWnd, (HMENU)CONTROL_START,
wndEx.hInstance, NULL);
        SendMessage(hStart, WM_SETFONT, (WPARAM)hFont, TRUE);

        HWND hProgress = CreateWindow(PROGRESS_CLASS, _T(""), WS_CHILD
        | WS_VISIBLE | WS_BORDER, 20, 165, 250, 25, hWnd,
        (HMENU)CONTROL_PROGRESS, wndEx.hInstance, NULL);
        SendMessage(hProgress, PBM_SETSTEP, (WPARAM)1, 0);
        SendMessage(hProgress, PBM_SETPOS, (WPARAM)0, 0);

        void* params[2] = { hProgress, hResult };
        HANDLE hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)
ListenerRoutine, params, 0, NULL);
        Sleep(100);
        CloseHandle(hThread);

        ShowWindow(hWnd, iWndShow);

        HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, EVENT_NAME);
        HANDLE hMutex = CreateMutex(NULL, FALSE, MUTEX_NAME);
        HANDLE hMapping = CreateFileMapping((HANDLE)-1, NULL,
        PAGE_READWRITE, 0, sizeof(QueueElement), MAPPING_NAME);

        MSG msg;
        while (GetMessage(&msg, 0, 0, 0))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        CloseHandle(hMapping);
        CloseHandle(hMutex);
        CloseHandle(hEvent);
        UnregisterClass(wndEx.lpszClassName, wndEx.hInstance);

        return (int)msg.wParam;
    }

LRESULT CALLBACK WindowProcedure(HWND hWnd, UINT uMsg, WPARAM
wParam, LPARAM lParam)

```

```
{
switch (uMsg)
{
case WM_DESTROY:
{
PostQuitMessage(0);
break;
}
case WM_CLOSE:
{
HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE,
EVENT_NAME);
SetEvent(hEvent);
CloseHandle(hEvent);

DestroyWindow(hWnd);
break;
}
case WM_COMMAND:
{
switch (LOWORD(wParam))
{
case CONTROL_BROWSE:
{
if (!FileDialog(hWnd, szFilePath))
{
MessageBox(hWnd,
_T("You must choose valid file path!"),
_T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
}
else
{
char szBuffer[MAX_PATH];
wsprintfA(szBuffer,
"\n File: %s Press \"Start\" now.",
strrchr(szFilePath, '\\') + 1);

SetWindowTextA(GetDlgItem(hWnd, CONTROL_TEXT),
szBuffer);
}
break;
}
case CONTROL_START:
{
```

```
    if (!*szFilePath)
    {
        MessageBox(hWnd,
            _T("You must choose valid file path first!"),
            _T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
        break;
    }

    ifstream infile(szFilePath);

    if (infile.is_open())
    {
        string line;
        while (std::getline(infile, line))
        {
            QueueElement* pQElement = new QueueElement();
            istringstream iss(line);

            if (!(iss >> pQElement->iA1 >> pQElement->iB1 >>
pQElement->iC1 >> pQElement->iA2 >> pQElement->iB2 >> pQElement-
>iC2))
            {
                break;
            }

            pQElement->hWndProgress = GetDlgItem(hWnd,
                CONTROL_PROGRESS);
            pQElement->hWndResult = GetDlgItem(hWnd,
                CONTROL_RESULT);

            queue.Enqueue(pQElement);
        }

        infile.close();
        SendMessage(GetDlgItem(hWnd, CONTROL_PROGRESS),
            PBM_SETRANGE, 0, MAKELPARAM(0, queue.Count()));

        int iCount = queue.Count();
        for (int iIndex = 0; iIndex < iCount; iIndex++)
        {
            StartProcess(GetDlgItem(hWnd, CONTROL_TEXT));
            Sleep(100);
        }
    }
}
```

```

        else
        {
            MessageBox(hWnd, _T("Cannot open file!"),
_T("Error!"), MB_OK | MB_TOPMOST | MB_ICONERROR);
        }
        break;
    }
    }
    break;
}
default:
{
    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
}
return 0;
}

DWORD WINAPI ListenerRoutine(LPVOID lpParameter)
{
    void** lpParameters = (void**)lpParameter;
    HWND hWndProgress = (HWND)lpParameters[0];
    HWND hWndResult = (HWND)lpParameters[1];

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, EVENT_NAME);

    while (TRUE)
    {
        DWORD dwStatus = WaitForSingleObject(hEvent, INFINITE);
        if (dwStatus == WAIT_OBJECT_0)
        {
            break;
        }

        HANDLE hPipe = CreateNamedPipe(PIPE_NAME, PIPE_ACCESS_DUPLEX,
PIPE_WAIT | PIPE_READMODE_MESSAGE
    | PIPE_TYPE_MESSAGE, PIPE_UNLIMITED_INSTANCES, BUFFER_SIZE,
BUFFER_SIZE, 0, NULL);
        if (hPipe == INVALID_HANDLE_VALUE)
        {
            char szBuffer[MAX_PATH];
            wsprintfA(szBuffer, " Error: [%u]\n", GetLastError());

            SetWindowTextA(hWndResult, szBuffer);

```

```
        return 2L;
    }

    if (ConnectNamedPipe(hPipe, NULL))
    {
        void* params[2] = { hPipe, hWndResult };
        HANDLE hThread = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)StartAddress, params, 0, NULL);
        Sleep(100);
        CloseHandle(hThread);
    }

    SendMessage(hWndProgress, PBM_STEPIT, 0, 0);
}

CloseHandle(hEvent);
return 0L;
}

DWORD WINAPI StartAddress(LPVOID lpParameter)
{
    void** lpParameters = (void**)lpParameter;
    HANDLE hPipe = (HANDLE)lpParameters[0];
    HWND hWndResult = (HWND)lpParameters[1];

    char* szRequest = (char*)HeapAlloc(GetProcessHeap(), 0,
        BUFFER_SIZE * sizeof(char));
    char* szReply = (char*)HeapAlloc(GetProcessHeap(), 0,
        BUFFER_SIZE * sizeof(char));

    DWORD dwBytesRead = 0;
    DWORD dwReplyBytes = 0;
    DWORD dwBytesWritten = 0;

    memset(szRequest, 0, BUFFER_SIZE * sizeof(char));
    memset(szReply, 0, BUFFER_SIZE * sizeof(char));

    if (!ReadFile(hPipe, szRequest, BUFFER_SIZE * sizeof(char),
        &dwBytesRead, NULL))
    {
        return 2L;
    }

    char szBuffer[1024] = { 0 };
```

```

wsprintfA(szBuffer, "  PID: [%s] connected.\n", szRequest);
strcat_s(szResult, szBuffer);
SetWindowTextA(hWndResult, szResult);

HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE, MUTEX_NAME);

WaitForSingleObject(hMutex, INFINITE);

QueueElement* pQElement = (QueueElement*)queue.Dequeue();

HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE,
MAPPING_NAME);
QueueElement* pMapping = (QueueElement*)MapViewOfFile(hMapping,
FILE_MAP_ALL_ACCESS, 0, 0, 0);

memcpy(pMapping, pQElement, sizeof(QueueElement));

UnmapViewOfFile(pMapping);
CloseHandle(hMapping);

ReleaseMutex(hMutex);
CloseHandle(hMutex);

sprintf_s(szReply, BUFFER_SIZE * sizeof(char), "OK%s",
szRequest);
dwReplyBytes = (DWORD)((strlen(szReply) + 1) * sizeof(char));

if (!WriteFile(hPipe, szReply, dwReplyBytes, &dwBytesWritten,
NULL))
{
    return 2L;
}

memset(szRequest, 0, BUFFER_SIZE * sizeof(char));

if (!ReadFile(hPipe, szRequest, BUFFER_SIZE * sizeof(char),
&dwBytesRead, NULL))
{
    return 2L;
}

wsprintfA(szBuffer, "%s", szRequest);
strcat_s(szResult, szBuffer);

```



```
SetWindowTextA(hWndResult, szResult);

delete pQElement;

FlushFileBuffers(hPipe);
DisconnectNamedPipe(hPipe);
CloseHandle(hPipe);
HeapFree(GetProcessHeap(), 0, szRequest);
HeapFree(GetProcessHeap(), 0, szReply);

return 0L;
}

BOOL FileDialog(HWND hWnd, LPSTR szFileName)
{
    OPENFILENAMEA ofn;

    char szFile[MAX_PATH];

    ZeroMemory(&ofn, sizeof(ofn));

    ofn.lStructSize = sizeof(ofn);
    ofn.hwndOwner = hWnd;
    ofn.lpstrFile = szFile;
    ofn.lpstrFile[0] = '\\0';
    ofn.nMaxFile = sizeof(szFile);
    ofn.lpstrFilter = "All\\0*.*\\0Text\\0*.TXT\\0";
    ofn.nFilterIndex = 1;
    ofn.lpstrFileTitle = NULL;
    ofn.nMaxFileTitle = 0;
    ofn.lpstrInitialDir = NULL;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    if (GetOpenFileNameA(&ofn) == TRUE)
    {
        strcpy_s(szFileName, MAX_PATH - 1, szFile);
        return TRUE;
    }

    return FALSE;
}

BOOL StartProcess(HWND hWndResult)
```

```

{
    STARTUPINFO startupInfo = { 0 };
    PROCESS_INFORMATION processInformation = { 0 };

    BOOL bSuccess = CreateProcess(
        _T("../x64\\Debug\\ClientProcess.exe"), NULL, NULL, NULL,
        FALSE, 0, NULL, NULL, &startupInfo, &processInformation);

    if (!bSuccess)
    {
        char szBuffer[MAX_PATH];
        wsprintfA(szBuffer, " Process creation fails: [%u]\\n",
            GetLastError());

        SetWindowTextA(hWndResult, szBuffer);
    }

    return bSuccess;
}

```

7. Open **Solution Explorer** and right-click on **Solution concurrent_operations5**. Add a new empty Console application named `ClientProcess`. Now right-click on the `ClientProcess` project. Add a new source file named `main`. Open `main.cpp`.
8. Add the following code:

```

#pragma once

#include "../concurrent_operations5/main.h"

int _tmain(void)
{
    char szBuffer[BUFFER_SIZE];
    DWORD dwRead = 0;
    DWORD dwWritten = 0;
    DWORD dwMode = PIPE_READMODE_MESSAGE;

    char szMessage[1024] = { 0 };
    wsprintfA(szMessage, "%u", GetCurrentProcessId());

    DWORD dwToWrite = (DWORD)(strlen(szMessage) * sizeof(char));

    HANDLE hPipe = NULL;

    while (true)

```

```
{
    hPipe = CreateFile(PIPE_NAME, GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, 0, NULL);
    if (hPipe != INVALID_HANDLE_VALUE)
    {
        WaitNamedPipe(PIPE_NAME, INFINITE);
        SetNamedPipeHandleState(hPipe, &dwMode, NULL, NULL);
        break;
    }

    Sleep(100);
}

if (!WriteFile(hPipe, szMessage, dwToWrite, &dwWritten, NULL))
{
    printf("Error: [%u]\n", GetLastError());
    return system("pause");
}

printf("Request sent.\nReceiving task:\n");
char szResult[1024] = { 0 };

while (ReadFile(hPipe, szBuffer, BUFFER_SIZE * sizeof(char),
&dwRead, NULL) && GetLastError() != ERROR_MORE_DATA)
{
    if (szBuffer[0] == 'O' && szBuffer[1] == 'K' && ((DWORD)
strtol(szBuffer + 2, NULL, 10) == GetCurrentProcessId()))
    {
        printf("Client process: [%s]\n", szBuffer + 2);

        HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS,
FALSE, MUTEX_NAME);
        WaitForSingleObject(hMutex, INFINITE);

        HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS,
FALSE, MAPPING_NAME);
        QueueElement* pMapping = (QueueElement*)
MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);

        if (pMapping != 0)
        {
            if (CalculateCramer(pMapping, szResult))
            {

```

```

        dwToWrite = (DWORD)(strlen(szResult) * sizeof(char));

        if (!WriteFile(hPipe, szResult, dwToWrite, &dwWritten,
NULL))
        {
            printf("Error: [%u]\n", GetLastError());
            break;
        }
        else
        {
            printf("Result: %s\n", szResult);
        }
    }

    UnmapViewOfFile(pMapping);
    CloseHandle(hMapping);

    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}
else
{
    printf("Error in connection [%u]\n", GetLastError());
}
}

printf("\nSuccess!\nPress ENTER to exit.");
scanf_s("%c", szBuffer);

CloseHandle(hPipe);
return 0;
}

bool CalculateCramer(QueueElement* pQElement, char* szResult)
{
    // We'll solve the linear system using Cramer's rule:
    double dDeterminant = (pQElement->iA1 * pQElement->iB2) -
(pQElement->iB1 * pQElement->iA2);
    if (dDeterminant != 0)
    {
        double dX = ((pQElement->iC1 * pQElement->iB2) -
(pQElement->iB1 * pQElement->iC2)) / dDeterminant;
    }
}

```

```
double dY = ((pQElement->iA1 * pQElement->iC2) -
             (pQElement->iC1 * pQElement->iA2)) / dDeterminant;
sprintf_s(szResult, strlen(szResult) - 1,
          " x = %8.4lf, \ty = %8.4lf\n", dX, dY);
}
else
{
    sprintf_s(szResult, strlen(szResult) - 1,
              " Determinant is zero.\n");
}

return true;
}
```

How it works...

Well, this example has a lot more code. However, the good part is that, because there is no security among threads (without proper synchronization, of course), and one thread can mess with the other thread's data, or global variables, working with processes is safer by definition. What we mean by this is that processes are divided by their own address space, global variables, and synchronization objects. On the other hand, if one thread blocks all the threads that wait for its output or shared resources will also be blocked. When you learn about synchronization, it is important to understand the process model and **Interprocess Communication (IPC)** as well. Let's explain `main.h` first. As more projects will use a pipe name as well as a mutex and file mapping name, we will define them as macros inside the main header file. All the rest remains the same.

Now, let's focus on `concurrent_operations5`. The application entry point remains pretty much the same. We'll create an event, mutex, and file mapping before the main thread enters the message loop. We'll dispose all the mentioned objects after the application is closed. The other thing about file mapping is that it is important that we set the size of the shared object correctly; so we will set the size to `sizeof(QueueElement)`.

We'll also create a **listener** thread that will use `ListenerRoutine` as a start address. Its task is to loop forever, or until the event state becomes `signaled`. It will create a named pipe and wait for client connections, by calling the `ConnectNamedPipe` API, as explained in *Chapter 4, Message Passing*.

When the client connects, it will start another thread responsible for request processing and will wait for another connection. The working thread starts from the `StartAddress` routine. Its task is to allocate enough space for requests and replies, and then it will call the `ReadFile` API in order to wait until the client writes a request and sends it back. Again a very important task is to recognize what we need to protect. If multiple processes will read and write to shared file mapping, then that is the place we must protect!

It will take the ownership of the mutex, get the element from the queue, write to shared file mapping, and then it will release the mutex. When a reply is prepared, it will write to the pipe using the `WriteFile` API, and again wait for the client's reply using `ReadFile`. After receiving the calculation result from an external process, it will release the used resources and exit.

Inside `WindowProcedure`, we've changed so that instead of creating threads we create processes—as many as there are elements in the queue. It would be wiser if we had set only one working process to calculate and receive tasks one by one, but we wanted to point out important parts of Interprocess Communication as well as synchronization—so an example designed like this makes more sense.

Now, let's review the client process: first, it will try to connect to the pipe. As our main application creates processes fast, with a 0.1 second gap between them, there is a great possibility that these processes will try to connect to the pipe simultaneously. Well that is impossible, and you'll probably get an `ERROR_INVALID_HANDLE` error! That's why the process will loop, with a 0.1 second delay until it's connected. After successful connection, the process will introduce itself using a unique process ID. It will then use the `ReadFile` API to wait until a request arrives. If the request format is `OK12345`, where `12345` stands for its process identifier, it will ask for the ownership of the mutex, and when the process gets it, it will read coordinates from the shared file mapping and respond using `WriteFile` once again. After it responds successfully, it will release the mutex and write its result to the console output.

There's more...

All these techniques that have been shown in this chapter are ground zero for the user's future development. The user can apply the knowledge later. We tried to point out the most important steps in design as well as in development, but the user should take this as a starting point for development. Further learning and studying operating systems is necessary if development of complex systems (tasks) is required. Again, a smart design of the application itself is extremely important! Remember to take application design very carefully, especially when parallel execution will take place.

6

Threads in the .NET Framework

In this chapter, we will cover the following topics:

- ▶ Managed versus unmanaged code
- ▶ How threading works in .NET
- ▶ The difference between foreground and background threads
- ▶ Understanding the .NET synchronization essentials
- ▶ Locking and avoiding deadlocks
- ▶ Thread safety and types of the .NET framework
- ▶ Signaling with event wait handles
- ▶ Event-based Asynchronous Pattern
- ▶ The BackgroundWorker class
- ▶ Interrupting, aborting, and safe canceling the thread execution
- ▶ Non-blocking synchronization
- ▶ Signaling with Wait and Pulse
- ▶ The Barrier Class

Introduction

As many of you already know, there are many frameworks today that programmers can use for their development, which give them many features, rich tools, visually stunning user experiences, and much easier programming.

Our goal is to have a closer look at one of the most powerful frameworks today—the Microsoft .NET framework. The Microsoft .NET framework provides excellent programming tools such as UI forms, templates, various predefined controls, code maps, and SQL Server development tools, with multiple programming languages, such as C#, F#, and Visual Basic. We will focus on the **Common Language Runtime (CLR)** C++ language, also known as C++/CLI, where CLI stands for *Common Language Infrastructure*, which is a standard part of the framework.

Since the .NET framework is very large and complex—we will focus on explaining the multithreaded approach and synchronization of objects and techniques.

Managed versus unmanaged code

You can think of a framework as a set of routines and objects that helps you create an application more easily and with rich content, due to a variety of predefined controls and UI objects. When you write the code, at the end, you need to compile it. After compilation, you often get a compiled application (*.exe), a dynamic library (*.dll), a kernel driver (*.sys), and so on. This compiled file (or more files) contains the machine language (code) that consists of only zeroes and ones, and it is not readable by humans. This, by definition, is called unmanaged code. So, native C++ code (the one that we were coding so far) is compiled directly to the machine language and we call it unmanaged. Unlike managed code, it is fast and is executed directly on **Hardware Abstraction Layer (HAL)**, which provides direct interaction with the machine hardware and low-level execution.



The Windows NT HAL refers to a layer of software that deals directly with your computer hardware. As the HAL operates at a level between the hardware and the Windows NT executive services, applications and device drivers need not be aware of any hardware-specific information. It hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms. Applications and device drivers are no longer allowed to deal with the hardware directly and must make calls to the HAL routines to determine hardware-specific information (from MSDN).

On the other hand, managed code is a bit different. It is executed over common language runtime, or in another words, a certain layer that resides between the machine hardware and your application. The code you write isn't compiled directly into the machine language. It is compiled into an intermediate language, MSIL or Microsoft Intermediate Language. This ability gives your managed code advantages and disadvantages.

Let's review the advantages. Managed code, after it is compiled into MSIL, is checked for various errors, such as static type checking (which is not performed on native code compilation) and runtime checking. It offers a garbage collector, or the ability to free unused dynamic memory allocated by the application and so on. We already said that the framework provides rich programming tools, so when we summarize everything, there are lots of advantages to using the .NET framework.

However, there are disadvantages too. The application developed using any framework, such as .NET or **Java Runtime Environment (JRE)**, is much slower than an application developed using native code. This happens because managed code is executed over a CLR layer, which is then executed on the machine hardware. When you use a framework, it is easier for you to develop the application, but you will never have complete control over the application's execution, along with its interaction with the machine itself. The best you can do is what the framework provides you with.

To conclude, you should use a native approach whenever you need fast execution and/or complete control over the application execution, while the framework should be used whenever you need an application that targets rich UI and much simpler development. Over the years, Microsoft has done an excellent job with the .NET framework, but it will never be able to provide everything you can do with the native approach. That's why we are focusing on managed code only in one chapter, and the rest of the chapters are covered with a native approach.

Note that everything that you can do with frameworks you can achieve with a native approach. You would need more time and much more effort—but it is doable. You can't do everything with a managed approach, not even a part of what you can do with native code. To explain this even better, let's examine two scenarios. First, we need to create software that will control nuclear power plant machines. The second task is to create software that will use a club to rent DVDs. The first task should be developed using native C++ (or even the assembly language) to provide extremely fast and precise execution. This is because, and you would agree with me, nuclear energy is too sensitive a thing to be taken leniently. On the other hand, DVD software should look good (so that you can sell it better), and the execution time doesn't need to be in micro or nanoseconds. It is enough to be user friendly and work correctly, so we could use .NET to develop it faster and with lesser effort.

How threading works in .NET

An application that is executed on a CPU with more than one core should benefit from concurrent execution, with as many concurrent tasks as there are logical cores to execute on. Of course, there are many questions that need to be taken care of when we want to execute parallel tasks:

- ▶ How to divide a set of tasks so that each of them is executed in a multicore environment?
- ▶ How to ensure that the number of concurrent tasks doesn't exceed the number of CPU cores?
- ▶ If some task is stopped, such as when you wait for I/O, how to discover this and make the CPU execute something else while waiting for I/O completion?
- ▶ How to find out when one or more parallel operations are completed?

When you use a framework, answers to these questions depend on the framework itself. Microsoft provides a `Thread` class as well as a `Task` class and a collection of auxiliary types that resides in the `System::Threading::Task` namespace.

The `Task` class represents abstraction of concurrent operations. CLR implements tasks and schedules them for execution using the `Thread` object and the `ThreadPool` class. Multithreaded design can be performed using the `Thread` class, which resides in the `System::Threading` namespace. However, for Windows 8 and the Windows store application, the `Thread` class isn't available. You should use the `Task` class because it provides very powerful abstraction of multitasking, and if you design an application that creates a specific (but finite) number of threads, your application won't be scalable enough. In other words, if the number of threads exceeds the number of cores, you could get slow and unresponsive behavior. CLR optimizes the number of requested threads so that you could implement concurrent tasks in the best way with respect to the number of available cores. When your application creates a `Task` object, an actual task is added to the working queue. When a thread becomes available, the task is removed from the queue and the available thread executes it. The `ThreadPool` class implements several optimization techniques and uses an algorithm for the *work-stealing* strategy for better thread scheduling.



Work-stealing is a feature of the scheduling algorithm, where problems are analyzed and then the execution (for solving the problem) is performed. Considering the number of quantum and the calling depth, if the problem is easy to solve, then it is directly solved. However, if the problem is considered complex, it is split into smaller parts using already-created, available threads from other CPU cores, for task solving. Later, the subresults are merged into the final result. The idea is to use (steal) available threads, of course, if possible.

Now, we will create our very first example using CLR. Visual Studio, before 2012 and 2013, had the C++ Windows Forms template included. Newer versions that were released in 2012 and 2013 don't ship with the forms template, but you can achieve everything with a little more effort. Our first example will create a simple application, which will calculate the *factorial* (the product of all positive integers less than or equal to n) with n number of iterations, where the working thread will calculate while the main thread will update the progress bar—only to demonstrate the basic CLR program and explain a very important feature: cross-thread operations.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code to it:

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code to it:

```
#pragma once
namespace CLRApplication
{
    using namespace System;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading::Tasks;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
```

```
        InitializeComponent();
        this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
    }
protected:
    ~MyForm()
    {
        if (components)
        {
            delete components;
        }
    }
private:
    System::ComponentModel::Container ^components;
    Label^ label;
    ProgressBar^ progress;
    static const int N = 25;

    System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
    {
        InitializeApp();
    }

    System::Void InitializeApp(void)
    {
        label = gcnew Label();
        label->Location = Point(2, 25);

        progress = gcnew ProgressBar();
        progress->Location = Point(2, 235);
        progress->Width = 280;
        progress->Height = 25;
        progress->Step = 1;
        progress->Maximum = MyForm::N;
        progress->Value = 0;

        this->Controls->Add(label);
        this->Controls->Add(progress);

        Task^ task = gcnew Task(gcnew Action<Object^>
            (&MyForm::StartAddress), this);
        task->Start();
    }

    static System::Void StartAddress(Object^ parameter)
```

```

{
    MyForm^ form = (MyForm^)parameter;

    UpdateProgressBarDelegate^ action =
        gcnew UpdateProgressBarDelegate(form,
            &MyForm::UpdateProgressBar);
    __int64 iResult = 1;

    for (int iIndex = 1;
        iIndex <= MyForm::N; iIndex++)
    {
        iResult *= iIndex;
        form->BeginInvoke(action);

        System::Threading::Thread::Sleep(100);
    }

    UpdateLabelTextDelegate^ textAction =
        gcnew UpdateLabelTextDelegate(form,
            &MyForm::UpdateLabelText);
    form->BeginInvoke(textAction, L"Result: "
        + Convert::ToString(iResult));
}

delegate void UpdateProgressBarDelegate(void);

System::Void UpdateProgressBar(void)
{
    progress->PerformStep();
}

delegate void UpdateLabelTextDelegate(String^ szText);

System::Void UpdateLabelText(String^ szText)
{
    label->AutoSize = false;
    label->Text = szText;
    label->AutoSize = true;
}

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer
/// support - do not modify
/// the contents of this method with the code editor.
/// </summary>

```

```
void InitializeComponent(void)
{
    this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
    this->Size = System::Drawing::Size(300, 300);
    this->Text = L"MyForm";
    this->Padding = System::Windows::Forms::Padding(0);
    this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
}
#pragma endregion
};
}
```

How it works...

Let's explain `MyForm.h` first. Every .NET application usually starts by defining the application (one or more) namespaces; in our case, we are using the `CLRApplication` default namespace (produced by the designer). As we want to use the framework's features, we need to specify libraries (also called assemblies) that we want to use, as shown in the following code:

```
using namespace System;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Drawing;
using namespace System::Threading::Tasks;
```

These assemblies allow us to use UI tools such as forms, points, and location objects. Then, we will derive our UI class, `MyForm`, which represents the template window derived from the `System::Windows::Forms::Form` base class. Pay attention to the `MyForm` constructor, where the first command is a call to `InitializeComponent`. After that goes our code. `InitializeComponent` loads the compiled page of a component and initializes base values, such as size, text, and padding. Implementation of `InitializeComponent` resides in a region called `Windows Form Designer generated code`. The designer also generates a destructor under the protected scope of our class, and a single `components` variable under the `private` scope, which encapsulates zero or more components (controls). The code follows:

```
System::ComponentModel::Container ^components;
```

We've added `label`, `progress`, and `N` variables for the progress bar control, label control, and the iterations limiter `N` respectively. The `InitializeApp` method creates a new label and sets its location (the `x` and `y` coordinates on the parent control). It also creates a new progress bar and initializes its location, size (the width and height), step (how many steps the progress bar will update after calling its `PerformStep` method), maximum (the maximum value that progress can have), and a starting value. After that, we will add these two controls to the main form by calling the `Add` method:

```
this->Controls->Add( label );
this->Controls->Add( progress );
```

Now, it is time to create our thread (task). As in the native approach, we must specify its start address, and, if needed, pass an argument to its starting routine. As our start routine is marked `static`, we will pass the `this` pointer to the routine in order to use our class instance later. The task constructor expects a pointer to the `Action` object, and if the `Action<Object^>` overload is used, it also expects a parameter constructor. We will create a pointer to `Action<Object^>` as follows:

```
gcnew Action<Object^>( &MyForm::StartAddress )
```

The `Action` constructor expects the delegate (function pointer) to pass on, and we are providing the address of the `StartAddress` method.

Now, let's explain the `StartAddress` method. First, we need to cast `Object^` (similar to `void*`) to `MyForm^`; next, we will instantiate a delegate that will later be used for invoking the main thread to update the progress bar. This part is very important to understand! Unlike in the native approach, CLR pays attention to which thread creates what control, and only the thread that has created a certain control can manipulate it. As the main thread has created the progress bar in the `InitializeApp` method, only the main thread can manipulate it. If you try to update the progress bar from another thread, you'll get an exception: **Cross-thread operation not valid: Control 'progress' accessed from a thread other than the thread it was created on.** In other words, the working thread has to ask the main thread to update it, because the main thread has created the progress bar. This can be achieved by using the `Invoke` or `BeginInvoke` methods. These two methods are doing the same thing; they execute a delegate on the thread that the control's underlying handle was created on. The only difference between them is that `BeginInvoke` does that asynchronously, while `Invoke` does that by waiting for the operation to return. Then, we will create a loop that will iterate *n* times, and at the end of each iteration we will call `BeginInvoke` to update the progress bar. After the iterations are completed, we will perform the same task for setting the label text.

As we said earlier, we will instantiate a delegate for both the previously described operations. In order to achieve that, we must declare delegates and methods to be invoked. The declarations are shown in the following code:

```
delegate void UpdateProgressBarDelegate( void );
System::Void UpdateProgressBar( void )
{
    progress->PerformStep( );
}
```


We will follow similar steps to set the label text. After we've completed creating our class, we need to specify the application entry point. This is done in the `main.cpp` file. Just like in our unmanaged code examples, we will include certain header files to be able to use predefined types and the `_tWinMain` method. We need to include the `MyForm.h` form to be able to use our class and assemblies that are required for the framework to work. We need the `System::Windows::Forms` assembly in order to use the `Application` class. We also need `System::Threading` to be able to set the thread apartment state and the `CLRApplication` namespace in order to use `MyForm` as an instance to the `Application::Run` method. We will set the `ApartmentState` variable to **Single-Threaded Apartment (STA)**. This is required for the proper lazy initialization of COM to be UI-friendly, for example when we want to use drag-and-drop features on windows UI elements, and for hosting COM controls. The `Application::EnableVisualStyles` method is used for rich UI styles, similar to `#pragma comment (linker, ...)` used in our unmanaged example before. At the end, we will call `Application::Run` that will launch the application window and enter its message loop.

There's more...

Even though we've explained a large part of our .NET application, there still remains a lot of explanation, which is not a part of this book. The framework is so big that we would need a few books to take care of everything. We tried to explain the most important parts that affect our application's execution and are essential to the multithreading concept that we are focusing on. One more thing that is important to understand is the `gcnew` operator. Even though we are using managed C++, we could still use unmanaged code. Well, because of this excellent feature, the `new` operator is used for unmanaged memory allocation, which has a fixed address and must be deallocated by the programmer. Unlike unmanaged allocation, the `gcnew` operator allocates memory from the CLR heap, and as such, is managed by the framework. This means that the addresses are not fixed; it could change during execution. It also means that the programmer doesn't need to take care of its reallocation. CLR will delete it using **Garbage Collector (GC)** when it is not needed anymore. This also stands for the `^` and `*` operators, which denote pointers, `^` for managed and `*` for unmanaged, and for the `%` and `&` operators, which denote references for managed and unmanaged respectively. Comparison follows.

Operator	Unmanaged	Managed
Indirection (pointer)	*	^
Reference	&	%

The difference between foreground and background threads

In .NET, there are two types of threads: foreground and background. Threads are, by default, created as foreground threads. You could set the thread state explicitly as background. The difference is that foreground threads keep an application alive as long as they are running, while background threads do not. In other words, when you close applications, all background threads are automatically terminated. So, when all the foreground threads complete execution, the application can exit before the background threads return. After all the foreground threads have been stopped, or after the application exits, the system stops all background threads.

Our previous example used the `Task` object, which runs in the background by default. Our following example will be the same as the previous one, with the focus on foreground execution. That's why we will use the `Thread` object instead of `Task`.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication2`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication2;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
```

```
Application::EnableVisualStyles();
Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MyForm());

return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named **MyForm**. Open **MyForm.h**.
4. Add the following code:

```
#pragma once
namespace CLRApplication2
{
    using namespace System;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();
            this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
        }
        protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }
    private:
        System::ComponentModel::Container ^components;
        Label^ label;
        ProgressBar^ progress;
    };
}
```

```

static const int N = 25;

System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
{
    InitializeApp();
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    Thread^ thread = gcnew Thread(gcnew
ParameterizedThreadStart( &MyForm::StartAddress));
    thread->Start(this);
}

static System::Void StartAddress(Object^ parameter)
{
    MyForm^ form = (MyForm^)parameter;

    UpdateProgressBarDelegate^ action = gcnew
UpdateProgressBarDelegate(form, &MyForm::UpdateProgressBar);

    __int64 iResult = 1;
    for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
    {
        iResult *= iIndex;
        form->BeginInvoke(action);

        System::Threading::Thread::Sleep(100);
    }
}

```

```
    }

    UpdateLabelTextDelegate^ textAction = gcnew
UpdateLabelTextDelegate(form, &MyForm::UpdateLabelText);
    form->BeginInvoke(textAction, L"Result: " +
Convert::ToString(iResult));
    }

    delegate void UpdateProgressBarDelegate(void);

    System::Void UpdateProgressBar(void)
    {
        progress->PerformStep();
    }

    delegate void UpdateLabelTextDelegate( String^ szText);

    System::Void UpdateLabelText(String^ szText)
    {
        label->AutoSize = false;
        label->Text = szText;
        label->AutoSize = true;
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    void InitializeComponent(void)
    {
        this->components =
            gcnew System::ComponentModel::Container();
        this->Size = System::Drawing::Size(300, 300);
        this->Text = L"MyForm";
        this->Padding = System::Windows::Forms::Padding(0);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
    }
#pragma endregion
};
}
```

How it works...

While threads are created as a foreground, tasks are not. They are, by default, created as background threads. In your development, you should carefully choose which type of thread you need and select the appropriate one. For example, if we want to calculate a large factorial (for example, for 10,000), it would take a long time, and we would use the `Task` object in order to be able to interrupt execution by simply closing the application. On the other hand, if our program must complete its task without any possible interruption from the user (even though there are other ways to prevent this), we would use the `Thread` object and its ability to perform tasks as a foreground thread.

There's more...

We'll also point out that you could use the `Thread` object for both types of execution, foreground and background, while the `Task` object cannot be used for this. In order to use `Thread` as a background thread, you would only need to change its `IsBackground` property to `true` before starting it. The code to do so is as follows:

```
Thread^ thread = gcnew Thread(
    gcnew ParameterizedThreadStart(&MyForm::StartAddress));
thread->IsBackground = true;
thread->Start(this);
```

Understanding the .NET synchronization essentials

After basic thread operations such as starting the thread or task and providing its start address, .NET also provides synchronization mechanisms. These mechanisms are important in order to understand the proper use of threads for concurrent operations without interfering with threads. Furthermore, we will explain blocking methods, locking, signaling, and non-blocking synchronization.

In .NET, the blocking methods are `Thread::Sleep`, `Thread::Join`, and `Task::Wait`. It is better to block the thread instead of spinning it and wasting processor time. When we say spin, we think of a loop where a thread will iterate the loop until some condition is satisfied. The `Sleep` method simply changes the thread state to `suspended` for a certain period of time. The `Join` method blocks the calling thread until the running thread terminates. The `Wait` method waits for `Task` to complete its execution within a specified time interval.

Our following example will calculate the factorial of 25 and will then ask the user whether it should continue or not. In this way, we would need to use the `Sleep` method as the most primitive way of spinning with a 1-second sleep interval, in order to not waste too much processor time. In order to communicate between the main thread and the working thread, we will use the `PostMessage` API, as the most primitive way, but enough for now, because we will explain other communication techniques later.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication3`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"
```

```
using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication3;
```

```
[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.

4. Add the following code:

```

#pragma once

#include <Windows.h>
#pragma comment ( lib, "User32.lib" )
#define END_TASK WM_USER + 1

namespace CLRApplication3
{
    using namespace System;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading::Tasks;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void) : bContinue(true)
        {
            InitializeComponent();
            this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
        }
    protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }

        virtual void WndProc(Message% msg) override
        {
            switch (msg.Msg)
            {
                case END_TASK:
                {
                    System::Windows::Forms::DialogResult^ dlgResult =
                    MessageBox::Show( L"Do you want to calculate again?", L"Question",
                    MessageBoxButtons::YesNo, MessageBoxIcon::Question);

                    if (*dlgResult ==
                        System::Windows::Forms::DialogResult::Yes)

```



```

        {
            bContinue = true;
            progress->Value = 0;
            label->Text = L"Calculating...";
        }
        else
        {
            PostMessage( (HWND)this->Handle.ToPointer(), WM_CLOSE,
0, 0);
        }

        break;
    }
}

Form::WndProc(msg);
}
private:
    System::ComponentModel::Container ^components;
    Label^ label;
    ProgressBar^ progress;
    static const int N = 25;
    bool bContinue;

    System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
    {
        InitializeApp();
    }

    System::Void InitializeApp(void)
    {
        label = gcnew Label();
        label->Location = Point(2, 25);

        progress = gcnew ProgressBar();
        progress->Location = Point(2, 235);
        progress->Width = 280;
        progress->Height = 25;
        progress->Step = 1;
        progress->Maximum = MyForm::N;
    }
}

```

```
progress->Value = 0;

this->Controls->Add(label);
this->Controls->Add(progress);

Task^ task = gcnew Task(gcnew Action<Object^>
    (&MyForm::StartAddress), this);
task->Start();
}

static System::Void StartAddress(Object^ parameter)
{
    MyForm^ form = (MyForm^)parameter;

    GetFormHandleDelegate^ getHandle = gcnew
    GetFormHandleDelegate(form, &MyForm::GetFormHandle);
    IntPtr handle = (IntPtr)form->Invoke(getHandle);

    UpdateProgressBarDelegate^ action = gcnew
    UpdateProgressBarDelegate(form, &MyForm::UpdateProgressBar);
    UpdateLabelTextDelegate^ textAction = gcnew
    UpdateLabelTextDelegate(form, &MyForm::UpdateLabelText);

    Int64 iResult = 1;

    while (true)
    {
        if (!form->bContinue)
        {
            System::Threading::Thread::Sleep(1000);
            continue;
        }

        iResult = 1;
        for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
        {
            iResult *= iIndex;
            form->BeginInvoke(action);

            System::Threading::Thread::Sleep(100);
        }
    }
}
```

```

    }

    form->BeginInvoke(textAction, L"Result: " +
Convert::ToString(iResult));

    PostMessage((HWND)handle.ToPointer(), END_TASK, 0, 0);

    form->bContinue = false;

    System::Threading::Thread::Sleep(1000);
}
}

delegate void UpdateProgressBarDelegate(void);
System::Void UpdateProgressBar(void)
{
    progress->PerformStep();
}

delegate void UpdateLabelTextDelegate(String^ szText);
System::Void UpdateLabelText(String^ szText)
{
    label->AutoSize = false;
    label->Text = szText;
    label->AutoSize = true;
}

delegate IntPtr GetFormHandleDelegate(void);
System::IntPtr GetFormHandle(void)
{
    return this->Handle;
}

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer
/// support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
    this->components =

```

```

        gnew System::ComponentModel::Container();

        this->Size = System::Drawing::Size(300, 300);
        this->Text = L"MyForm";
        this->Padding = System::Windows::Forms::Padding(0);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
    }
#pragma endregion
};
}

```

How it works...

The code is pretty much the same here, but we chose the `Task` object in order to run the threads in the background, because when the user chooses not to continue, the application will exit and the working thread needs to exit forcibly. We could also choose the `Thread` object with the `IsBackground` property set. As we want to use the `PostMessage` API, we need to include the `User32` library, as shown in the following code:

```
#pragma comment (lib, "User32.lib")
```

When we are not using the framework, the `User32` (among others) library is, by default, included, while `mscorlib.dll` isn't. This happens when we are using a framework. The `mscorlib.dll` library is included by default while `User32` isn't. Next, we will define the `END_TASK` message in order to be able to communicate between threads. We will expand our class and will add the `WndProc` overridden method to process messages for the main window. Once again, we are using the window message queue in order to signal the main thread. We must add another delegate that will invoke the `GetFormHandle` method and obtain the main window handle that the working thread needs to post the message. We've also changed the `StartAddress` method so that the thread now executes an infinite loop with sleeping periods of 1 second so that we do not waste too much processor time. After each calculation is completed, we will signal the main thread to ask the user whether the application will continue or not. If not, we will simply post the `WM_CLOSE` message and terminate the background thread (our `Task` object) and exit the application. However, if the user chooses to continue, we will perform the calculations again.

There's more...

Even though we've accomplished the given task, we were using primitive techniques for both operations. We've used `PostMessage` to signal the main thread, and the `Sleep` method to spin the thread while waiting for an answer from the user. Further on, we'll show the .NET features that will help us to achieve this with integrated (OR built-in) objects and techniques.

Locking and avoiding deadlocks

In our previous example, we had the `bContinue` variable that both threads were accessing without any restrictions on which thread can read or write in it. Well, that's usually application bottleneck and should be avoided. The Microsoft .NET framework provides us with mechanisms to avoid situations like that with very little difficulty.

Our following example will implement another class, `Lock`, similar to the previous one that we used in *Chapter 3, Managing Threads*. The `Lock` class will help us to stop one thread from reading while the other one is writing and vice versa. Unlike the C# language, which provides the `lock` operator, C++ doesn't. That's why we will implement our own lock using the `Monitor` object to lock the execution and wait while the other thread accesses the shared object.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication4`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication4;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```
#pragma once

#include <Windows.h>
#pragma comment ( lib, "User32.lib" )

#define END_TASK WM_USER + 1

namespace CLRApplication4
{
    using namespace System;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading;
    using namespace System::Threading::Tasks;

    ref class Lock
    {
    public:
        Object^ lockObject;
        Lock(Object^ lock) : lockObject(lock)
        {
            Monitor::Enter(lockObject);
        }
    protected:
        ~Lock()
        {
            Monitor::Exit(lockObject);
        }
    };

    public ref class MyForm : public Form
    {
    public:
        MyForm(void) : bContinue(true)
        {
            lockObject = gcnew Object();
            InitializeComponent();
            this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
        }
    };
}
```

```
protected:
    ~MyForm()
    {
        if (components)
        {
            delete components;
        }
        delete lockObject;
    }

    virtual void WndProc(Message% msg) override
    {
        switch (msg.Msg)
        {
            case END_TASK:
            {
                System::Windows::Forms::DialogResult^ dlgResult =
                MessageBox::Show(L"Do you want to calculate again?", L"Question",
                MessageBoxButtons::YesNo, MessageBoxIcon::Question);

                if (*dlgResult ==
                System::Windows::Forms::DialogResult::Yes)
                {
                    SetState(true);
                    progress->Value = 0;
                    label->Text = L"Calculating...";
                }
                else
                {
                    PostMessage((HWND)this->Handle.ToPointer(), WM_CLOSE,
0, 0);
                }
                break;
            }
        }

        Form::WndProc(msg);
    }

private:
    System::ComponentModel::Container ^components;
    Label^ label;
    ProgressBar^ progress;
    static const int N = 25;
```

```
bool bContinue;
Object^ lockObject;

System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
{
    InitializeApp();
}

System::Void SetState(bool bState)
{
    Lock^ lock = gcnew Lock(this->lockObject);
    bContinue = bState;
    delete lock;
}

bool QueryState(void)
{
    Lock^ lock = gcnew Lock(this->lockObject);
    bool bState = bContinue;
    delete lock;

    return bState;
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    Task^ task = gcnew Task(gcnew Action<Object^>
        (&MyForm::StartAddress), this);
```



```

        task->Start();
    }

    static System::Void StartAddress(Object^ parameter)
    {
        MyForm^ form = (MyForm^)parameter;

        GetFormHandleDelegate^ getHandle = gcnew
        GetFormHandleDelegate(form, &MyForm::GetFormHandle);
        IntPtr handle = (IntPtr)form->Invoke(getHandle);

        UpdateProgressBarDelegate^ action = gcnew
        UpdateProgressBarDelegate(form, &MyForm::UpdateProgressBar);
        UpdateLabelTextDelegate^ textAction = gcnew
        UpdateLabelTextDelegate(form, &MyForm::UpdateLabelText);

        Int64 iResult = 1;
        while (true)
        {
            if (!form->QueryState())
            {
                Thread::Sleep(1000);
                continue;
            }

            iResult = 1;

            for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
            {
                iResult *= iIndex;
                form->BeginInvoke(action);

                Thread::Sleep(100);
            }

            form->BeginInvoke(textAction, L"Result: " +
            Convert::ToString(iResult));

            PostMessage((HWND)handle.ToPointer(), END_TASK, 0, 0);

            form->bContinue = false;

            Thread::Sleep(1000);
        }
    }

```

```

    }
}

delegate void UpdateProgressBarDelegate(void);
System::Void UpdateProgressBar(void)
{
    progress->PerformStep();
}

delegate void UpdateLabelTextDelegate( String^ szText);

System::Void UpdateLabelText(String^ szText)
{
    label->AutoSize = false;
    label->Text = szText;
    label->AutoSize = true;
}

delegate IntPtr GetFormHandleDelegate(void);
System::IntPtr GetFormHandle(void)
{
    return this->Handle;
}

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer
/// support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
    this->components =
        gcnew System::ComponentModel::Container();
    this->Size = System::Drawing::Size(300, 300);
    this->Text = L"MyForm";
    this->Padding = System::Windows::Forms::Padding(0);
    this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
}
#pragma endregion
};
}

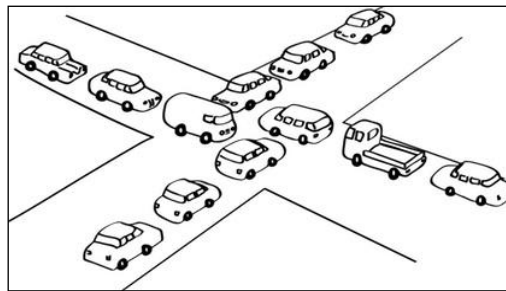
```

How it works...

First, we'll add a new `Lock` class. The idea behind the `Lock` object is to prevent a shared object from being accessed by more than one thread at one instance of time. For that purpose, we will use the `Monitor` object, which resides in the `System::Threading` namespace, and its `Enter` and `Exit` methods. To be able to use `Monitor::Enter` and `Monitor::Exit`, you must supply a pointer to `System::Object` in order to prevent the shared object from being accessed at the same time by two or more threads. We'll instantiate a pointer to the `Lock` object before accessing the shared object and will delete the `Lock` object right after accessing the shared object. This is due to the fact that the `Monitor::Exit` call is performed on the `Lock` destructor. In this way, we can safely set the value of `bContinue` or read it, without worrying whether the two threads will access it at the same time and perform unpredictable results.

There's more...

Let's discuss deadlocks. A deadlock is a situation where one or more threads are waiting for exclusive access to some object that is held by the thread that is suspended or unable to proceed for some reason. A deadlock can appear in a case where thread A is waiting for thread B, while thread B is waiting for thread A. An illustration is given in the following figure:



Deadlocks can appear in many situations where exclusive access is present. For example, we've added exclusive access to our example in a way that, when the main thread wants to set the value of `bContinue`, the working thread can't read it, or when the working thread wants to read its value, the main thread can't write. What if we suspend the working thread just after it has set a lock on the `bContinue` object? Well, we would encounter a deadlock because the main thread will be blocked until the working thread continues, but the working thread can't continue because it is suspended. There are many algorithms on how to detect, avoid, and successfully deal with deadlocks, but this is not the topic here. Deadlock detection and resolving will be implemented and explained in detail in *Chapter 8, Advanced Thread Management*. Our intention is to inform the developer about possible scenarios, while thorough research should be performed on the deadlock theory using well-documented materials in the operating system area.

Thread safety and types of the .NET framework

What is thread safety? It is the ability to perform exclusive access without interfering with separate threads. Why is this important when we are using .NET? This is because of the .NET internal implementation of certain types and operations—which are designed to be thread-safe by Microsoft itself. A good practice is to continue to develop in the same manner and try to avoid problems from designing the application to the coding itself.

In .NET, all static members are thread-safe while instance members are not. For example, if two threads want to enumerate a certain `List` collection concurrently, they will both get different enumerator objects. It is okay if they do not modify collection values inside enumeration. However, if some change during enumeration is needed, we would have to use locks.

Let's review this by showing the code that will enumerate through the list and is thread-safe:

```
List<int>^ list = gcnew List<int>( );

for (int iValue = 0; iValue < 5; iValue++)
{
    list->Add(iValue);
}

array<Object>^ ^_t1Params = gcnew array<Object>(2)
    {list, L"Task _t1" };
Task^ _t1 = gcnew Task(gcnew Action<Object>
    (&EnumerateList), _t1Params );
_t1->Start( );

array<Object>^ ^_t2Params = gcnew array<Object>(2)
    {list, L"Task _t2" };
Task^ _t2 = gcnew Task(gcnew Action<Object>
    (&EnumerateList), _t2Params );
_t2->Start( );

static void EnumerateList(Object^ parameter)
{
    array<Object>^ ^_tParams = (array<Object>^ ^) parameter;

    List<int>^ lst = (List<int>^) _tParams[0];
```

```
String^ szValues = gcnew String(L" ");

for each (int iValue in lst)
{
    szValues += iValue + L" ";
}

MessageBox::Show(szValues, (String^) _tParams[1]);
}
...
```

As we said before, each thread will get a different enumerating object. Hence, we say that it is thread-safe. Another feature that .NET provides us with is the `initonly` keyword, which enables a concurrent read operation to be safe. So when two (or more) threads are accessing such a type, it is safe. The code follows:

```
ref class SafeValue
{
public:
    SafeValue( void ){ }
    void Run( void )
    {
        Task^ _t1 = gcnew Task( gcnew Action( &AccessValue ) );
        _t1->Start( );

        Task^ _t2 = gcnew Task( gcnew Action( &AccessValue ) );
        _t2->Start( );
    }
protected:
    ~SafeValue( void ){ }
private:
    static void AccessValue( void )
    {
        MessageBox::Show( Convert::ToString( SafeValue::iValue ) );
    }
    static initonly int iValue = 10;
};

int main( )
{
    SafeValue^ safeValue = gcnew SafeValue( );
    safeValue->Run( );
}
```

Adding safety to your application usually means surrounding all exclusive access with locks, but at the same time, you should be aware of the fact that exclusion means waiting between threads, which results in spending a lot of time, and could potentially lead to a deadlock. Once again, a proper application design is needed before you actually start coding anything. Furthermore, in *Chapter 7, Understanding Concurrent Code Design*, we will thoroughly explain how to properly think and design an application in such a way that you avoid all potentially dangerous situations and known problems with respect to the task itself.

Signaling with event wait handles

In the *Understanding the .NET synchronization essentials* recipe, we used an unsafe way to access the `bContinue` shared object, and we used the `Sleep` method to release the processor for one second while waiting for the user to make a choice between continuing and exiting the application. Later, in the *Locking and avoiding deadlocks* recipe, we've improved our application to be thread-safe by adding locks to ensure exclusive access—again with the `Sleep` method inside the loop. The Microsoft .NET framework provides signaling objects that are mostly similar to `Event`, which we have explained in the previous chapters. Those are the `AutoResetEvent`, `ManualResetEvent`, `ManualResetEventSlim`, `CountDownEvent`, `Barrier`, and `Wait` and `Pulse` methods.

We chose to use `ManualResetEventSlim` in our following example due to its extremely small overhead—about 40 nanoseconds. We'll now remove the `Lock` class and access the shared object—as it's no longer needed. When the working thread completes the calculation for the first time, it will notify the main thread of the completion of the operation and will call the `ManualResetEventSlim::Wait` method to suspend execution. Later, if the user chooses to continue the calculation, we'll simply set the event object, which will signal the thread to continue execution.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication5`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```
#include <Windows.h>
#include <tchar.h>
```

```
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication5;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named MyForm. Open MyForm.h.
4. Add the following code:

```
#pragma once

#include <Windows.h>
#pragma comment ( lib, "User32.lib" )

#define END_TASK WM_USER + 1

namespace CLRApplication5
{
    using namespace System;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading;
    using namespace System::Threading::Tasks;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            mEvent = gcnew ManualResetEventSlim(false);
            InitializeComponent();
            this->Load += gcnew System::EventHandler(this,
```

```

&MyForm::Form_OnLoad);
    }
protected:
    ~MyForm()
    {
        if (components)
        {
            delete components;
        }
        delete mEvent;
    }

    virtual void WndProc(Message% msg) override
    {
        switch (msg.Msg)
        {
            case END_TASK:
            {
                System::Windows::Forms::DialogResult^ dlgResult =
                MessageBox::Show(L"Do you want to calculate again?", L"Question",
                MessageBoxButtons::YesNo, MessageBoxIcon::Question);

                if (*dlgResult ==
                System::Windows::Forms::DialogResult::Yes)
                {
                    progress->Value = 0;
                    label->Text = L"Calculating...";
                    mEvent->Set();
                }
                else
                {
                    {
                        this->Close();
                    }
                    break;
                }
            }
        }

        Form::WndProc(msg);
    }
private:
    System::ComponentModel::Container ^components;
    Label^ label;
    ProgressBar^ progress;
    static const int N = 25;

```



```
static ManualResetEventSlim^ mEvent;

System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
{
    InitializeApp();
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    Task^ task = gcnew Task(gcnew Action<Object^>
        (&MyForm::StartAddress), this);
    task->Start();
}

static System::Void StartAddress(Object^ parameter)
{
    MyForm^ form = (MyForm^)parameter;

    GetFormHandleDelegate^ getHandle = gcnew
    GetFormHandleDelegate(form, &MyForm::GetFormHandle);
    IntPtr handle = (IntPtr)form->Invoke(getHandle);

    UpdateProgressBarDelegate^ action = gcnew
    UpdateProgressBarDelegate(form, &MyForm::UpdateProgressBar);
    UpdateLabelTextDelegate^ textAction = gcnew
    UpdateLabelTextDelegate(form, &MyForm::UpdateLabelText);

    Int64 iResult = 1;
    while (true)
    {
```

```

        iResult = 1;
        mEvent->Reset();

        for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
        {
            iResult *= iIndex;
            form->BeginInvoke(action);

            Sleep(100);
        }

        form->BeginInvoke(textAction, L"Result: " +
Convert::ToString(iResult));

        PostMessage((HWND)handle.ToPointer(), END_TASK, 0, 0);

        mEvent->Wait();
    }
}

delegate void UpdateProgressBarDelegate(void);
System::Void UpdateProgressBar(void)
{
    progress->PerformStep();
}

delegate void UpdateLabelTextDelegate(String^ szText);
System::Void UpdateLabelText(String^ szText)
{
    label->AutoSize = false;
    label->Text = szText;
    label->AutoSize = true;
}

delegate IntPtr GetFormHandleDelegate(void);
System::IntPtr GetFormHandle(void)
{
    return this->Handle;
}

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer
/// support - do not modify

```

```
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
    this->components = gcnew
        System::ComponentModel::Container();
    this->Size = System::Drawing::Size(300, 300);
    this->Text = L"MyForm";
    this->Padding =
        System::Windows::Forms::Padding(0);
    this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
}
#pragma endregion
};
}
```

How it works...

This is a much better approach than the previous two examples. We've removed the `Lock` class and at the same time we avoided using the shared object, which is the primary spot for errors and interference. We've added the `ManualResetEventSlim` variable in the private scope. It is declared `static` due to required safety, which was explained before. In the `MyForm` constructor, we will create a new instance of `ManualResetEventSlim`, which is not good if you have more than one instance of application. For simplicity of the given example, we chose to do it like this. It is left to the user to additionally improve the application if more than one instance is required. Also, the `ManualResetEventSlim` constructor expects a `bool` parameter, which designates whether to set the initial state of an event to `signaled` or not. Furthermore, when the working thread starts execution, it will set the event state to `nonsignaled`. By calling the `Wait` method, it will enter a suspended state and will wait for an event until its state becomes `signaled`. In the main thread, inside the `WndProc` function, after the user chooses to continue, we'll set the event state to `signaled`. In this way, the working thread can continue with its execution.

There's more...

`ManualResetEventSlim` provides a much faster version of `ManualResetEvent` and gives better performance when the wait time is expected to be less, and when the event is not shared between different processes. When the wait time is smaller, the `ManualResetEventSlim` overhead is much less expensive, usually in situations other than waiting for some of wait handles. However, if the state doesn't become `signaled` within a short time period, the event behaves as a regular `ManualResetEvent` object.

Event-based Asynchronous Pattern

Microsoft .NET provides **Event-based Asynchronous Pattern (EAP)**, which offers multithreading capability without the need for a programmer to explicitly create or start a thread or task. Such classes are `BackgroundWorker` or `WebClient`. We'll focus on `BackgroundWorker` later in detail. Now, we'll simply point out the features of `WebClient` and will expand our factorial example to use an event instead of posting messages.

The managed `WebClient` class, among others, provides methods for downloading data in synchronous and asynchronous execution. Our interest is in asynchronous methods. The code follows

```
[HostProtectionAttribute(SecurityAction::LinkDemand, ExternalThreading
= true)]
public: void DownloadDataAsync( Uri^ address )
public: void DownloadDataAsync( Uri^ address, Object^ userToken )
public: event DownloadDataCompletedEventHandler^ DownloadDataCompleted
{
    void add (DownloadDataCompletedEventHandler^ value);
    void remove (DownloadDataCompletedEventHandler^ value);
}
```

The idea behind the asynchronous approach is that these methods create a new thread to execute the download operation, while the developer needs to register event notifications, or in other words, it must provide a method that will be called on completion of the operation.

We'll do something similar with our factorial example. We'll create an event that will fire when the calculation is over in order to ask the user whether they want to repeat the operation. Earlier, we've accomplished this by posting messages to the main thread window message queue, but it's not always possible to do that. What if we create a console application? We would not have a window there and we can't use the window message queue. That's why it's important to understand the mechanism of the event in detail.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication6`.

2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main.cpp`. Copy and paste the following code:

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication6;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```
#pragma once

namespace CLRApplication6
{
    using namespace System;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;
    using namespace System::Threading;
    using namespace System::Threading::Tasks;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();
            mEvent = gcnew ManualResetEventSlim(false);
            EndTaskEvent += gcnew EndTaskEventHandler(&MyForm::EndTask);
        }
    };
}
```

```

        updateProgressBar = gcnew UpdateProgressBarDelegate(this,
&MyForm::UpdateProgressBar);
        updateLabelText = gcnew UpdateLabelTextDelegate(this,
&MyForm::UpdateLabelText);
        updateProgressBarValue = gcnew
UpdateProgressBarValueDelegate(this,
&MyForm::UpdateProgressBarValue);
        closeForm = gcnew FormCloseDelegate(this,
&MyForm::FormClose);

        this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
    }
protected:
    ~MyForm()
    {
        if (components)
        {
            delete components;
        }
        delete mEvent;
    }
private:
    System::ComponentModel::Container ^components;

    delegate void UpdateProgressBarDelegate(void);
    System::Void UpdateProgressBar(void)
    {
        progress->PerformStep();
    }

    delegate void UpdateProgressBarValueDelegate(int iValue);
    System::Void UpdateProgressBarValue(int iValue)
    {
        progress->Value = iValue;
    }

    delegate void UpdateLabelTextDelegate(String^ szText);
    System::Void UpdateLabelText(String^ szText)
    {
        label->AutoSize = false;
        label->Text = szText;
        label->AutoSize = true;
    }

```

```
delegate void FormCloseDelegate(void);
System::Void FormClose(void)
{
    this->Close();
}

Label^ label;
ProgressBar^ progress;
static const int N = 25;
static ManualResetEventSlim^ mEvent;
delegate void EndTaskEventHandler(MyForm^);
static event EndTaskEventHandler^ EndTaskEvent;
UpdateProgressBarDelegate^ updateProgressBar;
UpdateLabelTextDelegate^ updateLabelText;
UpdateProgressBarValueDelegate^ updateProgressBarValue;
FormCloseDelegate^ closeForm;

System::Void Form_OnLoad(Object^ sender, System::EventArgs^ e)
{
    InitializeApp();
}

static System::Void EndTask(MyForm^ form)
{
    System::Windows::Forms::DialogResult^ dlgResult =
    MessageBox::Show(L"Do you want to calculate again?", L"Question",
    MessageBoxButtons::YesNo, MessageBoxIcon::Question);

    if (*dlgResult == System::Windows::Forms::DialogResult::Yes)
    {
        if (form->progress->InvokeRequired)
        {
            form->BeginInvoke( form->updateProgressBarValue,
            (Object^) 0);
        }
        else
        {
            form->progress->Value = 0;
        }

        if (form->label->InvokeRequired)
        {
            form->BeginInvoke( form->updateLabelText,
            L"Calculating...");
        }
    }
}
```

```
        else
        {
            form->label->Text = L"Calculating...";
        }

        mEvent->Set();
    }
    else
    {
        if (form->InvokeRequired)
        {
            form->BeginInvoke(form->closeForm);
        }
        else
        {
            form->Close();
        }
    }
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    Task^ task = gcnew Task(gcnew Action<Object^>
        (&MyForm::StartAddress), this);
    task->Start();
}

static System::Void StartAddress(Object^ parameter)
{

```



```

        MyForm^ form = (MyForm^)parameter;

        Int64 iResult = 1;
        while (true)
        {
            iResult = 1;

            mEvent->Reset();

            for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
            {
                iResult *= iIndex;
                form->BeginInvoke(form->updateProgressBar);
                System::Threading::Thread::Sleep(100);
            }

            form->BeginInvoke(form->updateLabelText, L"Result: " +
                Convert::ToString(iResult));

            MyForm::EndTaskEvent(form);

            mEvent->Wait();
        }
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    void InitializeComponent(void)
    {
        this->components = gcnew
            System::ComponentModel::Container();
        this->Size = System::Drawing::Size(300, 300);
        this->Text = L"MyForm";
        this->Padding =
            System::Windows::Forms::Padding(0);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
    }
#pragma endregion
};
}

```

How it works...

We are presenting event usage and we've removed posting messages to the main window message queue. There is an advantage to this. We're now independent of whether or not the application has a window. However, there is a down side. As the working thread fires an event (calling a routine), all manipulation with UI elements, such as changing the value of the progress bar or label text, must be done by invoking a proper method. We intentionally created an example like this only to point out cross-thread operations. This is very important to understand before doing anything asynchronously in .NET, which involves UI controls' manipulation. Let's review the code. We've changed the `MyForm` constructor in a way that the first six lines of code are creating new instances of events and delegates. We've changed delegates to be class attributes instead of locally-declared variables because more than one method will use them. We've added a `static` routine `EndTask` that will be called on raising the `EndTaskEventHandler` event. And finally, inside the `StartAddress` method, instead of using the `PostMessage` API, we are raising an event to ask the user if they want to continue. So, by following a slightly different approach, we're using EAP in the proper way, which the .NET framework provides.

There's more...

In order for you to understand the built EAP, we'll give an example of the `WebClientDownloadDataAsync` method usage:

```
...
String^ szUrl = L"http://cplusplus.expert.its.me/testfile.txt";
WebClient^ client = gcnew WebClient();
client->DownloadDataCompleted +=
    gcnew DownloadDataCompletedEventHandler(
        &MyForm::AsyncDownloadDataCompleted);
client->DownloadDataAsync(gcnew Uri(szUrl));
...

static void AsyncDownloadDataCompleted(Object^ sender,
    DownloadDataCompletedEventArgs^ e)
{
    String^ szResult;
    System::Text::Encoding^ enc = System::Text::Encoding::UTF8;
    szResult = enc->GetString(e->Result);

    if (e->Error == nullptr
        && !String::IsNullOrEmpty(szResult))
    {
        System::IO::File::WriteAllText(
            L"C:\\testFile.txt", szResult);
        MessageBox::Show(L"Download complete.");
    }
}
```

Similar to our previous example, it is required to register for an event, and we are using the `AsyncDownloadDataCompleted` method for that. When we call `WebClient::DownloadDataAsync`, CLR creates a new thread that will perform the download operation in parallel and, on completion, it will raise an event with a subsequent call to the `AsyncDownloadDataCompleted` method.

Using the BackgroundWorker class

As we said before, EAP is a very good mechanism, which enables a programmer to create multithreaded applications easily and with much less coding. The `BackgroundWorker` object is a general-purpose implementation of the EAP, and has many good features, such as the handy cancelation model, the ability to safely update another thread's UI controls, the reporting progress feature, exception forwarding, raising completion event, and more. We'll now implement our factorial example using the `BackgroundWorker` class that resides in the `System::ComponentModel` namespace.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication7`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication7;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
```

```

Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MyForm());

return 0;
}

```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```

#pragma once

namespace CLRApplication7
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();

            this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);
        }
    protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }
    private:
        System::ComponentModel::Container ^components;
        Label^ label;
        ProgressBar^ progress;

```

```

        static const int N = 25;

        System::Void Form_OnLoad(System::Object^ sender,
        System::EventArgs^ e)
        {
            InitializeApp();
        }

        System::Void InitializeApp(void)
        {
            label = gcnew Label();
            label->Location = Point(2, 25);

            progress = gcnew ProgressBar();
            progress->Location = Point(2, 235);
            progress->Width = 280;
            progress->Height = 25;
            progress->Step = 1;
            progress->Maximum = MyForm::N;
            progress->Value = 0;

            this->Controls->Add(label);
            this->Controls->Add(progress);

            BackgroundWorker^ bckWorker = gcnew BackgroundWorker();
            bckWorker->WorkerReportsProgress = true;
            bckWorker->ProgressChanged +=
                gcnew ProgressChangedEventHandler(this,
                &MyForm::ProgressChanged);

            bckWorker->RunWorkerCompleted +=
                gcnew RunWorkerCompletedEventHandler(this,
                &MyForm::RunningCompleted);
            bckWorker->DoWork += gcnew DoWorkEventHandler(this,
                &MyForm::StartAddress);
            bckWorker->RunWorkerAsync(bckWorker);
        }

        System::Void StartAddress(Object^ sender, DoWorkEventArgs^ e)
        {
            BackgroundWorker^ bckWorker = (BackgroundWorker^)
            e->Argument;

            Int64 iResult = 1;
            for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)

```

```

        {
            iResult *= iIndex;

            bckWorker->ReportProgress(iIndex);

            System::Threading::Thread::Sleep(100);
        }

        e->Result = iResult;
    }

    System::Void ProgressChanged(Object^ sender,
    ProgressChangedEventArgs^ e)
    {
        progress->Value = e->ProgressPercentage;
    }

    System::Void RunningCompleted(Object^ sender,
    RunWorkerCompletedEventArgs^ e)
    {
        label->AutoSize = false;
        label->Text = L"Result: " + Convert::ToString(e->Result);
        label->AutoSize = true;
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    void InitializeComponent(void)
    {
        this->components = gcnew
            System::ComponentModel::Container();
        this->Size = System::Drawing::Size(300, 300);
        this->Text = L"MyForm";
        this->Padding =
            System::Windows::Forms::Padding(0);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
    }
#pragma endregion
    };
}

```

How it works...

As you will notice, the deeper we go into the framework's features, the less development time and coding effort is needed. In our previous variations of the factorial application, we had to implement delegates in order to manipulate with different thread UI controls. `BackgroundWorker` does all that for us with integrated events such as `ProgressChanged` or `RunWorkerCompleted`. These features make coding a lot easier. Here, you need not worry about cross-thread operations, which are usually a programmer's nightmare. Programming logic also becomes a lot simpler. Let's review the code. Major changes were done inside `InitializeApp` where we had to properly initialize the `BackgroundWorker` object; we had to provide method delegates for three events and set the `WorkerReportsProgress` property to `true`. Note that, in order to match the event prototype, we had to change the `StartAddress` prototype's signature to the following:

```
System::Void StartAddress(Object^ sender, DoWorkEventArgs^ e)
```

We can still pass an argument via the `RunWorkerAsync` method and obtain it later, inside the `StartAddress` method via the `Argument` attribute of the `DoWorkEventArgs` object. When we want to raise a `ProgressChanged` event, we simply call the `BackgroundWorker::ReportProgress` method and pass an integer as an argument, which tells how much of the progress will be updated. At the end, we want to know when the working thread completes execution. That's why we've registered the `RunWorkerCompleted` event. When the thread exits, that event is raised.

There's more...

Unlike the `Thread` class, which is implemented as a sealed class, `BackgroundWorker` is not. This means that you could subclass and derive your own version of the `BackgroundWorker` object, which will be expanded for any need that you have. This might come in handy when you have a complex task, such as a weather forecast prediction, where your `BackgroundWorker` class should have numerous attributes in order to save a lot of data that needs to be processed. Note that you need to use the `OnDoWork` virtual method in case of deriving the `BackgroundWorker` class. One variation should look like this:

```
ref class Worker : BackgroundWorker
{
public:
    Worker( ) : BackgroundWorker( )
    {
        WorkerReportsProgress = true;
    }

protected:
    virtual void OnDoWork( DoWorkEventArgs^ e ) override
    {
```

```

Worker^ bckWorker = ( Worker^ ) e->Argument;

Int64 iResult = 1;
for (int iIndex = 1; iIndex <= Worker::N; iIndex++)
{
    iResult *= iIndex;
    bckWorker->ReportProgress( iIndex );

    System::Threading::Thread::Sleep( 100 );
}

e->Result = iResult;
}
private:
    static const int N = 25;
    // additional fields
};

```

The use would be as follows:

```

Worker^ worker = gcnew Worker( );
worker->RunWorkerAsync( worker );

```

Interrupting, aborting, and safe canceling the thread execution

Some tasks require ways to forcibly interrupt the execution of the thread. In our previous examples, we've used the `Task` and `BackgroundWorker` objects due to their background operations and the ability to be forcibly closed on exiting the application. Now, let's explain the .NET features that target normal thread interruption and safe cancelation. Each blocking method will block the thread forever if some unblocking condition is never met or the timeout is not set. However, sometimes, you need to cancel thread execution and maybe continue the execution of the application without allowing the thread to continue. In other situations, you need to cancel the thread, for example, to receive user input, and start the thread again. The methods for ending the thread are `Thread::Interrupt` and `Thread::Abort`.

When you call the `Interrupt` method, the thread that's not blocked continues execution until it is blocked the next time, at which point a `ThreadInterruptedException` is thrown, while calling an interrupt on a blocked thread forcibly terminates it and again throws `ThreadInterruptedException`. As the `Abort` method has an effect on non-blocked threads, you should almost always use `Abort` instead of `Interrupt`. When you use the `Abort` method, a similar exception is thrown: `ThreadAbortException`. The `Abort` and `Interrupt` methods are not considered safe. This is because if you have an unmanaged code execution inside the thread routine, when you use `Abort`, the execution will continue as far as the next managed code statement is executed. On the other hand, `Interrupt` won't affect execution until the thread isn't blocked.

In our following example, we'll show how to abort a thread using the `Abort` method.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication8`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main.cpp`. Open `main.cpp`.

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"
```

```
using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication8;
```

```
[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```
#pragma once

namespace CLRApplication8
{
    using namespace System;
    using namespace System::Threading;
    using namespace System::Windows::Forms;
```

```

using namespace System::Drawing;

public ref class MyForm : public Form
{
public:
    MyForm(void)
    {
        InitializeComponent();
        this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);

        updateProgressBar = gcnew UpdateProgressBarDelegate(this,
&MyForm::UpdateProgressBar);
        updateLabelText = gcnew UpdateLabelTextDelegate( this,
&MyForm::UpdateLabelText);
        this->FormClosing += gcnew FormClosingEventHandler(this,
&MyForm::FormClosingEvent);
    }
protected:
    ~MyForm()
    {
        if (components)
        {
            delete components;
        }
    }
private:
    delegate void UpdateProgressBarDelegate(void);
    System::Void UpdateProgressBar(void)
    {
        progress->PerformStep();
    }

    delegate void UpdateLabelTextDelegate(String^ szText);
    System::Void UpdateLabelText(String^ szText)
    {
        label->AutoSize = false;
        label->Text = szText;
        label->AutoSize = true;
    }

    System::ComponentModel::Container ^components;
    Label^ label;
    ProgressBar^ progress;
    static const int N = 25;

```

```
Thread^ worker;
UpdateProgressBarDelegate^ updateProgressBar;
UpdateLabelTextDelegate^ updateLabelText;

System::Void Form_OnLoad(System::Object^ sender,
System::EventArgs^ e)
{
    InitializeApp();
}

System::Void FormClosingEvent(Object^ sender,
FormClosingEventArgs^ e)
{
    System::Windows::Forms::DialogResult^ dlgResult =
    MessageBox::Show(L"Do you want to abort execution? ", L"Question",
    MessageBoxButtons::YesNo, MessageBoxIcon::Question);

    if (*dlgResult == System::Windows::Forms::DialogResult::Yes)
    {
        worker->Abort();
    }
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    worker = gcnew Thread(gcnew ParameterizedThreadStart(
    &MyForm::StartAddress));
    worker->Start(this);
}
```

```

    }

    static System::Void StartAddress(Object^ parameter)
    {
        MyForm^ form = (MyForm^)parameter;

        Int64 iResult = 1;
        for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
        {
            iResult *= iIndex;
            form->BeginInvoke(form->updateProgressBar);

            Thread::Sleep(100);
        }

        form->BeginInvoke(form->updateLabelText, L"Result: " +
            Convert::ToString(iResult));
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    void InitializeComponent(void)
    {
        this->SuspendLayout();
        this->AutoScaleDimensions =
            System::Drawing::SizeF(6, 13);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
        this->Size = System::Drawing::Size(284, 261);
        this->MaximizeBox = false;
        this->MinimizeBox = false;
        this->Name = L"MyForm";
        this->Text = L"MyForm";
        this->ResumeLayout(false);
    }
#pragma endregion
};
}

```

How it works...

We're using the same example as before. Our only intention was to show how to use the `Abort` method to the user. We recommend that you use these methods with caution, and only on your own implementation of threads, simply because you would then know exactly what would happen.

Safe cancellation

Even though the .NET framework implements the `Abort` and `Interrupt` methods, you should avoid their usage whenever possible. Another approach that we would recommend is to use (when possible) the `BackgroundWorker.WorkerSupportsCancellation` property and its `CancelAsync` method.

Our following example will show how to abort the `BackgroundWorker` thread using the `CancelAsync` method.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication9`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`.

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"
```

```
using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication9;
```

```
[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
```

```

Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MyForm());

return 0;
}

```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```

#pragma once

namespace CLRApplication9
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();
            this->Load += gcnew System::EventHandler(this,
&MyForm::Form_OnLoad);

            this->FormClosing += gcnew FormClosingEventHandler(this,
&MyForm::FormClosingEvent);
        }
    protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }
    private:
        System::ComponentModel::Container ^components;
        Label^ label;
        ProgressBar^ progress;
        static const int N = 25;

```

```
BackgroundWorker^ bckWorker;

System::Void Form_OnLoad(Object^ sender, EventArgs^ e)
{
    InitializeApp();
}

System::Void FormClosingEvent(Object^ sender,
FormClosingEventArgs^ e)
{
    System::Windows::Forms::DialogResult^ dlgResult =
    MessageBox::Show(L"Do you want to abort execution?", L"Question",
    MessageBoxButtons::YesNo, MessageBoxIcon::Question);

    if (*dlgResult == System::Windows::Forms::DialogResult::Yes)
    {
        bckWorker->CancelAsync();
    }
    else
    {
        {
            e->Cancel = true;
        }
    }
}

System::Void InitializeApp(void)
{
    label = gcnew Label();
    label->Location = Point(2, 25);

    progress = gcnew ProgressBar();
    progress->Location = Point(2, 235);
    progress->Width = 280;
    progress->Height = 25;
    progress->Step = 1;
    progress->Maximum = MyForm::N;
    progress->Value = 0;

    this->Controls->Add(label);
    this->Controls->Add(progress);

    bckWorker = gcnew BackgroundWorker();
    bckWorker->WorkerReportsProgress = true;
    bckWorker->WorkerSupportsCancellation = true;
    bckWorker->ProgressChanged +=
        gcnew ProgressChangedEventHandler(this,
        &MyForm::ProgressChanged);
```

```

        bckWorker->RunWorkerCompleted +=
            gcnew RunWorkerCompletedEventHandler(this,
                &MyForm::RunningCompleted);
        bckWorker->DoWork += gcnew DoWorkEventHandler( this,
            &MyForm::StartAddress);
        bckWorker->RunWorkerAsync(bckWorker);
    }

    System::Void StartAddress(Object^ sender, DoWorkEventArgs^ e)
    {
        BackgroundWorker^ worker = (BackgroundWorker^) e->Argument;

        Int64 iResult = 1;
        for (int iIndex = 1; iIndex <= MyForm::N; iIndex++)
        {
            iResult *= iIndex;
            worker->ReportProgress(iIndex);

            System::Threading::Thread::Sleep(100);

            if (worker->CancellationPending)
            {
                return;
            }
        }

        e->Result = iResult;
    }

    System::Void ProgressChanged(Object^ sender,
        ProgressChangedEventArgs^ e)
    {
        progress->Value = e->ProgressPercentage;
    }

    System::Void RunningCompleted(Object^ sender,
        RunWorkerCompletedEventArgs^ e)
    {
        label->AutoSize = false;
        label->Text = L"Result: " + Convert::ToString(e->Result);
        label->AutoSize = true;
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer

```



```
/// support - do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
    this->components = gcnew
        System::ComponentModel::Container();
    this->Size = System::Drawing::Size(300, 300);
    this->Text = L"MyForm";
    this->Padding =
        System::Windows::Forms::Padding(0);
    this->AutoScaleMode =
        System::Windows::Forms::AutoScaleMode::Font;
}
#pragma endregion
};
}
```

How it works...

We've achieved the same thing, only this time safely, again using .NET's integrated features—the `BackgroundWorker::CancelAsync` method and the `BackgroundWorker` property `WorkerSupportsCancellation`. It is not always possible to implement the described mechanism, especially in situations where you must have full control over execution, and thread cancellation must be performed instantly. In such a case, you could use a `bool` variable (or another type) to mimic the described property, surrounded with a `Lock` object, to ensure exclusive access.

Let's review the code. We've added `FormClosingEventHandler` to handle the form closing event, and we've set the `BackgroundWorker` property `WorkerSupportsCancellation` to `true`. Inside `StartAddress`, we've added the following statement:

```
if (worker->CancellationPending)
{
    return;
}
```

This will cancel the thread execution in an orderly fashion, if needed.

There's more...

Let's review the `Task` class at the end. A very good and new feature that the .NET `Task` object provides is the `ContinueWith` method. When the task ends, you can immediately prepare it for another task using *continuation*, to continue the method.

Continuation is good for tasks that should be executed only after a certain task is done. For example, if we have two tasks where a second task must use the result of the first task, then the second task should be considered as a continuation task. When you use the `ContinueWith` method, the dispatcher reallocates the thread to continue with the start address supplied as the argument of the `ContinueWith` method. The code follows:

```
static System::Void StartAddress(Object^ parameter);
static System::Void ContinueAddress(Task^ task);
...
Task^ task = gcnew Task(gcnew Action<Object^>(&StartAddress), this);
task->Start( );
Task^ continueTask = task->ContinueWith(gcnew Action<Task^>(&ContinueAddress));
...
```

The `ContinueWith` method has many overloads, and you can use it with different values, such as `TaskContinuationOptions`, `TaskScheduler`, or `TaskCreationOptions`. More information can be found on MSDN (<http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskcontinuationoptions%28v=vs.110%29.aspx?cs-save-lang=1&cs-lang=cpp#code-snippet-1>). Among the variety of benefits, the `Task` object provides the `WaitAll` and `WaitAny` methods that are similar to the `WaitForSingleObject` and `WaitForMultipleObjects` APIs.

Non-blocking synchronization

Among many features that .NET provides, programmers must be aware of certain CLR behavior so as to avoid mistakes when using multiple threads. Both, the Microsoft CLI/C++ compiler as well as CLR can reorder certain statements at runtime for the sake of improving efficiency or caching optimizations. For these reasons, it is not always enough to properly organize the code, it must be protected as well. That's why .NET provides the `MemoryBarrier` class that prevents the effects of instruction reordering and read/write caching, as shown in the following class:

```
...
ref class TestClass
{
public:
    TestClass( )
    {
        bState = false;
        iValue = 0;
    }
    static void SetMethod( )
    {
        iValue = 5;
    }
};
```

```
        bState = true;
    }
    static void GetMethod( )
    {
        if (bState)
        {
            MessageBox::Show( Convert::ToString( iValue ) );
        }
    }
private:
    static bool bState;
    static int iValue;
};

ref class TestWorker
{
public:
    TestWorker( )
    {
        TestClass _t;
        Task^ _t1 = gcnew Task( gcnew Action( &_t.SetMethod ) );
        _t1->Start( );
        Task^ _t2 = gcnew Task( gcnew Action( &_t.GetMethod ) );
        _t2->Start( );
    }
};

...
```

In your opinion, is it possible that `GetMethod` shows 0 for `iValue`? Well, for the mentioned reasons—yes. The simplest solution is to create a *memory fence* or surround usage of the attribute `bState` with `MemoryBarrier` statements, as shown in the following code:

```
...
static void SetMethod( )
{
    iValue = 5;
    Thread::MemoryBarrier( );
    bState = true;
    Thread::MemoryBarrier( );
}
static void GetMethod( )
{
    Thread::MemoryBarrier( );
    if (bState)
    {
```

```

        Thread::MemoryBarrier( );
        MessageBox::Show( Convert::ToString( iValue ) );
    }
}
...

```

This is called *full fence*, and it takes around 10 nanoseconds for the execution of this memory protection.

There is another feature that the C++ compiler provides: the `volatile` keyword. The use of the `volatile` keyword instructs the compiler to create an *acquire fence* for each read operation on a `volatile` memory location, and a *release fence* for each write operation on a memory location that is declared as `volatile`. The *acquire fence* prevents read/write operations to be moved above the fence, and the *release fence* prevents read/write operations to be moved below the fence. Implementation using a `volatile` memory location could be as follows:

```

...
ref class TestClass
{
public:
    TestClass( )
    {
        bState = false;
        iValue = 0;
    }
    static void SetMethod( )
    {
        iValue = 5;
        bState = true;
    }
    static void GetMethod( )
    {
        if (bState)
        {
            MessageBox::Show( Convert::ToString( iValue ) );
        }
    }
private:
    static volatile bool bState;
    static int iValue;
};
...

```

Using `volatile` memory objects or so-called *half fences* is much faster due to even more space for runtime optimization.

Signaling with Wait and Pulse

In one of our previous examples, we've used the `Lock` class whose implementation uses `Monitor::Enter` and `Monitor::Exit`. Another feature of the `Monitor` object is very important: the `Monitor::Wait` and `Monitor::Pulse` methods (and the `Pulse` variation `PulseAll`). These methods also make EAP constructs. Their ability to wait until a signal is received from another thread makes them excellent for usage. One of the most important features is that they prevent spinning, because of which no processor time is wasted. The only constraint is that they must be enclosed with locks (`Monitor::Enter` and `Monitor::Exit`). Once again, we're introducing the `Lock` class in order to gain exclusive access to shared objects.

Our following example will show how to use the static `Wait` and `Pulse` methods as a technique to synchronize multiple threads.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRApplication10`.
2. Open **Solution Explorer** and right-click on **Source files**. Add a new source file named `main`. Open `main.cpp`.

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRApplication10;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);

    Application::Run(gcnew MyForm());

    return 0;
}
```

3. Open **Solution Explorer** and right-click on **Header files**. Under **Visual C++**, select **UI**. Add a new Windows Form named `MyForm`. Open `MyForm.h`.
4. Add the following code:

```
#pragma once

namespace CLRApplication10
{
    using namespace System;
    using namespace System::Threading;
    using namespace System::Threading::Tasks;
    using namespace System::Windows::Forms;
    using namespace System::Drawing;

    ref class Lock
    {
        Object^ lockObject;
    public:
        Lock(Object^ lock) : lockObject(lock)
        {
            Monitor::Enter(lockObject);
        }
    protected:
        ~Lock()
        {
            Monitor::Exit(lockObject);
        }
    };

    public ref class MyForm : public Form
    {
    public:
        MyForm(void) : syncHandle(gcnew Object())
        {
            InitializeComponent();
            this->Load += gcnew EventHandler(this,
                &MyForm::Form_OnLoad);
        }
    protected:
        ~MyForm()
        {
            if (components)
            {
                delete components;
            }
        }
    };
}
```

```
    }  
    }  
    virtual void SetWait(void)  
    {  
        Lock^ lock = gcnew Lock(syncHandle);  
        Monitor::Wait(syncHandle);  
        delete lock;  
    }  
    virtual void SetPulse(void)  
    {  
        Lock^ lock = gcnew Lock(syncHandle);  
        Monitor::Pulse(syncHandle);  
        delete lock;  
    }  
private:  
    System::ComponentModel::Container ^components;  
    Label^ label;  
    ProgressBar^ progress;  
    static const unsigned N = 25;  
    initonly Object^ syncHandle;  
  
    System::Void Form_OnLoad(Object^ sender, EventArgs^ e)  
    {  
        InitializeApp();  
    }  
  
    delegate void TaskCompletedDelegate(UInt64 uResult);  
    void TaskCompleted(UInt64 uResult)  
    {  
        label->AutoSize = false;  
        label->Text = L"Result: " + uResult;  
        label->AutoSize = true;  
  
        System::Windows::Forms::DialogResult^ dlgResult =  
        MessageBox::Show(L"Do you want to continue?", L"Question",  
        MessageBoxButtons::YesNo, MessageBoxIcon::Question);  
  
        if (*dlgResult == System::Windows::Forms::DialogResult::Yes)  
        {  
            this->SetPulse();  
            progress->Value = 0;  
            label->Text = L"Calculating...";  
        }  
        else
```

```

        {
            this->Close();
        }
    }

    delegate void UpdateProgressBarDelegate(void);
    System::Void UpdateProgressBar(void)
    {
        progress->PerformStep();
    }

    System::Void InitializeApp(void)
    {
        label = gcnew Label();
        label->Location = Point(2, 25);

        progress = gcnew ProgressBar();
        progress->Location = Point(2, 235);
        progress->Width = 280;
        progress->Height = 25;
        progress->Step = 1;
        progress->Maximum = MyForm::N;
        progress->Value = 0;

        this->Controls->Add(label);
        this->Controls->Add(progress);

        Task^ task = gcnew Task(gcnew Action<Object^>
            (&MyForm::StartAddress), gcnew array<Object^>(2)
            { this, task });
        task->Start();
    }

    static System::Void StartAddress(Object^ parameter)
    {
        array<Object^>^ parameters = (array<Object^>^) parameter;

        MyForm^ form = (MyForm^)parameters[0];
        Task^ task = (Task^)parameters[1];

        TaskCompletedDelegate^ taskCompleted = gcnew
        TaskCompletedDelegate(form, &MyForm::TaskCompleted);
        UpdateProgressBarDelegate^ updateProgressBar = gcnew
        UpdateProgressBarDelegate(form, &MyForm::UpdateProgressBar);

        UInt64 uResult = 1;
    }

```



```

        while (true)
        {
            uResult = 1;

            for (unsigned uIndex = 1; uIndex <= MyForm::N; uIndex++)
            {
                uResult *= uIndex;
                form->progress->BeginInvoke( updateProgressBar);

                Thread::Sleep(100);
            }

            form->BeginInvoke(taskCompleted, uResult);

            form->SetWait();
        }
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>
    void InitializeComponent(void)
    {
        this->SuspendLayout();
        this->AutoScaleDimensions =
            System::Drawing::SizeF(6, 13);
        this->AutoScaleMode =
            System::Windows::Forms::AutoScaleMode::Font;
        this->Size = System::Drawing::Size(284, 300);
        this->MaximizeBox = false;
        this->MinimizeBox = false;
        this->Name = L"MyForm";
        this->Text = L"MyForm";
        this->ResumeLayout(false);
    }
#pragma endregion
};
}

```

How it works...

Implementation of the `Lock` class remains the same, while the `MyForm` class introduces new methods: `SetWait` and `SetPulse`. These methods are declared `protected` in order to allow other programmers who may create a derivation from the `MyForm` class further on to expand its behavior. As we said before, a programmer must enclose the `Wait` and `Pulse` statements with locks. At a `private` scope, we expanded the `MyForm` class with the `initonly` `syncHandle` attribute. This attribute will be used as a synchronization object for all the `Monitor` method calls. The only difference with the `StartAddress` method is that we've used the `Control::BeginInvoke` method to asynchronously invoke the `TaskCompleted` routine. This is required for the following reason: if we had invoked (synchronously) the `TaskCompleted` method from the working thread, we would need to create delegates for all statements inside `TaskCompleted` that manipulates with the UI created by the main thread. This would of course raise an exception about cross-thread operations. By invoking `TaskCompleted` asynchronously, we are avoiding all this coding and are giving control to the main thread. In order to wait until the user provides an answer for task continuation, we will use the `Monitor::Wait` method. The working thread then enters a suspended state, and no processor time is spent. So, the control has been given to the main thread. Its task now is to ask the user whether they want to continue; if the answer is "Yes", then it will call `Monitor::Pulse` in order to change the working thread's state to `nonsignaled`. In this way, the working thread can continue execution and repeat the process.

There's more...

`Monitor::Wait` and `Monitor::Pulse`, in combination with locks, are providing excellent performance, and the code itself is much smaller. The same features that `ResetEvent` (Auto or Manual) or `Semaphore` provides can be achieved with these routines. If you are aware that `Pulse` takes around 100 nanoseconds for execution despite knowing that 400 nanoseconds is taken by the `Set` method to signal an event handle, then you should consider using `Pulse` whenever possible.

The Barrier class

Among the synchronization base classes for task coordination, an excellent feature of .NET is the `Barrier` class. Even though locking is an excellent mechanism, you will almost certainly find yourself in a situation where some very complex task issues another mechanism that you would need to solve successfully. The `Barrier` class enables you to temporarily stop (pause) execution of a task or a collection of tasks at a certain point in the application and continue when all tasks reach that point. For synchronization, this feature is important in order for a series of multiple tasks to be executed in parallel steps.

When an application creates a `Barrier` object, it has to specify the number of tasks in `set` that will be synchronized. This value can be observed as a task counter that is internally maintained inside the `Barrier` object. This value can be incremented using the `Barrier::AddParticipant` method, or decremented using the `Barrier::RemoveParticipant` method. When the task reaches the synchronization point, it calls the `Barrier::SignalAndAwait` method, which decrements the thread counter inside the `Barrier` object. If this counter is greater than zero, the thread is suspended. Only when the counter reaches zero (no more threads to wait), all tasks that wait will be released, and only then will they continue execution.

The `Barrier` object provides the `ParticipantCount` property, which returns the number of tasks that need to be synchronized, along with the `ParticipantRemaining` property, which returns the number of tasks needed for a `SignalAndAwait` call to be made before the barrier is reached, and the suspended tasks continue execution. You can also provide the `postPhaseAction` method delegate as an argument of the `Barrier` constructor, which will be called when all tasks reach the barrier. The `Barrier` object is passed as a parameter into this method. `Barrier` is not enabled, and tasks are not resumed until this method returns.

Our following example will calculate a certain mathematical formula, and plot its graph in order to show synchronization using the `Barrier` object.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ CLR empty project application named `CLRBarrier`.
2. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```
#include <Windows.h>
#include <tchar.h>
#include "MyForm.h"

using namespace System::Windows::Forms;
using namespace System::Threading;
using namespace CLRBarrier;

[STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hThis, HINSTANCE hPrev, LPTSTR
szCommandLine, int iCmdShow)
{
```

```

Application::EnableVisualStyles();
Application::SetCompatibleTextRenderingDefault(false);

Application::Run(gcnew MyForm());

return 0;
}

```

3. Open **Solution Explorer** and right-click on **Header file**. Under **Visual C++**, select **UI**. Add a new Windows Form named MyForm. Open MyForm.h.
4. Add the following code:

```

#pragma once

namespace CLRBarrier
{
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections::Generic;
    using namespace System::Windows::Forms;
    using namespace System::Threading;
    using namespace System::Threading::Tasks;
    using namespace System::Drawing;

    public ref class MyForm : public Form
    {
    public:
        MyForm(void)
        {
            InitializeComponent();

            iWidth = 800;
            iHeight = 600;

            bitmapGraph = gcnew Bitmap(iWidth, iHeight);
            colorArray = gcnew array<Color>(iWidth * iHeight);

            color = Color::FromArgb(0, 0, 255);

            progress->Maximum = iWidth / 2;
        }
    protected:
        ~MyForm()

```

```
{
    if (components)
    {
        delete components;
    }
}

private:
    System::Windows::Forms::GroupBox^ groupBox;
    System::Windows::Forms::Label^ status;
    System::Windows::Forms::ProgressBar^ progress;
    System::Windows::Forms::Button^ btnStart;
    System::Windows::Forms::PictureBox^ canvas;
    System::ComponentModel::Container ^components;

    int iWidth;
    int iHeight;
    array<Color>^ colorArray;
    Bitmap^ bitmapGraph;
    Barrier^ barrier;
    Color color;

    delegate void UpdateProgressDelegate(void);
    System::Void UpdateProgress(void)
    {
        progress->PerformStep();
    }

    delegate void SetCanvasDelegate(void);
    void SetCanvas(void)
    {
        canvas->BackColor = Color::Black;
        canvas->Image = bitmapGraph;
    }

    delegate void SetLabelTextDelegate(String^ szText);
    void SetLabelText(String^ szText)
    {
        status->Text = szText;
    }

    System::Void EndTask(Barrier^ b)
    {
```

```

        FillBitmap();

        SetLabelTextDelegate^ setLabelText = gcnew
SetLabelTextDelegate(this, &MyForm::SetLabelText);
        status->BeginInvoke(setLabelText, L"Done.");

        SetCanvasDelegate^ canvasDelegate = gcnew
SetCanvasDelegate(this, &MyForm::SetCanvas);
        canvas->BeginInvoke(canvasDelegate);
    }

    System::Void btnStart_Click(Object^ sender, EventArgs^ e)
    {
        status->Text = L"Calculating...";

        Action^ action = gcnew Action(this,
&MyForm::CalculateValues);

        barrier = gcnew Barrier(2, gcnew Action<Barrier^>(this,
&MyForm::EndTask));

        Tasks::Parallel::Invoke(action, action);
    }

    System::Void FillBitmap(void)
    {
        for (int iYPosition = 0; iYPosition < iHeight; iYPosition++)
        {
            for (int iXPosition = 0; iXPosition < iWidth;
iXPosition++)
            {
                bitmapGraph->SetPixel(iXPosition, iYPosition,
colorArray[(iXPosition + (iWidth * iYPosition))]);
            }
        }
    }

    System::Void CalculateValues(void)
    {
        int iLeft = iWidth / 2;
        int iArea = iLeft * iLeft;
    }

```

```

        int iTop = iHeight / 2;

        UpdateProgressDelegate^ updateProgress = gcnew
        UpdateProgressDelegate(this, &MyForm::UpdateProgress);

        for (int iXAxis = 0; iXAxis < iLeft; iXAxis++)
        {
            int iSurface = iXAxis * iXAxis;
            double dSquare = Math::Sqrt(iArea - iSurface);

            for (double dIndex = -dSquare; dIndex < dSquare; dIndex +=
3)
            {
                double dRadius = Math::Sqrt(iSurface + dIndex * dIndex)
/ iLeft;
                double dBottom = (dRadius - .96) * Math::Sin(32 *
dRadius);
                double dYAxis = dIndex / 3 + (dBottom * iTop);

                SetPixelValue(colorArray, (int) (-iXAxis + (iWidth /
2)), (int) (dYAxis + (iHeight / 2)));
                SetPixelValue(colorArray, (int) (iXAxis + (iWidth / 2)),
(int) (dYAxis + (iHeight / 2)));
            }

            progress->BeginInvoke(updateProgress);
        }

        barrier->SignalAndWait();
    }

    System::Void SetPixelValue(array<Color>^ graphArray, int
iXPos, int iYPos)
    {
        int iIndex = (iXPos + iYPos * iWidth);
        graphArray[iIndex] = color;
    }

#pragma region Windows Form Designer generated code
    /// <summary>
    /// Required method for Designer
    /// support - do not modify
    /// the contents of this method with the code editor.
    /// </summary>

```

```
void InitializeComponent(void)
{
    this->groupBox = (gcnew
        System::Windows::Forms::GroupBox());
    this->status = (gcnew
        System::Windows::Forms::Label());
    this->progress = (gcnew
        System::Windows::Forms::ProgressBar());
    this->btnStart = (gcnew
        System::Windows::Forms::Button());
    this->canvas = (gcnew
        System::Windows::Forms::PictureBox());
    this->groupBox->SuspendLayout();
    (cli::safe_cast
        <System::ComponentModel::ISupportInitialize^>
        (this->canvas))->BeginInit();
    this->SuspendLayout();
    //
    // groupBox
    //
    this->groupBox->Controls->Add(this->status);
    this->groupBox->Controls->Add(this->progress);
    this->groupBox->Controls->Add(this->btnStart);
    this->groupBox->Location =
        System::Drawing::Point(9, 615);
    this->groupBox->Name = L"groupBox";
    this->groupBox->Size =
        System::Drawing::Size(800, 59);
    this->groupBox->TabIndex = 0;
    this->groupBox->TabStop = false;
    //
    // status
    //
    this->status->Location =
        System::Drawing::Point(13, 20);
    this->status->Name = L"status";
    this->status->Size =
        System::Drawing::Size(270, 27);
    this->status->TabIndex = 2;
    this->status->TextAlign =
        System::Drawing::ContentAlignment::MiddleLeft;
    //
    // progress
```



```
//
this->progress->Location =
    System::Drawing::Point(301, 20);
this->progress->Name = L"progress";
this->progress->Size =
    System::Drawing::Size(387, 27);
this->progress->Step = 1;
this->progress->TabIndex = 1;
//
// btnStart
//
this->btnStart->Location =
    System::Drawing::Point(704, 20);
this->btnStart->Name = L"btnStart";
this->btnStart->Size =
    System::Drawing::Size(81, 27);
this->btnStart->TabIndex = 0;
this->btnStart->Text = L"Start";
this->btnStart->UseVisualStyleBackColor = true;
this->btnStart->Click += gcnew
    System::EventHandler(this,
        &MyForm::btnStart_Click);
//
// canvas
//
this->canvas->Location =
    System::Drawing::Point(9, 9);
this->canvas->Name = L"canvas";
this->canvas->Size =
    System::Drawing::Size(800, 600);
this->canvas->TabIndex = 1;
this->canvas->TabStop = false;
//
// MyForm
//
this->AutoScaleDimensions =
    System::Drawing::SizeF(6, 13);
this->AutoScaleMode =
    System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize =
    System::Drawing::Size(817, 684);
this->Controls->Add(this->canvas);
this->Controls->Add(this->groupBox);
this->MaximizeBox = false;
```

```

        this->MinimizeBox = false;
        this->Name = L"MyForm";
        this->Text = L"The Barrier example";
        this->groupBox->ResumeLayout(false);
        (cli::safe_cast
            <System::ComponentModel::ISupportInitialize^>
            (this->canvas))->EndInit();
        this->ResumeLayout(false);

    }
#pragma endregion
    };
}

```

How it works...

We chose to draw an 800 x 600 (pixels) figure in our example. That's why we've set `iwidth` to 800 and `iHeight` to 600. Then, we'll allocate `Bitmap` and the color array. As we will draw a symmetrical left and right part of the canvas, we will iterate through half of the pixels' width. That's why we'll set the progress maximum to `iwidth / 2`. When we click on the **Start** button, we'll create a method delegate in order to pass it to the `Tasks::Parallel::Invoke` method. Before we call a parallel invoker, we need to instantiate the barrier object with two participants and a `postPhaseAction` method delegate. This post action method delegate is important for us because we want to draw the figure after the entire calculation is done. We could accomplish this with an event, or we could post a message to the window message queue, but this is the proper way when using the `Barrier` object. Our post action method is `EndTask`, which needs to draw the plot and set the label text. We must use delegates to invoke proper methods because a different thread will make this call and we must avoid cross-thread operations with UI elements.

There's more...

All these .NET framework constructs that we have explained are very good, but each of them can be more or less efficient for a given task. As you saw there are many built-in .NET constructs for user to use in its development. However, it is left to the programmer to explore and research further on which construct matches which task in its development.

7

Understanding Concurrent Code Design

In this chapter, we will cover the following topics:

- ▶ How to design parallel applications
- ▶ Understanding parallelism in code design
- ▶ Turning on to a parallel approach
- ▶ Improving the performance factors

Introduction

Even though synchronization and time are significant factors in developing parallel applications, adding concurrency to programs brings many additional concerns.

Parallel programming can be a sword with two edges: you can do great things with it, such as scaling a program to be executed over up-to-date modern multicore processors. Furthermore, parallelism presents great software capabilities where you can develop a responsive and rich user interface in GUI applications. However, you must still be aware of the fact that if parallelism is used incorrectly, it can cause serious performance issues.

Poorly designed parallel programs have worse performance compared to sequential programs because of the great overhead that occurs on the CPU while it reschedules the threads for execution, fetches the memory blocks, sets the signaled state on a synchronization object, maintains protection, and so on.

In this chapter, we will explain the parallel approach when creating applications, but before we do, first, we need to remind the user about sequential execution.

The sequential programming approach is very important to the life cycle of the application, because the algorithm that the program will follow must determine the exact steps that the application (thread) must execute. Even in parallel applications, it is very important that you choose the correct sequential approach, as much optimized as possible, and then move on to parallelization.

Here, we will try to find solutions on how to avoid such situations as well as see the examples, which demonstrate the correct usage of parallelism.

How to design parallel applications

In order to prevent problematic situations, we need to first identify the possible scenarios. We can divide problems related to concurrency into two major categories:

- ▶ **Correctness problems:** These can cause a program to produce incorrect results
- ▶ **Liveness problems:** These can cause a program to stop producing results completely

Correctness problems are very hard to discover because of the nature of the concurrency itself. Since each thread takes a different execution path, these situations can occur at any point of time without the programmer realizing that the program's output is incorrect, and something went wrong. In the worst case, we can run such a program many times with the correct results before a scenario occurs where, for example, different parameters can lead to a situation that will produce incorrect results.

One of these scenarios could be a **Data Race**—where the relation between the program state and the control flow must not be violated, or else your application will produce an incorrect result. Let's review the following situation:

```
int iValue = 5;
...
int iAValue = iValue;
...
int iBValue = iValue;
```

If more than one thread is accessing a shared object, `iValue`, we must add some kind of concurrency safety, such as exclusive access or data synchronization, in order to prevent possible unwanted scenarios and, at the same time, incorrect results as well.

In these situations, we must properly decide what we need to synchronize and what are the most preferable objects or techniques for this task.

Another scenario could be **Inconsistent Synchronization**, where it is not enough to secure the object using some kind of protection. Hence, it is imperative to use the same kind of protection for all the objects that will access the object. In the following code, pay attention to the `funcA` and `funcB` methods where a lock is declared with a different synchronization object:

```
class CLock
{
public:
    CLock( TCHAR* szMutexName );
    ~CLock( );
private:
    HANDLE hMutex;
};

CLock::CLock( TCHAR* szMutexName )
{
    hMutex = CreateMutex( NULL, FALSE, szMutexName );
    WaitForSingleObject( hMutex, INFINITE );
}

CLock::~CLock( )
{
    ReleaseMutex( hMutex );
    CloseHandle( hMutex );
}

...
int iValue = 5;
void funcA( void )
{
    CLock* lock = new CLock( TEXT( "_tmp_lock_01_" ) );
    ++iValue;
    delete lock;
}

void funcB( void )
{
    CLock* lock = new CLock( TEXT( "_tmp_lock_02_" ) );
    ++iValue;
    delete lock;
}
```

In a situation like the one shown here, when the method is called from different threads, we'd get incorrect results due to an inconsistent protection mechanism. There are so-called benign situations, where separate threads will only perform the read operation on a shared object, which is potentially dangerous.

Another scenario is called **Composite Actions**, where a constant tradeoff between single-threaded performance (due to more locks) and parallel (without locks) execution is made. In other words, proper decision on what are the statements or areas of code to protect is critical.

Imagine a situation where we use a simple linked list, and we remove the head node. When we mention the tradeoff, and when we want to remove the list's head node, we'd have two critical regions. First, when the query head node equals to zero (0), that is, `node == NULL`, and then on the assignment of a new node, `node = node->next`. If we protect a lot of statements, we'll lose parallelism. Understand the tradeoff: the fewer protected statements, the more concurrency we can achieve.

Assume that our list has only one node. We'll use the `CLock` class from the previous example for an exclusive access. The code is as follows:

```
template class<T>
class CList<T>
{
public:
    CList( ) : next( 0 ), value( 0 ) { }
    T Pop( void );
private:
    CList* next;
    T* value;
};

template class<T>
T* CList::Pop( void )
{
    CList* node = next;
    if (node == NULL)
    {
        return 0;
    }

    CLock* lock = new CLock(_T("_tmp_lock_"));
    next = node->next;
    delete lock;

    return node->value;
}
```

The preceding code can lead to issues. Can you see it? The situation is that we have protected the critical region around the `assignment` statement. We must take care about the execution flow all the way! What if we execute this portion of code in parallel? It could lead to the following situation: thread A enters the `Pop` method and executes the `node == NULL` statement and gets `FALSE`. It then continues, and the operating system suddenly selects another thread for execution. Now, thread B enters the `Pop` method and executes the `node == NULL` statement and gets `FALSE`. Again, the OS selects thread A for execution and it continues from where it left off. It will acquire a lock and remove the head node.

This is a bad situation because when the OS selects thread B the next time, it will continue from where it left off, and proceed to acquire the lock and access the non-existing object. This will most certainly lead to the app crashing! It can be illustrated as follows:

T	Thread A	Thread B
0	<code>CList* node = next</code>	
1	<code>next == NULL</code>	
2		<code>CList* node = next</code>
3		<code>next == NULL</code>
4	<code>CLock* lock = new CLock(_T("_tmp_lock_"));</code>	
5	<code>next = node->next;</code>	
6		<code>CLock* lock = new CLock(_T("_tmp_lock_"));</code>
7	<code>delete lock;</code>	
8		<code>next = node->next; // APP CRASH !!!!</code>

The first column (**T**) is the time required to execute the statement on the CPU. The next column consists of the **Thread A** statements followed by the **Thread B** column with its statements. Even if we've included protection, it is not designed correctly. A proper way of solving such a problem would be to simply protect both statements as follows:

```
template class<T>
class CList<T>
{
public:
    CList( ) : next( 0 ), value( 0 ) { }
    T Pop( void );
private:
    CList* next;
    T* value;
};

template class<T>
```



```
T* CList::Pop( void )
{
    CLock* lock = new CLock(_T("_tmp_lock_"));

    CList* node = next;
    if (node == NULL)
    {
        return 0;
    }

    next = node->next;
    delete lock;

    return node->value;
}
```

Now, even though the OS will change its execution from one thread to another, we won't have situations where the **Correctness problem** can occur, because we have properly secured the intensive operations and statements. However, protecting more statements means more sequential execution, so you must be aware of the performance improvement at all times.

The idea is to design an application execution flow in such a way that the critical points are grouped in small heavily-protected portions, and we must take care of as many usage scenarios (use cases) as possible, only to determine the correct parallel execution flow.

Let's talk a bit about **Liveness problems**. They don't usually terminate the program, and as such, are very difficult to detect and diagnose. In most situations, these problems will lead to the application ceasing to respond. In some cases, the application will stop temporarily. Deadlock is one of these scenarios—so let's review the example where we need to transfer money from two bank accounts. We'll implement the `CBankAccount` class with the ability to transfer money with lock protection (in case that both accounts are used in parallel). An obvious approach would look like this:

```
class CBankAccount
{
public:
    CBankAccount(){ _tcscopy( szLock, LockName( ) ); }
    static void Transfer(CBankAccount* a, CBankAccount* b, double
dAmount);
```

```

    static TCHAR* LockName(void);
private:
    unsigned uID;
    double dBalance;
    TCHAR szLock[16];
};

TCHAR* CBankAccount::LockName(void)
{
    static int iCount = 0;
    static TCHAR szBuffer[16];
    wsprintf( szBuffer, _T("_lock_%d_"), ++iCount );
    return szBuffer;
}

void CBankAccount::Transfer(CBankAccount* a, CBankAccount* b, double
dAmount )
{
    CLock* outerLock = new CLock(a->szLock);
    if (dAmount < a->dBalance)
    {
        CLock* innerLock = new CLock(b->szLock);
        a->dBalance -= dAmount;
        b->dBalance += dAmount;
        delete innerLock;
    }
    delete outerLock;
}

```

The preceding code looks correct, but it isn't. It's incorrect! Assume the situation where thread A wants to transfer some amount from account X to account Y, while at the same time, thread B wants to transfer some amount from account Y to account X. When thread A enters the Transfer method, it will acquire the lock that belongs to account X. Now, assume that the name of the mutex that is used for account X is *lock_1*. Then, assume that the next scheduled thread is B and it acquires the lock that belongs to account Y, and its mutex name is *lock_2*. Next, thread A continues and tries to acquire the lock that belongs to account Y. It blocks—because *lock_2* is held by thread B and belongs to account Y. Now, thread B continues and tries to acquire *lock_1*, which is held by the previously suspended thread A.

This is illustrated by the following figure:

T	Thread A (transfer from X to Y)	Thread B (transfer from Y to X)
0	CLock* outerLock = new CLock(a->LockName()); <i>// thread A acquires a lock_1 on account X</i>	
1		CLock* outerLock = new CLock(a->LockName()); <i>// thread B acquires a lock_2 on account Y</i>
2	if (dAmount < a->dBalance)	
3		if (dAmount < a->dBalance)
4	CLock* outerLock = new CLock(b->LockName()); <i>// thread A tries to acquire a lock_2 on // account Y and it gets suspended</i>	
5	suspended	CLock* outerLock = new CLock(b->LockName()); <i>// thread B tries to acquire a lock_1 on // account X and it gets suspended</i>
6	<i>// THREAD WAIT FOREVER !!!!</i>	suspended
7		<i>// THREAD WAIT FOREVER !!!!</i>

Again, the first column (**T**) is the time required to execute the statements. The following column consists of the **Thread A** code statements. Next is the **Thread B** code statements. The illustrated situation may occur very rarely, but when we are going to design parallel code, we must be aware of such situations at all times.

Understanding parallelism in code design

We have already said that the most difficult work for a programmer is not to create the code—it is to properly design the application. When you follow these steps, after creating a good and stable design, you only have one step left: to write the code following the created design step by step. Our following example will try to teach you in a step-by-step manner how to approach the application creation from scratch, and at the same time think ahead on how to add parallelism later.

We will create a program that will calculate Schwefel's function. Usually, for the calculation of Schwefel's function and similar functions where probability and statistics make a significant difference in the results the genetic algorithm is used. This is because such an algorithm generates a huge population of genomes with its features, and compares their score (result). For simplicity of the example, we will create our own class, `CSchwefel`, which will satisfy our needs.

The Schwefel's function is as follows:

$$f(x) = \sum_{i=1}^n \left(\sum_{j=1}^i x_j \right)^2, -65.536 \leq x_i \leq 65.536$$

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new C++ Console application named `ConcurrentDesign`.
2. Open **Solution Explorer** and open the `stdafx.h` file. Copy and paste the following code to it:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
#include <Windows.h>
#include <cfloat>
#include <time.h>
#include <omp.h>

#include "Schwefel.h"
```

3. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `Schwefel.h`. Open `Schwefel.h` and paste the following code to it:

```
#ifndef __SCHWEFEL__
#define __SCHWEFEL__

#include "stdafx.h"

SYSTEMTIME operator - (const SYSTEMTIME& stFirst, const
SYSTEMTIME& stSecond);

class CSchwefel
```

```
{
public:
    CSchwefel(void) : pdAllele(0) {}
    CSchwefel(unsigned uSize) { pdAllele = new double[this->uSize =
uSize]; Initialize(); }
    double* Allele(void) const { return pdAllele; }
    double Allele(unsigned uIndex) const { return pdAllele[uIndex];
}
    double Allele(double dValue, unsigned uIndex) { return
pdAllele[uIndex] = dValue; }
    unsigned Size(void) const { return uSize; }
    unsigned Size(unsigned uSize) { return this->uSize = uSize; }
    virtual ~CSchwefel(void) { if (pdAllele != 0) delete[] pdAllele;
}

    double& Sum(void) { return dSum; }
    double& Min(void) { return dMin; }
    double& Score(void) { return dScore; }
    double& Best(void) { return dBest; }
    virtual double Evaluate(void* lpParameters = 0);
    void Initialize();
    void Initialize(unsigned uSize);
    SYSTEMTIME& StartTime(void) { return stStart; }
    SYSTEMTIME& EndTime(void) { return stEnd; }
    TCHAR* Elapsed(void);
    TCHAR* Elapsed(TCHAR* szElapsed);
    static const unsigned ThreadNumber = 4;
protected:
    CSchwefel(const CSchwefel&) { }
    CSchwefel& operator = (const CSchwefel&) { return *this; }
    virtual TCHAR* ObjectName(void) const { return _T("Schwefel"); }
private:
    double* pdAllele;
    unsigned uSize;
    double dSum;
    double dMin;
    double dScore;
    double dBest;
    SYSTEMTIME stStart;
    SYSTEMTIME stEnd;

    double Random(double dMin, double dMax);
    static unsigned Seed(void) { static unsigned uSeed = rand() %
time(NULL); return uSeed++; }
};
```

4. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named `Schwefel`. Open `Schwefel.cpp`. Copy and paste the following code to it:

```
#include "stdafx.h"

double CSchwefel::Evaluate(void* lpParameters)
{
    GetSystemTime(&stStart);

    for (unsigned i = 0; i < uSize; i++)
    {
        dSum += pdAllele[i];

        if (dMin > pdAllele[i])
        {
            dMin = pdAllele[i];
        }

        double dInnerSum = 0;

        for (unsigned j = 0; j < i; j++)
        {
            dInnerSum += pdAllele[j];
        }

        double dTmp = dInnerSum * dInnerSum;
        dScore += dTmp;

        if (i != 0 && dBest > dTmp)
        {
            dBest = dTmp;
        }
    }

    GetSystemTime(&stEnd);

    return dScore;
}

void CSchwefel::Initialize(void)
{
    srand(Seed());

    dSum = 0;
    dMin = DBL_MAX;
    dScore = 0;
}
```

```
    dBest = DBL_MAX;
    stStart = { 0 };
    stEnd = { 0 };

    for (unsigned uIndex = 0; uIndex < uSize; uIndex++)
    {
        pdAllele[uIndex] = Random(-65.536, 65.536);

        dSum += pdAllele[uIndex];

        if (dMin > pdAllele[uIndex])
        {
            dMin = pdAllele[uIndex];
        }
    }
}

void CSchwefel::Initialize(unsigned uSize)
{
    if (pdAllele == NULL)
    {
        pdAllele = new double[this->uSize = uSize];
    }

    srand(Seed());

    dSum = 0;
    dMin = DBL_MAX;
    dScore = 0;
    dBest = DBL_MAX;
    stStart = { 0 };
    stEnd = { 0 };

    for (unsigned uIndex = 0; uIndex < uSize; uIndex++)
    {
        pdAllele[uIndex] = Random(-65.536, 65.536);

        dSum += pdAllele[uIndex];

        if (dMin > pdAllele[uIndex])
        {
            dMin = pdAllele[uIndex];
        }
    }
}
```

321


```
    ULARGE_INTEGER uLargeInteger;

    __int64 i64Right;
    __int64 i64Left;
    __int64 i64Result;

    SystemTimeToFileTime(&stFirst, &ftTime);
    uLargeInteger.LowPart = ftTime.dwLowDateTime;
    uLargeInteger.HighPart = ftTime.dwHighDateTime;
    i64Right = uLargeInteger.QuadPart;

    SystemTimeToFileTime(&stSecond, &ftTime);
    uLargeInteger.LowPart = ftTime.dwLowDateTime;
    uLargeInteger.HighPart = ftTime.dwHighDateTime;
    i64Left = uLargeInteger.QuadPart;

    i64Result = i64Right - i64Left;

    uLargeInteger.QuadPart = i64Result;
    ftTime.dwLowDateTime = uLargeInteger.LowPart;
    ftTime.dwHighDateTime = uLargeInteger.HighPart;
    FileTimeToSystemTime(&ftTime, &stResult);

    return stResult;
}
```

5. Open **Solution Explorer** and open `ConcurrentDesign.cpp`. Copy and paste the following code to it:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    CSchwefel Schwefel(100000);
    Schwefel.Evaluate();

    _tprintf_s(Schwefel.Elapsed());

#ifdef _DEBUG
    return system("pause");
#endif

    return 0;
}
```

How it works...

First, we need to define the `CSchwefel` class. We are going to use it to store an array of double values ranging from -65536 to 65536, which is the range of the allowed values for this function. We will name this array, `allele`, similar to the genome structure. Also, we'll need the `allele` size, sum of elements, minimal value, score, the best value, and the start and end time.

We'll have two default constructors with `allele` size. If we use the default constructor, we'll simply create a dummy object, and we must call the `Initialize` method before any calculation. However, we can use another constructor with built-in initialization, but we must supply the `allele` size. For our example, the second one is the obvious choice. We need to implement the set and get methods for our attributes. `double* Allele(void)` simply returns the pointer to the `allele` array; `double Allele(unsigned uIndex)` returns the value at the array index specified with `uIndex`; `double Allele(double dValue, unsigned uIndex)` sets the value at the `uIndex` array with `dValue` and returns the newly-added value. The `unsigned Size(void)` attribute simply returns the `allele` size; `unsigned Size(unsigned uSize)` sets a new size and returns the newly-added size.

The `Sum`, `Min`, `Score`, `Best`, `StartTime`, and `EndTime` methods return a reference to the class attributes `dSum`, `dMin`, `dScore`, `dBest`, `stStart`, and `stEnd` respectively, so that we can use these methods for the get and set methods at the same time. The `Initialize(void)` and `Initialize(unsigned uSize)` methods fill the `allele` array with random values from -65536 to 65536, and at the same time, are calculating the sum of the `allele` array and min value from the same array.

The `Evaluate(void*)` method as well as destructor are declared as virtual methods because the derived classes can change the evaluation sequence or the approach that we will follow for our concurrent approach later on. The `Elapsed(void)` and `Elapsed(TCHAR*)` methods simply write the output to the console. We'll also add the constant value, `ThreadNumber`, which we'll use to set the maximum allowed threads. You can change this value to any preferred one.

In the `protected` scope, we've placed a copy constructor and overloaded operator `equal` (`=`) in order to disallow the programmer from using them. We want to disallow the programmer from using them because the default copy operation is performed in a manner such that all the class attributes' values are copied one by one. This is OK for any integral type but not for pointers. It is not enough to copy a pointer value—this would mean that the pointers from both the objects point to the same array or any other memory that is referenced by this pointer. You must implement both the copy constructor and the `equal` operator to allocate new memory for each instance of a class object, and then copy the values that the old pointer refers to, one by one. The `TCHAR* ObjectName(void)` method simply returns the class name when formatting the output string in the `Elapsed` method. In each of our derived classes, we will overload the `ObjectName` method to return the correct overloaded class name.

In the `private` scope, we've placed the `Random(double dMin, double dMax)` and `Seed(void)` methods that return a random double number from `dMin` to `dMax` and the seed for a random generator respectively.

As we previously mentioned, the `Evaluate` methods were declared as virtual, and for the `CSchwefel` class, implementation is straightforward. First, we call the `GetSystemTime` API, with `stStart` as the parameter in order to store the time before we start calculating; then, inside the outer loop, we'll calculate the sum and min; inside the inner loop, we'll calculate the inner sum and multiply it with itself. According to Schwefel's formula, we add it to the score and calculate the best value. When we finish the outer loop, we'll call `GetSystemTime` with the `stEnd` parameter in order to obtain the system time once again. When we subtract these two times, we'll get the elapsed time for calculation. One more thing about the `Evaluate` method: for the sake of future implementation (for example, class derivation), we've planned one argument `void*`, but you can pass as many arguments as you like in such a manner—just like in any of our examples, which use some routine for the `CreateThread` API.

Since the `SYSTEMTIME` structure does not implement the method for the `SYSTEMTIME` subtraction, we'll implement our own: `SYSTEMTIME operator - (const SYSTEMTIME& stFirst, const SYSTEMTIME& stSecond);`. Inside this method, we'll convert the system time to the file time using the `SystemTimeToFileTime` API so that we get the integer value from the human-readable time for both the arguments. Then, we'll subtract these two integers and we'll convert the result to a human readable time using the `FileTimeToSystemTime` API.

In our application's entry point, we'll simply instantiate the `CSchwefel` object with a size of 100,000 alleles and call the `Evaluate` and `Elapsed` methods.

There's more...

When we are talking about techniques, there are always variations. Usually, whatever you implement, there almost always exists another approach or technique to do that in some other way. Our intention here is to teach you how to properly think and solve the issues related to concurrency. However, in order to achieve the best results, practice is always the best teacher.

Turning on to a parallel approach

Now, we want to add concurrency to the same example. Note that we've designed the `CSchwefel` class for such a task, because beside the `Evaluate` method, everything remains the same! Again, the power of object-oriented programming saves us a lot of coding and time.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create the `CSchwefelMT` class and explain its structure. Perform the following steps:

1. Open **Solution Explorer** and open the file `stdafx.h`. Copy and paste the following code to it:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
#include <Windows.h>
#include <cfloat>
#include <time.h>
#include <omp.h>

#include "Schwefel.h"
#include "SchwefelMT.h"
```

2. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `SchwefelMT`. Open `SchwefelMT.h`. Copy and paste the following code to it:

```
#ifndef __SCHWEFELMT__
#define __SCHWEFELMT__

#include "stdafx.h"

class CSchwefelMT : public CSchwefel
{
public:
    CSchwefelMT() : CSchwefel() { }
    CSchwefelMT(unsigned uSize) : CSchwefel(uSize) { }
    virtual ~CSchwefelMT(void) { }
    virtual double Evaluate(void* lpParameters = 0);
protected:
    CSchwefelMT(const CSchwefel&) { }
    CSchwefelMT& operator = (const CSchwefel&) { return *this; }
```

```
    virtual TCHAR* ObjectName(void) const { return _T("SchwefelMT");  
    }  
private:  
    static DWORD WINAPI StartAddress(LPVOID lpParameter);  
    struct ThreadParams  
    {  
        CSchwefelMT* this_ptr;  
        unsigned uThreadIndex;  
    };  
};  
  
#endif
```

3. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named SchwefelMT. Open SchwefelMT.cpp. Copy and paste the following code to it:

```
#include "stdafx.h"  
  
DWORD WINAPI CSchwefelMT::StartAddress(LPVOID lpParameter)  
{  
    ThreadParams* params = (ThreadParams*)lpParameter;  
  
    unsigned uChunk = ((unsigned)params->this_ptr->Size()) /  
ThreadNumber;  
    unsigned uMax = (params->uThreadIndex + 1) * uChunk;  
  
    for (unsigned i = params->uThreadIndex * uChunk; i < uMax; i++)  
    {  
        params->this_ptr->Sum() += params->this_ptr->Allele(i);  
  
        if (params->this_ptr->Min() > params->this_ptr->Allele(i))  
        {  
            params->this_ptr->Min() = params->this_ptr->Allele(i);  
        }  
  
        double dInnerSum = 0;  
  
        for (unsigned j = 0; j < i; j++)  
        {  
            dInnerSum += params->this_ptr->Allele(j);  
        }  
  
        double dTmp = dInnerSum * dInnerSum;
```

```

        params->this_ptr->Score() += dTmp;

        if (i != 0 && params->this_ptr->Best() > dTmp)
        {
            params->this_ptr->Best() = dTmp;
        }
    }

    return 0L;
}

double CSchwefelMT::Evaluate(void* lpParameters)
{
    HANDLE hThreads[ThreadNumber];
    ThreadParams params[ThreadNumber];

    GetSystemTime(&StartTime());

    for (unsigned uIndex = 0; uIndex < ThreadNumber; uIndex++)
    {
        params[uIndex] = { this, uIndex };
        hThreads[uIndex] = CreateThread(NULL, 0, (LPTHREAD_START_
ROUTINE) StartAddress, &params[uIndex], 0, NULL);
    }

    WaitForMultipleObjects(ThreadNumber, hThreads, TRUE, INFINITE);

    GetSystemTime(&EndTime());

    for (unsigned uIndex = 0; uIndex < ThreadNumber; uIndex++)
    {
        CloseHandle(hThreads[uIndex]);
    }

    return Score();
}

```

4. Open **Solution Explorer** and open the `ConcurrentDesign.cpp` file. Copy and paste the following code to it:

```

#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])

```

```
{
    CSchwefel Schwefel(100000);
    Schwefel.Evaluate();

    _tprintf_s(Schwefel.Elapsed());

    CSchwefelMT SchwefelMT(100000);
    SchwefelMT.Evaluate();

    _tprintf_s(SchwefelMT.Elapsed());

#ifdef _DEBUG
    return system("pause");
#endif

    return 0;
}
```

How it works...

As `CSchwefelMT` is derived from `CSchwefel`, we only need to implement the constructors and destructors along with the virtual `Evaluate` method. We must disallow the programmer from using the copy constructor and the equal operator for the same reasons as in the `CSchwefel` class. In the private scope, we've added a `StartAddress` routine for the threads and the `ThreadParams` structure for passing the arguments to the threads.

What about the `Evaluate` method now? It's simple. We'll instantiate `ThreadNumber` of threads and `ThreadNumber` of parameters for them; then we'll call `GetSystemTime` and run the declared threads; we'll create a barrier, or in other words, we'll wait for their return using the `WaitForMultipleObjects` routine, and then call `GetSystemTime` once again and close all the thread handles.

What about `StartAddress`? Is it straightforward? Let's see. If we simply follow the logic from the previous single-threaded example, our `Evaluate` method will have many Correctness problems! Let's explain and illustrate some of them. Assume that there are two threads (it is the same for more than two); pay attention to this line:

```
params->this_ptr->Sum() += params->this_ptr->Allele(i);
```

We are accessing the `dSum` attribute from both the threads without any protection! Let's illustrate what can happen:

			Memory (RAM)	
T	Thread A on CPU_0	Thread B on CPU_1	address	value
		
			1008	5.036
0	MOV RAX, [4004]	MOV RAX, [4004]
			2016	-3.047
1	ADD RAX, [1008]	ADD RAX, [2016]
			4004	42.56

Thread A wants to add a value of `Allele(i)` to `dSum`. Both `Allele(i)` and `dSum` are memory locations that have a certain address. Let's assume that the address of `dSum` is 4004 and its current value is 42.56; assume that the address of `Allele(i)` for thread A is 1008; thread B also wants to add its own `Allele(i)` value on the current `dSum` value; the `dSum` address is the same, that is 4004, while the address of thread B `Allele(i)` is different than that of thread A (assume it is 2016); thread A is executing (for example) on CPU 0 while thread B is executing on CPU 1.

The statement at time 0 for thread A is `RAX = 42.56` while for thread B it is `RAX = 42.56` as you can see this. The statement at time 1 for thread A is `RAX = 42.56 + 5.036`, which is 47.596, while for thread B it is `RAX = 42.56 + (-3.047)`, which is 39.513. As you can see, this is totally wrong! The value of `dSum` won't be 44.549 as it should be after we add both the values, as that was the idea. Again, the correct application design is crucial with concurrency in action!

We must change the `Evaluate` method in a manner such that it follows the logic needed for the calculation of Schwefel's function, but at the same time, we must respect the multithreaded execution, or else we'll most certainly get correctness or liveness problems! The correct implementation is as follows:

```
DWORD WINAPI CSchwefelMT::StartAddress(LPVOID lpParameter)
{
    ThreadParams* params = (ThreadParams*)lpParameter;

    unsigned uChunk = ((unsigned)params->this_ptr->Size()) / ThreadNumber;
    unsigned uMax = (params->uThreadIndex + 1) * uChunk;

    double dSum = 0;
    double dMin = DBL_MAX;
```



```

double dScore = 0;
double dBest = DBL_MAX;

HANDLE hMutex = CreateMutex(NULL, FALSE, _T("__tmp_mutex__"));
WaitForSingleObject(hMutex, INFINITE);

dSum = params->this_ptr->Sum();
dMin = params->this_ptr->Min();
dScore = params->this_ptr->Score();
dBest = params->this_ptr->Best();

ReleaseMutex(hMutex);
CloseHandle(hMutex);

for (unsigned i = params->uThreadIndex * uChunk; i < uMax; i++)
{
    dSum += params->this_ptr->Allele(i);

    if (dMin > params->this_ptr->Allele(i))
    {
        dMin = params->this_ptr->Allele(i);
    }

    double dInnerSum = 0;
    for (unsigned j = 0; j < i; j++)
    {
        dInnerSum += params->this_ptr->Allele(j);
    }
    double dTmp = dInnerSum * dInnerSum;
    dScore += dTmp;

    if (i != 0 && dBest > dTmp)
    {
        dBest = dTmp;
    }
}

hMutex = CreateMutex(NULL, FALSE, _T("__tmp_mutex__"));
WaitForSingleObject(hMutex, INFINITE);

params->this_ptr->Sum() = dSum;
if (dMin < params->this_ptr->Min())

```

```

{
    params->this_ptr->Min() = dMin;
}
params->this_ptr->Score() = dScore;
if (dBest < params->this_ptr->Best())
{
    params->this_ptr->Best() = dBest;
}

ReleaseMutex(hMutex);
CloseHandle(hMutex);

return 0L;
}

```

Now, as you can see, we have designed statements in such a manner that all the accesses to the shared memory locations are protected. The references to the shared objects are removed inside the loop because using shared objects and protecting access for them inside the loop won't do much good! Code like that would only slow the application down because we would lose concurrent calculation if the majority of the statements inside the loop were protected! That's why we add the thread-local storage so that the calculations are made in parallel. In the beginning of the method, we have to protect the statements where we read the shared objects, then we'll use the thread-local variables, and at the end, we'll again protect the shared objects and we'll then perform the write operations. Safe and clean!

There's more...

When we think about how to improve the application's performance, we can consider using parallelism to achieve the same, but we must always take care of two things: the performance speed-up and the efficiency. Both the things matter when we compare the performances of the sequential algorithm to the parallel one.

In practice, there are always situations where your application should be fine-tuned for a specific demand of an important client. For such demands, a carefully-chosen decision about whether or not to use parallelism is very important. If you decide that performance improvement will occur, then you should create a strong sequential algorithm and then wisely transform it into parallel. All the time, you must think about side-effects such as shared objects or access violation.

These situations are like a beam balance (see-saw). If you want to get the leverage on one side, for example, you want to improve the application performance, then you will lose the leverage on the other one. You could use the parallel approach with more work. However, if you do it right, your application will benefit. This exact type of synchronization must be included from the beginning.

Improving the performance factors

We said that it is important to build a strong sequential algorithm before we go parallel. We also said that there are parameters such as performance speed-up and efficiency. How can you really measure these variables so that you can determine whether you should go parallel or not? If we do go parallel, will the application benefit from it and how significantly?

We will try to answer these questions in this topic, and we'll give an example of a built-in Visual Studio compiler feature, which relies on **OpenMP**, the API developed by Intel in cooperation with **Livermore National Laboratories** from the United States. OpenMP stands for **open multiprocessor** compilation, which is the Microsoft implementation of the OpenMP API. It is an excellent built-in feature that supports the native parallel execution by the compiler itself.

First, let's discuss speed up. How can we properly decide whether to use parallelism or not? Well, one way would be by measuring and comparing the execution time. Another way would be to prove a certain mathematical formula that is used to represent the application execution in parallel. However, the second approach, if chosen, must also be tested—so again, time measurement is needed.

In order to measure these values, we must bring some variables to the table. If we assume that **T(S)** is the execution time needed for the sequential algorithm and **T(P)** is the execution time needed for the parallel algorithm, then we could get the following equation:

$$I = \frac{T(S)}{T(P)}$$

In the preceding equation, *I* stands for improvement. However, we can distinguish the following variations of improvement in the result:

- ▶ $I < 1$ indicates a slowdown, which means that the parallel execution will only decrease performances
- ▶ $I < P$ indicates a sublinear improvement
- ▶ $I \sim P$ indicates a linear improvement
- ▶ $I > P$ indicates a superlinear improvement

Slowdown usually indicates that the sequential approach will have better performances, but that may not always be the case. It could also mean that the parallel code wasn't designed well or there were too many unnecessary protection or synchronization points. It could also point to some implementation mistake. In theory, such an algorithm itself could be transformed into a parallel one, but there is another question then: can the hardware support such an execution? What if you test it on four CPU cores and you get respectable results and then you try to execute it on two cores? Depending on the algorithm itself—it could happen that the application's execution time is much higher than, for example, using the sequential algorithm executing on a single CPU core. All these factors should be checked before you decide on what approach to choose and how your application should benefit from it. There are two significant factors that you should always think about:

- ▶ How can the application do more work in less time?
- ▶ How can the application use more resources without affecting the rest of the system or other applications?

In our following example, we will again calculate Schwefel's sum. We'll derive the `CSchwefel` base class and create a new `CSchwefelP` class that will use the OpenMP features only to demonstrate another technique that you can use for improved application performance, all the time taking care about the potential improvement possibility.

In order to use the OpenMP directives, you must properly set the Visual Studio project's property pages. The detailed settings that are required to run the following code can be found in the *Appendix*.

Getting ready

Make sure that Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Open the solution `ConcurrentDesign`.
2. Open **Solution Explorer** and right-click on **Header file**. Add a new header file named `CSchwefelP`. Open `CSchwefelP.h`.
3. Add the following code to it:

```
#ifndef __SCHWEFELP__
#define __SCHWEFELP__

#include "stdafx.h"

class CSchwefelP : public CSchwefel
```

```
{
public:
    CSchwefelP() : CSchwefel() { }
    CSchwefelP(unsigned uSize) : CSchwefel(uSize) { }
    virtual ~CSchwefelP(void) { }
    virtual double Evaluate(void* lpParameters = 0);
protected:
    CSchwefelP(const CSchwefel&) { }
    CSchwefelP& operator = (const CSchwefel&) { return *this; }
    virtual TCHAR* ObjectName(void) const { return _T("SchwefelP"); }
}
private:
    //
};

#endif
```

4. Open **Solution Explorer** and right-click on **Source file**. Add a new source file named CSchwefelP. Open CSchwefelP.cpp.
5. Add the following code to it:

```
#include "stdafx.h"

double CSchwefelP::Evaluate(void* lpParameters)
{
    GetSystemTime(&StartTime());

    double sum = 0;
    double score = 0;
    double min = DBL_MAX;
    double best = DBL_MAX;
    int iSize = (int)Size();

#pragma omp parallel firstprivate(min) firstprivate(best)
    {
#pragma omp for reduction(+: sum) reduction(+: score) nowait
        for (int i = 0; i < iSize; i++)
        {
            sum += Allele(i);

            if (min > Allele(i))
            {
                min = Allele(i);
            }
        }
    }
}
```

```

    }

    double dInnerSum = 0;

    for (int j = 0; j < i; j++)
    {
        dInnerSum += Allele(j);
    }

    double dTmp = dInnerSum * dInnerSum;
    score += dTmp;

    if (i != 0 && best > dTmp)
    {
        best = dTmp;
    }
}

#pragma omp critical
{
    if (Best() > best)
    {
        Best() = best;
    }
    if (Min() > min)
    {
        Min() = min;
    }
}

Sum() = sum;
Score() = score;

GetSystemTime(&EndTime());

return Score();
}

```

6. Open **Solution Explorer** and double-click on `stdafx.h`.

7. Add the following code to it:

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
#include <Windows.h>
#include <cfloat>
#include <time.h>
#include <omp.h>

#include "Schwefel.h"
#include "SchwefelMT.h"
#include "SchwefelP.h"
```

8. Open **Solution Explorer** and double-click on ConcurrentDesign.cpp.

9. Add the following code to ConcurrentDesign.cpp:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    CSchwefel Schwefel(100000);
    Schwefel.Evaluate();

    _tprintf_s(Schwefel.Elapsed());

    CSchwefelMT SchwefelMT(100000);
    SchwefelMT.Evaluate();

    _tprintf_s(SchwefelMT.Elapsed());

    CSchwefelP SchwefelP(100000);
    SchwefelP.Evaluate();

    _tprintf_s(SchwefelP.Elapsed());

#ifdef _DEBUG
    return system("pause");
#endif

    return 0;
}
```

How it works...

First, let's explain the `CSchwefelP` derived class. We must implement both, constructor and destructor, and the virtual method, `Evaluate`. The constructor as well as destructor implementation is straightforward. Now, let's see the `Evaluate` method's implementation. First, we'll obtain the current system time. Then, similar to the `CSchwefelMT` class, we'll instantiate the thread-local storage objects that we'll use for the private as well as shared objects. Note that we must strictly follow the concurrent logic. All the class attributes that will be accessed in parallel must be protected all the time! So `dSum`, `dScore`, `dMin`, and `dBest` must be protected from inconsistent access at all times. We'll also add the local variable, `iSize`, in order to obtain the allele size only once, because placing the `i < Size()` statement in the parallel region would be ineffective and would bring inconsistent access to the shared object.

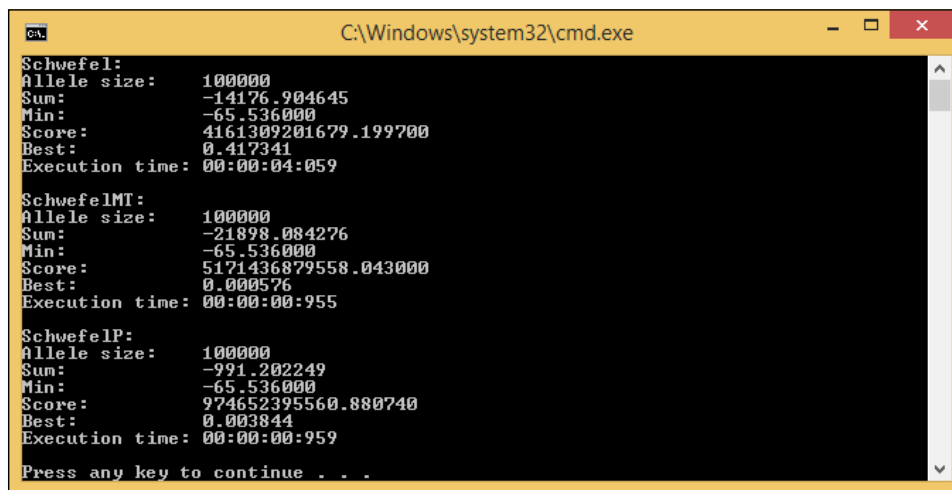
Now, we'll use the OpenMP directive, `#pragma omp parallel`, with additional clauses `firstprivate(min)` and `firstprivate(best)`. The `omp parallel` directive instructs the compiler to define a parallel construct (region), where the code will be executed by multiple threads in parallel. The `firstprivate` clause specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable because it exists before the parallel construct (from MSDN). Our intention is to make the `min` and `best` values private to the thread. This is because all the threads are calculating the minimum and best scores on their own portion of the allele array. In that way, we will ensure that there is no unprotected access to any shared class attribute, in our case `dMin` or `dBest`.

The next statement is `#pragma omp for reduction(+: sum) reduction(+: score) nowait`, followed by the `for` loop. The `#pragma omp [parallel] for` directive causes the work done in a `for` loop inside a parallel region to be divided among the threads (from MSDN). The `reduction(operation, variable)` clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region. This means that under the hood, the compiler will use the *variable* specified in the `reduction` clause as it is private for each thread on the *operation* specified in the `reduction` clause, but at the end of the parallel region, it will safely (with mutual exclusion) perform the necessary reduction—in our case, `sum`. So, whenever the parallel loop is executing, each thread will calculate the sum independently. At the end of the `for` loop, but before exiting from the loop, all the sum values that are private to the threads will be added into one. In this manner, parallelism is properly enabled because all the threads' private work was distinguished from a shared sum. Each thread will calculate its own portion of the allele array sum without interfering with another thread. At the end of the loop, all the threads' sum results will be added into one value. All we have to do now is to set the class attributes. One more thing about the `nowait` clause: it specifies to the compiler that if the thread has finished the work in the loop, it doesn't have to wait until other threads finish the same loop. You can think of `nowait` as a specification that has no barrier that the threads will hit and wait until all the others reach that point.

After the parallel loop has finished, we'll add another parallel construct: `#pragma omp critical`. It specifies to the compiler that the statements under the construct must be executed with exclusive access to any object inside. Each thread has its own minimum and best score. We must perform the comparison and assignment on each thread sequentially due to the synchronized access. After the end of the root parallel construct, we'll make the necessary assignments to `dSum` and `dScore` and obtain the system time once again to be able to calculate the elapsed time.

There's more...

We mentioned earlier about measuring performances. We'll use the previous example output as shown in the following screenshot:



```
C:\Windows\system32\cmd.exe
Schwefel:
Allele size: 100000
Sum: -14176.904645
Min: -65.536000
Score: 4161309201679.199700
Best: 0.417341
Execution time: 00:00:04:059

SchwefelMT:
Allele size: 100000
Sum: -21898.084276
Min: -65.536000
Score: 5171436879558.043000
Best: 0.000576
Execution time: 00:00:00:955

SchwefelP:
Allele size: 100000
Sum: -991.202249
Min: -65.536000
Score: 974652395560.880740
Best: 0.003844
Execution time: 00:00:00:959

Press any key to continue . . .
```

We're using the Intel i7-3820 processor. It has 8 logical cores with **Hyper-Threading** available, and it is capable of running 8 parallel hardware-supported threads.

Hyper-Threading Technology uses the processor resources more efficiently, enabling multiple threads to run on each core. As a performance feature, Intel HT Technology also increases the processor throughput, improving the overall performance on threaded software. To learn more about Intel HT, visit the following link:

<http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html>

In order to compare the `CSchwefelMT` object's performance with `CSchwefelP`, we've set the attribute `CSchwefelMT::ThreadNumber` to 8. Using these settings, we came up with the results from above.

We will now use the improved equation:

$$I = \frac{T(S)}{T(P)}$$

The first output belongs to the `CSchwefel` object with the sequential algorithm and a single thread. Its execution time is 4 seconds and 57 milliseconds. So $T(S)$ is 4057 (in milliseconds). The second output belongs to the `CSchwefelMT` object with 8 threads and its execution time is 955 milliseconds. With the second approach, we accomplished the following improvement:

$$I = \frac{4057}{955} = 4.248$$

The result $I < P$, $I = 4.248$ and $P = 8$ indicates a sublinear improvement, which is the most common improvement. A linear improvement is harder to achieve because you'll almost certainly have some kind of synchronization in your program and very often thread communication. All this affects the perfect linear improvement, making it harder to achieve. As opposed to these two—a superlinear improvement is almost impossible; if we have X number of processors, is it possible to execute something that is X times faster? Well, in certain situations where the shared objects can speed up the application by storing intermediate results—possibly. A situation like this could be storing the already calculated half of the tree from the **Fibonacci** sequence.

The third output belongs to the `CSchwefelP` object with 8 threads because we haven't explicitly set the `omp parallel` construct with `num_threads(int)`. In a situation like this, the compiler uses all the available threads when creating a dynamic thread pool. Its execution time is 959 milliseconds. With the third approach, we accomplished the following improvement:

$$I = \frac{4057}{959} = 4.231$$

$I < P$ also indicates a sublinear improvement. There is a slight smaller improvement in the performance than for the `CSchweffelMT` object, which can be related to many things such as the allele size, OpenMP API implementation, scheduler priority, and many more. Our intention was to show you that a proper approach from design to code must be planned and performed in steps. In the beginning, it is necessary to carefully determine whether or not parallelism would bring benefits and improvements using the described techniques. Practice is always the best indicator. The more successful work you've done, the better solutions you will produce.

See also

- ▶ You can find more on OpenMP at <http://openmp.org/wp/> or on <http://msdn.microsoft.com/en-us/library/tt15eb9t.aspx>

8

Advanced Thread Management

In this chapter, we will cover the following topics:

- ▶ Using thread pools
- ▶ Customizing the thread pool dispatcher
- ▶ Using remote threading

Introduction

When we want to create a complex application, we have to include some kind of sophisticated mechanisms in order to support the complex design. In my long work experience, I have seen many well-designed projects that have suffered due to an improper choice of software solutions. These choices can be a wrong class design or incorrect routine usage. Furthermore, if parallelism is added, you then need a managing mechanism that will dispatch and control concurrent threads.

A majority of modern applications as well as the Windows operating system have such mechanisms. One mechanism is the **Windows Dispatcher object** that manages processes threads, schedules their execution on the CPU, and takes care of synchronization. Another mechanism is the **Windows Virtual Memory manager** that manages memory operations and maintains inaccessible memory—reserved for operating system or zero-pointer assignments—and the usable user space memory for applications.

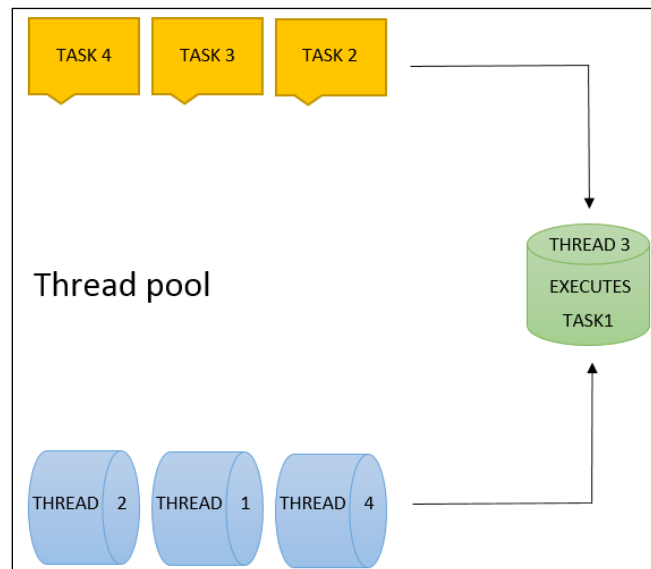
One thing that is common for such mechanisms is some abstraction of the root object that is responsible for managing its tasks. This is very important in order to maintain correct synchronization and exclusive access. Such an object—manager—should create and manage tasks, ensuring that the necessary resources are available. It must also control task execution in order to ensure that no two or more tasks will perform invalid or unsynchronized operations. It also should destroy tasks and release used resources correctly for future reuse.

One of these abstractions is a pool of threads responsible for executing tasks in parallel. When designing applications with complex or important concurrent operations, users must involve some type of thread hierarchy—where one thread is more important than the other threads and, in some way, it manages the others. In such situations, users must take care of used system resources, the correct way of granting exclusive access, synchronization, and so on. For all these reasons, the user should first design a smart object that will be able to satisfy all the above requirements, while at the same time, provide comfortable development.

Using thread pools

Here, we will explain a thread pool, which can be represented as a list of worker threads that, when available, will execute the required tasks. This is good when an application wants to reduce the possibly high number of threads and provide management for working threads. Another scenario where the application should benefit is when there is a large number of threads responsible for small tasks. In such situations, the overhead for creation and destruction of a large number of threads is significant, and should be avoided with a small number of threads that will execute the entire work.

We see a typical thread pool implementation in the following diagram:



We will follow this pattern when designing the `ThreadPool` class. We'll use a list of threads that will be available for dispatching. In other words, the application will ask for a worker from the pool instead of creating threads. The pool manager—Dispatcher—will manage and schedule workers upon a request from an application.

The Dispatcher is responsible for resource management, alerts and notifications, correct synchronization, and problem solving. It is important to use enough system resources so that the application can execute work in parallel, while at the same time, take care not to overload the system too much. Also, an alert mechanism is required in order for workers to be notified when some event occurs, such as in a situation where the user has made some input or as a result of I/O operation in the background.

Because of concurrent execution, synchronization is required between worker items. Features such as thread local storage must be provided where workers will save/load whatever is necessary for successful task execution. Another very important feature is potential problem solving. In a parallel environment, a lot of problems can occur as a result of a random situation. Even though the application looks properly designed, situations with a deadlock could occur as explained in *Chapter 7, Understanding Concurrent Code Design*. When designing a thread pool, where exclusive access will be featured, a strong protection mechanism should be carefully planned. Yet, if a situation with a deadlock suddenly occurs the Dispatcher must have some ability to resolve the problematic situation and ensure normal execution of tasks.

Our following example will implement the thread pool and use it to transfer funds between two bank accounts.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Console application named `ThreadPool`.
2. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `main`. Open `main.h`. Copy and paste the following code:

```
#ifndef __MAIN__
#define __MAIN__

#include <Windows.h>
#include <tchar.h>
#include <time.h>

#endif
```

3. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `CBankAccount.h`. Open `CBankAccount.h`. Copy and paste the following code:

```
#ifndef __BANKACCOUNT__
#define __BANKACCOUNT__

#include "main.h"

class CLock
{
public:
    CLock(TCHAR* szMutexName);
    ~CLock();
private:
    HANDLE hMutex;
};

inline CLock::CLock(TCHAR* szMutexName)
{
    hMutex = CreateMutex(NULL, FALSE, szMutexName);
    WaitForSingleObject(hMutex, INFINITE);
}

inline CLock::~CLock()
{
    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}

class CBankAccount;

class CParameters
{
public:
    CParameters(CBankAccount* fromAccount, CBankAccount* toAccount,
        double dAmount, bool bPrintOutput = true)
    {
        this->fromAccount = fromAccount;
        this->toAccount = toAccount;
        this->dAmount = dAmount;
        this->bPrintOutput = bPrintOutput;
    }
    CBankAccount* fromAccount;
    CBankAccount* toAccount;
    double dAmount;
};
```

```

        bool bPrintOutput;
    };

class CBankAccount
{
public:
    CBankAccount(double dBalance) : dBalance(dBalance) { uID =
NewId(); *szLock = 0; _tcscpy_s(szLock, LockName()); }
    CBankAccount(double dBalance, TCHAR* szLockName) :
dBalance(dBalance) { uID = NewId(); *szLock = 0; _tcscpy_s(szLock,
szLockName); }
    static DWORD WINAPI Transfer(LPVOID lpParameter);
    double& Balance() { return dBalance; }
    unsigned AccountID() const { return uID; }
    TCHAR* LockName(void);
private:
    unsigned uID;
    double dBalance;
    TCHAR szLock[32];
    static unsigned NewId() { static unsigned uSeed = 61524; return
uSeed++; }
};

TCHAR* CBankAccount::LockName(void)
{
    static int iCount = 0;
    static TCHAR szBuffer[32];

    if (*szLock == 0)
    {
        wsprintf(szBuffer, _T("_lock_%d_"), ++iCount);
    }
    else
    {
        return szLock;
    }

    return szBuffer;
}

DWORD WINAPI CBankAccount::Transfer(LPVOID lpParameter)
{
    CParameters* parameters = (CParameters*)lpParameter;

    CLock* outerLock = new CLock(parameters->fromAccount->szLock);

```



```
    if (parameters->dAmount < parameters->fromAccount->dBalance)
    {
        CLock* innerLock = new CLock(parameters->toAccount->szLock);
        parameters->fromAccount->dBalance -= parameters->dAmount;
        parameters->toAccount->dBalance += parameters->dAmount;
        delete innerLock;
        delete outerLock;

        if (parameters->bPrintOutput)
        {
            _tprintf_s(_T("%ws\n%ws\t\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n\n"),
                L"Transfer succeeded.",
                L"Amount:", parameters->dAmount,
                L"From account:", parameters->fromAccount->AccountID(),
                L"Balance:", parameters->fromAccount->Balance(),
                L"To account:", parameters->toAccount->AccountID(),
                L"Balance:", parameters->toAccount->Balance());
        }

        delete parameters;
        return 1;
    }
    delete outerLock;

    if (parameters->bPrintOutput)
    {
        _tprintf_s(_T("%ws\n%ws\t\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n%ws\n\n"),
            L"Transfer failed.",
            L"Amount:", parameters->dAmount,
            L"From account:", parameters->fromAccount->AccountID(),
            L"Balance:", parameters->fromAccount->Balance(),
            L"To account:", parameters->toAccount->AccountID(),
            L"Balance:", parameters->toAccount->Balance(),
            L"Not enough funds!");
    }

    delete parameters;
    return 0;
}

#endif
```

4. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `CThread.h`. Open `CThread.h`. Copy and paste the following code:

```
#ifndef __THREAD__
#define __THREAD__

#include "main.h"

class CThread
{
public:
    CThread(LPTHREAD_START_ROUTINE lpThreadStart) : dwExitCode(0),
lpThreadStart(lpThreadStart){ }
    ~CThread();
    void Start(LPVOID lpContext = 0);
    bool StillAlive(){ return ExitStatus() == STILL_ACTIVE; }
    HANDLE Handle() const { return hThread; }
    DWORD ThreadId() const { return dwThreadId; }
    DWORD ExitStatus(){ GetExitCodeThread(hThread, &dwExitCode);
return dwExitCode; }
    static DWORD GetThreadId(CThread* cThread);
private:
    LPTHREAD_START_ROUTINE lpThreadStart;
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwExitCode;
};

CThread::~CThread()
{
    if (StillAlive())
    {
        TerminateThread(hThread, (DWORD)-1);
    }
    CloseHandle(hThread);
}

void CThread::Start(LPVOID lpContext)
{
    hThread = CreateThread(NULL, 0, lpThreadStart, lpContext, 0,
&dwThreadId);
}

DWORD CThread::GetThreadId(CThread* cThread)
{
    return cThread->ThreadId();
}

#endif
```

5. Open **Solution Explorer** and right-click on **Header files**. Add an existing header file named `CList`. Open `CList.h` from *Chapter 1, Introduction to C++ Concepts and Features*.
6. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `CThreadPool`. Open `CThreadPool.h`. Copy and paste the following code:

```
#ifndef __THREADPOOL__
#define __THREADPOOL__

#include "main.h"
#include "CList.h"
#include "CThread.h"

class CThreadPool
{
public:
    CThreadPool();
    CThreadPool(unsigned uMaxThreads) : dwMaxCount(uMaxThreads),
    threadList(new CList<CThread>()) { }
    ~CThreadPool() { RemoveAll(); }
    DWORD Count() { return threadList->Count(); }
    void RemoveThread(DWORD dwThreadId);
    void WaitAll();
    void RemoveAll();
    DWORD& MaxCount() { return dwMaxCount; }
    void ReleaseThread(DWORD dwThreadId);
    CThread* RequestThread(LPTHREAD_START_ROUTINE threadStart);
private:
    CList<CThread>* threadList;
    DWORD dwMaxCount;
};

CThreadPool::CThreadPool()
{
    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);
    dwMaxCount = sysInfo.dwNumberOfProcessors;

    threadList = new CList<CThread>();
}

void CThreadPool::RemoveThread(DWORD dwThreadId)
{

```

```

        CThread* thread = threadList->Find(CThread::GetThreadId,
dwThreadId);
        if (thread)
        {
            delete thread;
        }
    }

void CThreadPool::WaitAll()
{
    HANDLE* hThreads = new HANDLE[threadList->Count()];
    CThread* thread = 0;

    for (unsigned uIndex = 0; uIndex < threadList->Count();
uIndex++)
    {
        thread = threadList->GetNext(thread);
        hThreads[uIndex] = thread->Handle();
    }

    WaitForMultipleObjects(threadList->Count(), hThreads, TRUE,
INFINITE);
    delete[] hThreads;
}

void CThreadPool::RemoveAll()
{
    CThread* thread = 0;
    while (thread = threadList->GetNext(thread))
    {
        threadList->Remove(thread);
        thread = 0;
    }
    delete threadList;
}

void CThreadPool::ReleaseThread(DWORD dwThreadId)
{
    CThread* thread = threadList->Find(CThread::GetThreadId,
dwThreadId);
    if (thread != NULL)
    {
        TerminateThread(thread->Handle(), (DWORD)-1);
    }
}

```

```

    }
}

CThread* CThreadPool::RequestThread(LPCTSTR lpThreadStart)
{
    CThread* thread = NULL;
    if (Count() < MaxCount())
    {
        threadList->Insert(thread = new CThread(lpThreadStart));
        return thread;
    }

    while (thread = threadList->GetNext(thread))
    {
        if (!thread->StillAlive())
        {
            break;
        }
        Sleep(100);
    }

    if (thread == NULL)
    {
        thread = threadList->GetFirst();
    }

    threadList->Remove(thread);

    threadList->Insert(thread = new CThread(lpThreadStart));
    return thread;
}

#endif

```

7. Open **Solution Explorer** and right-click on **Source files**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```

#include "main.h"
#include "CBankAccount.h"

```

```

#include "CThreadPool.h"

double Random(double dMin, double dMax)
{
    double dValue = (double)rand() / RAND_MAX;
    return dMin + dValue * (dMax - dMin);
}

int main(void)
{
    srand((unsigned)time(NULL));

#define ACCOUNTS_COUNT 12
    CBankAccount* accounts[ACCOUNTS_COUNT];
    for (unsigned uIndex = 0; uIndex < ACCOUNTS_COUNT; uIndex++)
    {
        accounts[uIndex] = new CBankAccount(Random(10, 1000));
    }

    CThreadPool* pool = new CThreadPool();

#define TASK_COUNT 5
    for (unsigned uIndex = 0; uIndex < TASK_COUNT; uIndex++)
    {
        int nFirstIndex = rand() % ACCOUNTS_COUNT;
        int nSecondIndex = -1;
        while ((nSecondIndex = rand() % ACCOUNTS_COUNT) ==
nFirstIndex){}

        CParameters* params = new CParameters(accounts[nFirstIndex],
accounts[nSecondIndex], Random(50, 200));
        pool->RequestThread(CBankAccount::Transfer)->Start(params);
    }

    pool->WaitAll();
    delete pool;

#ifdef _DEBUG
    return system("pause");
#endif
    return 0;
}

```

How it works...

Let's explain the header files first. In the `main.h` file, we include the necessary Windows headers. Then we create the `CBankAccount` class in `CBankAccount.h`. We'll use this class as an abstraction of a bank account object. We'll implement two helper classes too: `CLock` and `CParameters`. The `CLock` class implements a simple locking mechanism providing exclusive access from the `CLock` object creation until it's destroyed. The `CParameters` class is used for parameter passing to the working thread.

The `CBankAccount` class implements the bank account object. It has the `uID` attribute used for the account ID, `dBalance` used for the fund amount, and `szLock` used for the lock name. It has two constructors: the first where you can set only the balance, while the lock name will be generated using the `LockName` method; the second where you can set both balance and lock name. It has two static methods: `NewId` and `Transfer`. The `NewId` method is used for generating the account ID while `Transfer` is used for transferring actual funds from one account to another. The `LockName` method can be used for obtaining the current lock name from any bank account. The `Transfer` method tries to acquire a lock on the first account called `fromAccount`. When a lock is acquired, it tries to acquire another lock—this time on an account called `toAccount`. When both accounts are locked, the transfer can be performed by simply subtracting the requested amount from `fromAccount` and adding the requested amount to `toAccount`. If the transfer has been done successfully, the method returns 1 while for an unsuccessful transfer—in a situation where the balance is less than the requested amount—the method returns 0.

Furthermore, in `CThread.h`, we've implemented the `CThread` class. The `CThread` object will be used in the thread pool as an abstraction of the working object. It has the `lpThreadStart` attribute used for the thread start address, `hThread` used for thread handle, `dwThreadId` used for thread ID, and `dwExitCode` used for thread exit code. It implements one constructor where the exit code is set to zero while `lpThreadStart` points to actual start address provided by user. In the destructor, we'll query if the thread is still alive using the `StillAlive` method and if so call the `TerminateThread` API and then close the thread handle. The `Start` method creates the thread and (if provided) passes a parameter to its start address. As mentioned before, the `StillAlive` method compares the `STILL_ACTIVE` macro with the `ExitStatus` return value, and returns a **Boolean** value. The `Handle` and `ThreadId` methods return a handle to the thread and thread ID respectively. The `ExitStatus` method calls the `GetExitCodeThread` API to obtain its exit code or `STILL_ACTIVE`.

The `CThreadPool` class implements the actual pool. It has two attributes: a pointer to the list of items and the maximum possible number of threads that can be created by the pool. It has two constructors: the first one where thread max count is set by default using the machine's number of logical processors. The second constructor has one argument—the actual maximum threads that pool can instantiate. In the destructor, we'll use the `RemoveAll` method to free the used resources. The `Count` method returns the count of the items on the list. The `RemoveThread` method searches for the thread specified by the `dwThreadId` argument and if found, removes it. The `WaitAll` method creates an array of thread handles that will be used in the `WaitForMultipleObjects` API specifying `TRUE` for the `BOOL bWaitAll` parameter in order to wait for all threads to finish their tasks. After `WaitForMultipleObjects` returns, it deletes the array of handles. The `RemoveAll` method simply removes all the threads from the list using the `RemoveThread` method for each thread in the list. The `MaxCount` method returns a reference to the `dwMaxCount` attribute in order to get or set its value. The pool can increase or decrease its size, if necessary. The `ReleaseThread` method searches for the thread specified by the `dwThreadId` parameter and if found, terminates the thread. The `RequestThread` method is used for obtaining a free thread. If the maximum number of threads is not reached, the method creates a new item and inserts it into the list. If the maximum number is already reached, the method iterates through the entire list and queries the exit code for each thread.

If all threads are still active, it waits for 0.1 seconds and tries again until it finds the first thread that has finished its task. When the thread is found, the method removes it and creates a new one with the start address provided by the user.

In `main.cpp`, we have created the `Random` function to obtain a random double number in the `dMin` to `dMax` range. In the application entry point, we'll set a random generator to the current time and create twelve bank accounts with random balance values. Then we'll instantiate the pool with five tasks to process. For each task, we'll select a random bank account and then find another (random) bank account that must be different than the first one. After both accounts are found, we'll perform a transfer with a random value, using the `RequestThread` and `Start` methods. After we set all tasks, we'll call the `WaitAll` method in order to wait for the completion of the task. When `WaitAll` returns, we can delete the pool.

There's more...

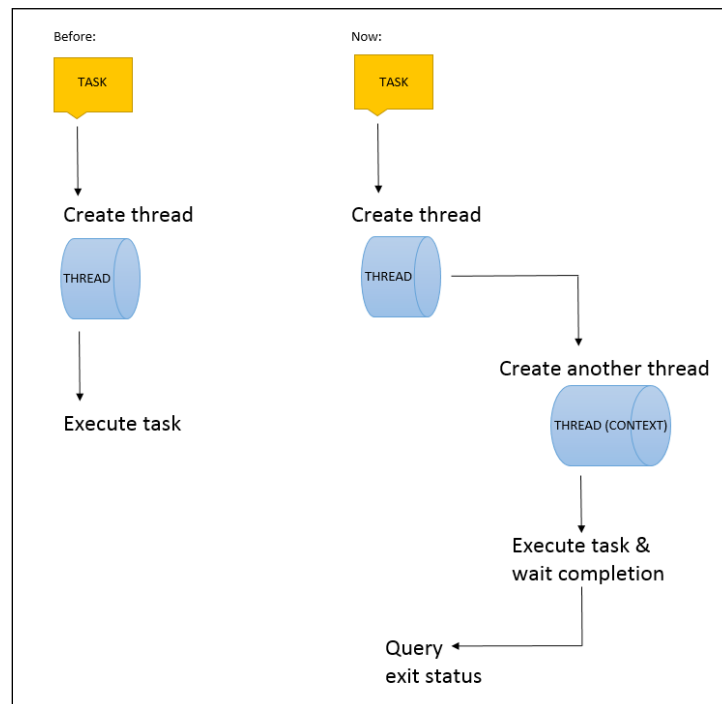
Even though we have carefully designed the pool, paying attention to all the necessary details, situations such as a deadlock can still occur. That will be rare—almost impossible, because the transfer from account *a* to account *b* must occur in the exact same time as transfer from account *b* to account *a*—but still it is possible. Our next example will expand the pool, making it much smarter with a feature to solve eventual problems such as a deadlock.

Customizing the thread pool dispatcher

In our previous example, we had to create the thread pool in order to manage certain tasks—fund transfer from one bank account to another. Our goal was to use enough system resources, but not too much. At the same time, we had to take care of busy and available threads and maintain synchronization. Our following example will expand the thread pool making it a lot smarter and have the ability to solve possible problems that could occur during execution.

Instead of iterating through the thread list and waiting for some thread to complete execution, we'll use message passing in order to select the first thread that completes its task and notify the dispatcher. Instead of simply executing a user-defined function passed as a start address for the thread, we'll add another layer of abstraction in order to have more control over task execution.

In the following diagram, we've illustrated the idea behind the new approach:



Even though we'll duplicate a number of threads, as for each task, we'll create two instead of one thread, we'll still maintain the number of threads. The thread created for the execution of the task will create one more thread and wait for it, only to have more control over the task execution. We'll call the first thread `worker` and we'll call the second thread, which will be created, `context thread`. The idea behind such an approach is that we have more control; if the context thread has been terminated, the worker that waits for it will query its exit code and notify the dispatcher that something went wrong. Or, if the context thread hangs or it is unresponsive, the worker will notice that something went wrong and will perform an appropriate action, again while notifying the dispatcher. This granularity gives us more control and makes the pool much smarter.

Our following example will target a scenario where a deadlock will be intentionally created and we will try to resolve such a situation.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Console application named `CThreadPool`.
2. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `main.h`. Open `main.h`. Copy and paste the following code:

```
#ifndef __MAIN__
#define __MAIN__

#include <Windows.h>
#include <tchar.h>
#include <time.h>

#define MSG_ENDTASK    0x1000
#define MAX_WAIT_TIME 120 * 1000

#endif
```

3. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named CBankAccount. Open CBankAccount.h. Copy and paste the following code:

```
#ifndef __BANKACCOUNT__
#define __BANKACCOUNT__

#include "main.h"

class CLock
{
public:
    CLock(TCHAR* szMutexName);
    ~CLock();
private:
    HANDLE hMutex;
};

inline CLock::CLock(TCHAR* szMutexName)
{
    hMutex = CreateMutex(NULL, FALSE, szMutexName);
    WaitForSingleObject(hMutex, INFINITE);
}

inline CLock::~CLock()
{
    ReleaseMutex(hMutex);
    CloseHandle(hMutex);
}

class CBankAccount;

class CParameters
{
public:
    CParameters(CBankAccount* fromAccount, CBankAccount* toAccount,
        double dAmount, bool bPrintOutput = true)
    {
        this->fromAccount = fromAccount;
        this->toAccount = toAccount;
        this->dAmount = dAmount;
        this->bPrintOutput = bPrintOutput;
    }
    CBankAccount* fromAccount;
    CBankAccount* toAccount;
    double dAmount;
    bool bPrintOutput;
};
```

```

};

class CBankAccount
{
public:
    CBankAccount(double dBalance) : dBalance(dBalance) { uID =
NewId(); *szLock = 0; _tcscpy_s(szLock, LockName()); }
    CBankAccount(double dBalance, TCHAR* szLockName) :
dBalance(dBalance) { uID = NewId(); *szLock = 0; _tcscpy_s(szLock,
szLockName); }
    static DWORD WINAPI Transfer(LPVOID lpParameter);
    double& Balance() { return dBalance; }
    unsigned AccountID() const { return uID; }
    TCHAR* LockName(void);
private:
    unsigned uID;
    double dBalance;
    TCHAR szLock[32];
    static unsigned NewId() { static unsigned uSeed = 61524; return
uSeed++; }
};

TCHAR* CBankAccount::LockName(void)
{
    static int iCount = 0;
    static TCHAR szBuffer[32];

    if (*szLock == 0)
    {
        wsprintf(szBuffer, _T("_lock_%d_"), ++iCount);
    }
    else
    {
        return szLock;
    }

    return szBuffer;
}

DWORD WINAPI CBankAccount::Transfer(LPVOID lpParameter)
{
    CParameters* parameters = (CParameters*)lpParameter;

    Sleep(100);

    CLock* outerLock = new CLock(parameters->fromAccount->szLock);

```

```
if (parameters->dAmount < parameters->fromAccount->dBalance)
{
    CLock* innerLock = new CLock(parameters->toAccount->szLock);
    parameters->fromAccount->dBalance -= parameters->dAmount;
    parameters->toAccount->dBalance += parameters->dAmount;
    delete innerLock;
    delete outerLock;

    if (parameters->bPrintOutput)
    {
        _tprintf_s(_T("%ws\n%ws\t\t%8.2lf\n%ws\t%8u\n%ws\n\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n\n"),
            L"Transfer succeeded.",
            L"Amount:", parameters->dAmount,
            L"From account:", parameters->fromAccount->AccountID(),
            L"Balance:", parameters->fromAccount->Balance(),
            L"To account:", parameters->toAccount->AccountID(),
            L"Balance:", parameters->toAccount->Balance());
    }

    delete parameters;
    return 1;
}

delete outerLock;

if (parameters->bPrintOutput)
{
    _tprintf_s(_T("%ws\n%ws\t\t%8.2lf\n%ws\t%8u\n%ws\n\t%8.2lf\n%ws\t%8u\n%ws\t%8.2lf\n\n"),
        L"Transfer failed.",
        L"Amount:", parameters->dAmount,
        L"From account:", parameters->fromAccount->AccountID(),
        L"Balance:", parameters->fromAccount->Balance(),
        L"To account:", parameters->toAccount->AccountID(),
        L"Balance:", parameters->toAccount->Balance(),
        L"Not enough funds!");
}

delete parameters;
return 0;
}

#endif
```

4. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `CThread.h`. Open `CThread.h`. Copy and paste the following code:

```
#ifndef __THREAD__
#define __THREAD__

#include "main.h"

class CThread
{
public:
    CThread(LPTHREAD_START_ROUTINE lpThreadStart, DWORD
dwDispatcherId);
    ~CThread();
    void Start(LPVOID lpContext = 0){ pThreadContext->lpParameter =
lpContext; ResumeThread(hThread); }
    bool StillAlive(){ return ExitStatus() == STILL_ACTIVE; }
    HANDLE Handle() const { return hThread; }
    HANDLE ContextHandle() const { return pThreadContext->hThread; }
    DWORD ThreadId() const { return dwThreadId; }
    DWORD ContextThreadId() const {
        return pThreadContext->dwThreadId; }
    DWORD ExitStatus(){ GetExitCodeThread(hThread, &dwExitCode);
return dwExitCode; }
    CThread* SetMaxWaitTime(DWORD dwMilliseconds){
        pThreadContext->dwMaxWaitTime = dwMilliseconds;
        return this; }
    static DWORD GetThreadId(CThread* cThread);
protected:
    static DWORD WINAPI StartAddress(LPVOID lpParam);
private:
    typedef struct
    {
        LPTHREAD_START_ROUTINE lpThreadStart;
        LPVOID lpParameter;
        HANDLE hThread;
        DWORD dwThreadId;
        DWORD dwDispatcherId;
        DWORD dwMaxWaitTime;
    } STARTCONTEXT, *PSTARTCONTEXT;
    HANDLE hThread;
    DWORD dwThreadId;
    DWORD dwExitCode;
    PSTARTCONTEXT pThreadContext;
```

```
};

CThread::CThread(LPTHREAD_START_ROUTINE lpThreadStart, DWORD
dwDispatcherId) : dwExitCode(0)
{
    pThreadContext = new STARTCONTEXT();
    pThreadContext->lpThreadStart = lpThreadStart;
    pThreadContext->dwDispatcherId = dwDispatcherId;
    pThreadContext->dwMaxWaitTime = MAX_WAIT_TIME;
    hThread = CreateThread(NULL, 0, StartAddress, this,
        CREATE_SUSPENDED, &dwThreadId);
}

CThread::~CThread()
{
    delete pThreadContext;

    if (StillAlive())
    {
        TerminateThread(hThread, (DWORD)-1);
    }
    CloseHandle(hThread);
}

DWORD CThread::GetThreadId(CThread* cThread)
{
    return cThread->ThreadId();
}

DWORD WINAPI CThread::StartAddress(LPVOID lpParam)
{
    CThread* thread = (CThread*)lpParam;
    thread->pThreadContext->hThread = CreateThread(NULL, 0,
        thread->pThreadContext->lpThreadStart,
        thread->pThreadContext->lpParameter, 0,
        &thread->pThreadContext->dwThreadId);

    DWORD dwStatus = WaitForSingleObject(
        thread->pThreadContext->hThread,
        thread->pThreadContext->dwMaxWaitTime);

    CloseHandle(thread->pThreadContext->hThread);

    PostThreadMessage(thread->pThreadContext->dwDispatcherId,
        MSG_ENDTASK, (WPARAM)thread->dwThreadId, 0);
    return 0L;
}

#endif
```

5. Open **Solution Explorer** and right-click on **Header files**. Add an existing header file named `CList` from *Chapter 1, Introduction to C++ Concepts and Features*.
6. Open **Solution Explorer** and right-click on **Header files**. Add a new header file named `CThreadPool`. Open `CThreadPool.h`. Copy and paste the following code:

```

#ifndef __THREADPOOL__
#define __THREADPOOL__

#include "main.h"
#include "CList.h"
#include "CThread.h"
#include <Wct.h>

#pragma comment (lib, "Advapi32.lib")

class CThreadPool
{
public:
    CThreadPool();
    CThreadPool(unsigned uMaxThreads);
    ~CThreadPool();
    DWORD Count() { return threadList->Count(); }
    void RemoveThread(DWORD dwThreadId);
    void WaitAll();
    void RemoveAll();
    DWORD& MaxCount() { return dwMaxCount; }
    void ReleaseThread(DWORD dwThreadId);
    CThread* RequestThread(LPTHREAD_START_ROUTINE threadStart);
private:
    void InitializePool();
    static DWORD WINAPI ProblemSolver(LPVOID lpParam);
    void ClearMessageQueue() { MSG msg; while (PeekMessage(&msg,
NULL, 0, 0, PM_REMOVE)); }
    HANDLE hProblemSolver;
    HANDLE hEvent;
    CList<CThread>* threadList;
    DWORD dwMaxCount;
};

CThreadPool::CThreadPool()
{
    SYSTEM_INFO sysInfo;
    GetSystemInfo(&sysInfo);
    dwMaxCount = sysInfo.dwNumberOfProcessors;

    InitializePool();

```



```

    }

    CThreadPool::CThreadPool(unsigned uMaxThreads)
    {
        dwMaxCount = uMaxThreads;

        InitializePool();
    }

    CThreadPool::~CThreadPool()
    {
        SetEvent(hEvent);

        if (WaitForSingleObject(hProblemSolver, MAX_WAIT_TIME) !=
            WAIT_OBJECT_0)
        {
            TerminateThread(hProblemSolver, (DWORD)-1);
        }
        CloseHandle(hProblemSolver);

        CloseHandle(hEvent);

        RemoveAll();
        ClearMessageQueue();
    }

    void CThreadPool::RemoveThread(DWORD dwThreadId)
    {
        CThread* thread = threadList->Find(CThread::GetThreadId,
dwThreadId);
        if (thread)
        {
            delete thread;
        }
    }

    void CThreadPool::WaitAll()
    {
        HANDLE* hThreads = new HANDLE[threadList->Count()];
        CThread* thread = 0;

        for (unsigned uIndex = 0; uIndex < threadList->Count();
uIndex++)
        {

```

```

        thread = threadList->GetNext(thread);
        hThreads[uIndex] = thread->Handle();
    }

    WaitForMultipleObjects(threadList->Count(), hThreads, TRUE,
INFINITE);
    delete[] hThreads;
}

void CThreadPool::RemoveAll()
{
    CThread* thread = 0;
    while (thread = threadList->GetNext(thread))
    {
        threadList->Remove(thread);
        thread = 0;
    }
    delete threadList;
}

void CThreadPool::ReleaseThread(DWORD dwThreadId)
{
    CThread* thread = threadList->Find(CThread::GetThreadId,
dwThreadId);
    if (thread != NULL)
    {
        TerminateThread(thread->Handle(), (DWORD)-1);
    }
}

CThread* CThreadPool::RequestThread(LPTHREAD_START_ROUTINE
threadStart)
{
    CThread* thread = NULL;
    if (Count() < MaxCount())
    {
        threadList->Insert(thread = new CThread(threadStart,
GetCurrentThreadId()));
        return thread;
    }

    while (thread = threadList->GetNext(thread))
    {
        if (!thread->StillAlive())
        {

```

```
        break;
    }
}

if (thread == NULL)
{
    MSG msg = { 0 };
    while (GetMessage(&msg, NULL, 0, 0) > 0)
    {
        thread = threadList->Find(CThread::GetThreadId,
            (DWORD)msg.wParam);

        if (thread)
        {
            break;
        }
    }

    threadList->Remove(thread);

    threadList->Insert(thread = new CThread(threadStart,
        GetCurrentThreadId()));
    return thread;
}

void CThreadPool::InitializePool()
{
    MSG msg;
    PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE);

    hEvent = CreateEvent(NULL, TRUE, FALSE, _T("__prbslv_1342__"));
    hProblemSolver = CreateThread(NULL, 0, ProblemSolver, this, 0,
        NULL);

    threadList = new CList<CThread>();
}

DWORD WINAPI CThreadPool::ProblemSolver(LPVOID lpParam)
{
    CThreadPool* pool = (CThreadPool*)lpParam;

    DWORD dwWaitStatus = 0;
    while (true)
    {
        dwWaitStatus = WaitForSingleObject(pool->hEvent, 10);
        if (dwWaitStatus == WAIT_OBJECT_0)
        {
            break;
        }
    }
}
```

```

    }

    CThread* thread = NULL;
    while (thread = pool->threadList->GetNext(thread))
    {
        if (thread->StillAlive())
        {
            HWCT hWct = NULL;
            DWORD dwNodeCount = WCT_MAX_NODE_COUNT;
            BOOL bDeadlock = FALSE;
            WAITCHAIN_NODE_INFO NodeInfoArray[WCT_MAX_NODE_COUNT];

            hWct = OpenThreadWaitChainSession(0, NULL);
            if (GetThreadWaitChain(hWct, NULL,
                WCTP_GETINFO_ALL_FLAGS, thread->ContextThreadId(),
                &dwNodeCount, NodeInfoArray, &bDeadlock))
            {
                if (bDeadlock)
                {
                    if (TerminateThread(thread->ContextHandle(),
                        (DWORD)-1))
                    {
                        _tprintf_s(
#ifdef _UNICODE
                            _T("%ws\n%ws\t[%u]%ws\n\n"),
                        #else
                            _T("%s\n%s\t[%u]%s\n\n"),
                        #endif
                            _T("Error! Deadlock found!"), _T("Thread:"),
                            thread->ContextThreadId(), _T("terminated!"));
                    }
                }
            }
            CloseThreadWaitChainSession(hWct);
        }
        Sleep(1000);
    }
    return 0L;
}

#endif

```

7. Open **Solution Explorer** and right-click on **Source files**. Add a new source file named `main`. Open `main.cpp`. Copy and paste the following code:

```
#include "main.h"
#include "CBankAccount.h"
#include "CThreadPool.h"

int main(void)
{
    int i = 0;
    while (i++ < 10)
    {
        CBankAccount* a = new CBankAccount(200);
        CBankAccount* b = new CBankAccount(200);

        CThreadPool* pool = new CThreadPool(4);

        CParameters* params1 = new CParameters(a, b, 100);
        pool->RequestThread(
            CBankAccount::Transfer)->SetMaxWaitTime(
                6000)->Start(params1);

        CParameters* params2 = new CParameters(b, a, 50);
        pool->RequestThread(CBankAccount::Transfer)->Start(params2);

        pool->WaitAll();
        delete pool;
    }

#ifdef _DEBUG
    return system("pause");
#endif
    return 0;
}
```

How it works...

Let's explain header files first. In `main.h`, we have included necessary Windows headers. We also define two macros: `MSG_ENDTASK` used for posting the message about a completed task from the worker to the dispatcher and `MAX_WAIT_TIME` in order for the thread not to wait too long for its context thread. The maximum waiting time could be increased or decreased from the `CThread` object using the `SetMaxWaitTime` method, which we will explain later.

The `CBankAccount` class implemented in `CBankAccount.h` along with `CLock` and `CParameters` remains the same. The `Transfer` method is changed in a way that at the beginning of the method, a sleep time of 0.1 second is added. This is intentionally added in order to create a situation with deadlock. After two threads are started, if we were to remove the sleep time—probably deadlock wouldn't occur because the system will give enough time on the CPU to thread that it won't request for lock in the same time as another thread.

Furthermore, in `CThread.h`, we've implemented a much smarter `CThread` class. We are using the `CThread` object in the thread pool as an abstraction of a working object. It has the `lpThreadStart` attribute used for the thread start address, `hThread` used for the thread handle, `dwThreadId` used for the thread ID, `dwExitCode` used for thread exit code, and `pThreadContext` used for the context thread. As illustrated in earlier diagram, we are creating another thread, which we'll call the context thread, in order to gain more control over execution. The context implementation is assembled as an internal structure consisting of the following attributes: `lpThreadStart` is used for the actual user-defined start address; `lpParameter` is passed by the user for the start address; `hThread` is used for the context thread handle; `dwThreadId` is used for the context thread ID; `dwDispatcherId` is used as an ID for the pool manager; and `dwMaxWaitTime` is used for setting the timeout for waiting for the context thread. The `CThread` class implements one constructor where the exit code is set to zero. Constructor arguments are slightly changed, besides `lpThreadStart`, `dwDispatcherId` is added in order to pass the actual manager identifier to the thread. In constructor, we create a `STARTCONTEXT` object where we set `lpThreadStart` with a user-defined start address. The `dwDispatcherId` will be passed, as well as `dwMaxWaitTime` with `MAX_WAIT_TIME` default value and we create a worker thread with the `CREATE_SUSPENDED` flag. In the destructor, we first delete the memory used by `pThreadContext` and then we query if the thread is still alive using the `StillAlive` method and if so, call the `TerminateThread` API and then close the thread handle. The `Start` method sets the `lpParameter` attribute passed by the user and resumes the worker thread. The `StillAlive` method compares the `STILL_ACTIVE` macro with the return value of `ExitStatus`, and returns a Boolean value. The `Handle` and `ThreadId` methods return a handle to the worker thread and worker thread ID respectively. The `ContextHandle` and `ContextThreadId` methods return a handle to the context thread and the context thread ID respectively. The `ExitStatus` method calls the `GetExitCodeThread` API to obtain its exit code or `STILL_ACTIVE`. The `SetMaxWaitTime` method sets the maximum waiting time attribute value. The `StartAddress` method is added in order to perform task execution by creating another thread the context thread—and wait for its completion, in the worst case, a maximum of the `pThreadContext->dwMaxWaitTime` interval. Either the time interval has passed or the context thread has returned. Then, the context thread handle is closed and the `MSG_ENDTASK` message is posted to the dispatcher message queue.

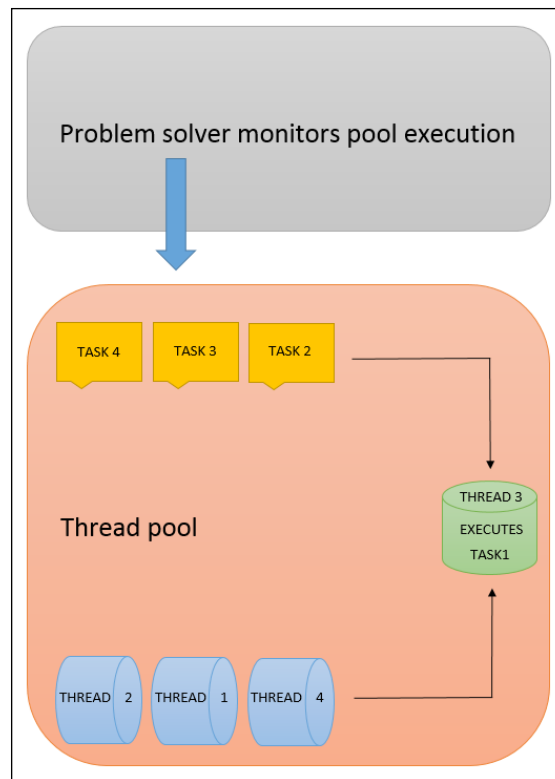
The `CParameters` class remains the same. Instead of searching through the list with a specified wait time, we'll block the dispatcher and wait for the `MSG_ENDTASK` message posted from the first available worker thread.

In the `CThreadPool` class, we include an additional header file called `wct.h`. We also add the `Advapi32` library in order to use the API that related to the thread-wait chains, which we'll explain below. The `CThreadPool` class has four attributes: a handle to the `hProblemSolver` thread that will occasionally check for potential problems; the `hEvent` handle to manually reset the event, used for alerting the problem solver thread to exit; the `threadList` pointer to the list of items; and `dwMaxCount` for the maximum possible number of threads that can be created by the pool. The class implements two constructors: the first where the thread max count is set by default, using the number of machine logical processors. The second constructor has one argument—the actual maximum number of threads that a pool can instantiate. Both constructors call the `InitializePool` method that creates the message queue for the dispatcher, also create an event, a problem solver thread, and a new list of threads. In the destructor, we first set the event to the `signaled` state and wait for `MAX_WAIT_TIME` for the problem solver to safely exit. If the time has passed without thread completion, we use the `TerminateThread` API to end the execution of the thread forcibly. Then we close both handles and we use the `RemoveAll` method to free the used resources and the `ClearMessageQueue` method to empty the message queue.

The `Count` method returns the count of the items in the list. The `RemoveThread` method searches for the thread specified by the `dwThreadId` parameter and if found, removes it. The `WaitAll` method creates an array of thread handles that will be used in the `WaitForMultipleObjects` API specifying `TRUE` for the `BOOL bWaitAll` parameter in order to wait for all threads to finish their tasks. After `WaitForMultipleObjects` returns, we delete the array of handles. The `RemoveAll` method simply removes all threads from the list using the `RemoveThread` method for each thread in the list. The `MaxCount` method returns a reference to the `dwMaxCount` attribute in order to get or set its value. Again, the pool size can be increased or decreased if necessary. The `ReleaseThread` method searches for the thread specified by the `dwThreadId` parameter and if found, terminates the thread.

The `RequestThread` method is used for obtaining free thread. If the maximum number of threads is not reached, the method creates a new item and inserts it into the list. If the maximum number of threads is reached, `RequestThread` iterates through the entire list and queries the exit code for each thread. If all threads are still active, the dispatcher will block using the `GetMessage` API and wait until the first available worker thread posts the message after finishing its task. We then search for the thread using the passed thread ID and if found, we remove it and create a new thread with the start address provided by the user. The `ProblemSolver` method is used as a monitoring routine for the thread pool.

As shown in the following diagram, it is important that the problem solver is another thread executing in parallel:



If a problem occurs and the pool is blocked for some reason, this thread will try to resolve the problem. That's why it's imperative that it does not execute under any thread that belongs to the pool and could be blocked along with the pool. Its task is to loop forever, or until the program exits, when the event is set to the `signaled` state. So, on each iteration, it will call `WaitForSingleObject` with a handle to the event passed along with a 0.01 second wait interval. If the event is set, the thread will exit. If not, it will iterate through the entire thread list, and it'll call the `StillAlive` method for each thread. If the thread is alive, it'll call the `OpenThreadWaitChainSession` API that creates WCT or Wait Chain Traversal.

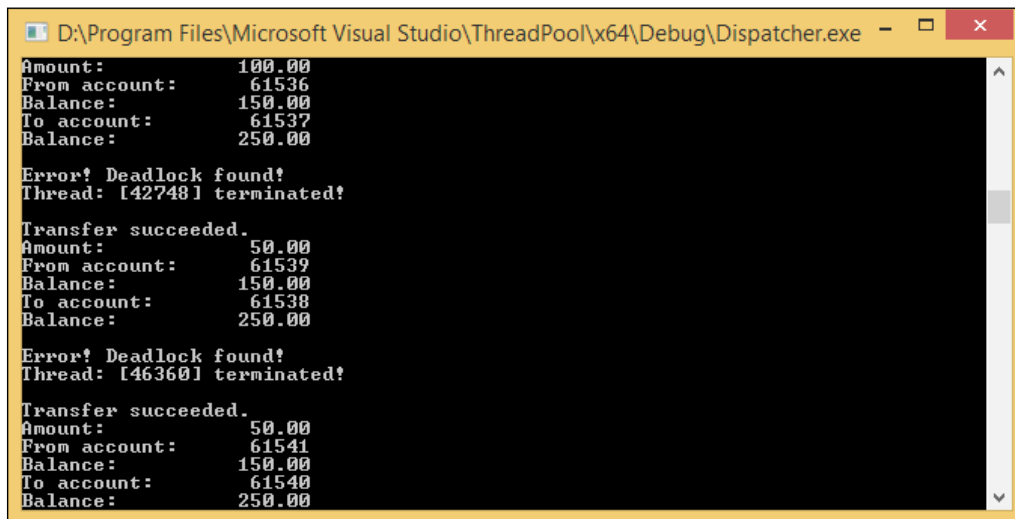


Wait Chain Traversal (WCT) enables diagnosing eventual application hangs and deadlocks. A wait chain is an alternating sequence of threads and synchronization objects; each thread waits for the object that follows it, which is owned by the subsequent thread in the chain (from MSDN).

Next, a call to `GetThreadWaitChain` follows. This API retrieves the wait chain for the specified thread. If any subset of nodes in the array forms a cycle, the function sets the `IsCycle` parameter to `TRUE` (from MSDN). So, if a cycle is formed, `bDeadlock` will be set to `TRUE`. If so, we terminate the deadlocked thread and close the WCT session using the `CloseThreadWaitChainSession` API. We then suspend the thread for one second and continue monitoring the pool execution.

In the application entry point, we create two bank accounts with certain amounts of money. After we've created the pool, we request for the thread to transfer funds from the first to the second account, and we request for another thread to transfer funds from the second to the first account, trying to create a problematic situation. We perform this in a loop that iterates ten times only to ensure that a deadlock will occur. If a deadlock is found, the problem solver will react.

One situation with a deadlock is shown in the following screenshot:



```
D:\Program Files\Microsoft Visual Studio\ThreadPool\x64\Debug\Dispatcher.exe
Amount:      100.00
From account: 61536
Balance:     150.00
To account:   61537
Balance:     250.00

Error! Deadlock found!
Thread: [42748] terminated!

Transfer succeeded.
Amount:      50.00
From account: 61539
Balance:     150.00
To account:   61538
Balance:     250.00

Error! Deadlock found!
Thread: [46360] terminated!

Transfer succeeded.
Amount:      50.00
From account: 61541
Balance:     150.00
To account:   61540
Balance:     250.00
```

There's more...

The explained example targets a certain possible scenario where a problem can occur. This situation is known as the **Banker's algorithm**, given by Edsger Dijkstra. It is most likely that a given pool implementation, along with problem solving, could be used for any scenario where a deadlock or a similar problem can occur. Our intention was to teach the user to carefully design the objects that will help them develop the program more easily, at the same time giving them the ability to control the execution, respect the resources used, and perform advanced management and monitoring of program execution. In general, as we said so many times, it is important to carefully design the application, and if necessary, to include advanced mechanisms to support such a design.

Using remote threading

So far, we were focusing on situations where we had total control of all application parts and the execution itself. In practice, this is far from the truth. In many situations, you would have to continue where some other developer stopped or you would have to use compiled parts or even external applications where changing the code is not possible. Even in such a situation, sometimes there is a need to change some application behavior, which we will explain later.

When we are unable to change the code, we could use **remote threading**. Remote threading is a feature presented by the Windows operating system, starting from XP OS, and maintained in all newer versions. A remote thread is actually a thread like any other, with the difference being that the remote thread does not execute under the current process context but in the address space of the remote process. That is an excellent feature with the earlier described scenarios.

There are few ways in which you can start a remote thread: using the `CreateRemoteThread` API, using the `SetWindowsHookEx` API, and using the undocumented `NtCreateThreadEx` API.

We'll focus on the `CreateRemoteThread` API. In our following example, we'll assume that Windows calculator (`calc.exe`) is some external application whose code we can't change, and yet we have to alter its main window title (for some reason). Even though we could change the title of the calculator window from our application without using a remote thread, we chose this scenario for the simplicity of the example.

Getting ready

Make sure Visual Studio is up and running.

How to do it...

Now, let's create our program and explain its structure. Perform the following steps:

1. Create a new empty C++ Console application named `RemoteThreading`.
2. Open **Windows Explorer** and navigate to your solution folder. Create a new text file named `common.txt`. Change its extension to `.h`.
3. Open **Solution Explorer** and right-click on the **RemoteThreading** solution. Select the **Add Existing Item** option and add `common.h`. Open `common.h`. Copy and paste the following code:

```
#include <windows.h>
#include <tchar.h>

#define MAPPING_NAME _T("__comm_61524_map__")
```

```
#define EVENT_NAME    _T("__evnt_68435_rst__")
```

```
typedef struct  
{  
    HMODULE hLibrary;  
} COMM_OBJECT, *PCOMM_OBJECT;
```

4. Open **Solution Explorer** and under the project **RemoteThreading** right-click on **Source files**. Add a new source file named `main`. Open `main.cpp`.
5. Add the following code:

```
#include <windows.h>  
#include <tchar.h>  
#include "..\common.h"  
  
int __stdcall RemoteLoadLibrary(HANDLE hProcess, char*  
szLibraryName)  
{  
    LPTHREAD_START_ROUTINE lpLoadLibrary = (LPTHREAD_START_ROUTINE)  
GetProcAddress(GetModuleHandleA("Kernel32.dll"), "LoadLibraryA");  
    if (lpLoadLibrary == NULL)  
    {  
        return -1;  
    }  
  
    if (hProcess == NULL)  
    {  
        _tprintf_s(_T("Handle to Process 0x%x was NULL!\n"), (int)  
hProcess);  
        return -1;  
    }  
  
    size_t uMemSize = strlen(szLibraryName) + 1;  
    void* lpRemoteMem = VirtualAllocEx(hProcess, NULL, uMemSize,  
MEM_COMMIT, PAGE_READWRITE);  
    if (lpRemoteMem == NULL)  
    {  
        _tprintf_s(  
#ifdef _UNICODE  
        _T("%ws\nError:\t%u\n"),  
#else  
        _T("%s\nError:\t%u\n"),  
#endif  
        _T("Could not allocate remote virtual memory!"),  
        GetLastError());  
        return -1;  
    }  
}
```

```

        BOOL bSuccess = WriteProcessMemory(hProcess, lpRemoteMem,
        szLibraryName, uMemSize, NULL);
        if (bSuccess == STATUS_ACCESS_VIOLATION || bSuccess == FALSE)
        {
            _tprintf_s(
                _T("Could not write remote virtual memory!\nError:\t%u\n"),
                GetLastError());
            VirtualFreeEx(hProcess, lpRemoteMem, 0, MEM_RELEASE);
            return -1;
        }

        HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
        lpLoadLibrary, lpRemoteMem, 0, NULL);
        if (hThread == NULL)
        {
            _tprintf_s(
                _T("Could not create remote thread!\nError:\t%u\n"),
                GetLastError());
            VirtualFreeEx(hProcess, lpRemoteMem, 0, MEM_RELEASE);
            return -1;
        }

        WaitForSingleObject(hThread, INFINITE);

        VirtualFreeEx(hProcess, lpRemoteMem, 0, MEM_RELEASE);
        CloseHandle(hThread);

        return 0;
    }

    int __stdcall RemoteFreeLibrary(HANDLE hProcess, HMODULE hLib)
    {
        LPTHREAD_START_ROUTINE lpLoadLibrary = (LPTHREAD_START_ROUTINE)
        GetProcAddress(GetModuleHandleA("Kernel32.dll"), "FreeLibrary");
        if (lpLoadLibrary == NULL)
        {
            return -1;
        }

        if (hProcess == NULL)
        {
            _tprintf_s(_T("Handle to Process 0x%x was NULL!\n"), (int)
        hProcess);
            return -1;
        }

        HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
        lpLoadLibrary, hLib, 0, NULL);
    }

```

```
    if (hThread == NULL)
    {
        _tprintf_s(
            _T("Could not create remote thread!\nError:\t%u\n"),
            GetLastError());
        return -1;
    }

    WaitForSingleObject(hThread, INFINITE);

    CloseHandle(hThread);

    return 0;
}

int main(void)
{
    char* szLibrary = PHYSICAL_PATH_TO_YOUR_DLL;

    HANDLE hMapping = CreateFileMapping((HANDLE)-1, NULL,
        PAGE_READWRITE, 0, sizeof(COMM_OBJECT), MAPPING_NAME);
    HANDLE hEvent = CreateEvent(NULL, TRUE, FALSE, EVENT_NAME);

    STARTUPINFO startInfo = { 0 };
    PROCESS_INFORMATION processInfo = { 0 };

    BOOL bSuccess = CreateProcess(
        _T("C:\\Windows\\System32\\calc.exe"), NULL, NULL,
        NULL, FALSE, 0, NULL, NULL, &startInfo, &processInfo);

    if (!bSuccess)
    {
        _tprintf_s(_T("Error:\t%u\n"), GetLastError());
    }
    else
    {
        WaitForSingleObject(processInfo.hThread, 500);
        RemoteLoadLibrary(processInfo.hProcess, szLibrary);

        WaitForSingleObject(hEvent, INFINITE);

        PCOMM_OBJECT pCommObject = (PCOMM_OBJECT)
        MapViewOfFile(hMapping, FILE_MAP_READ, 0, 0, 0);

        if (pCommObject)
        {

```

```

        RemoteFreeLibrary(processInfo.hProcess,
            pCommObject->hLibrary);
    }
}

CloseHandle(hEvent);
CloseHandle(hMapping);

#ifdef _DEBUG
    return system("pause");
#endif

    return 0;
}

```

6. Open **Solution Explorer** and add a new empty C++ console application project named **RemoteStartAddress**. Select **Application Type – DLL**. Select **Additional Options – Empty Project**. Right-click on **Source files**. Add a new source file named **main**. Open **main.cpp**.
7. Add the following code:

```

#include <Windows.h>
#include <tchar.h>
#include "..\common.h"

typedef struct
{
    DWORD dwProcessId;
    HWND hWnd;
} WINDOW_INFORMATION, *PWINDOW_INFORMATION;

BOOL IsMainWindow(HWND hWnd)
{
    return GetWindow(hWnd, GW_OWNER) == (HWND)0 &&
        IsWindowVisible(hWnd);
}

BOOL CALLBACK EnumWindowsCallback(HWND hWnd, LPARAM lParam)
{
    PWINDOW_INFORMATION pWindowInformation = (PWINDOW_INFORMATION)
        lParam;
    DWORD dwProcessId = 0;

    GetWindowThreadProcessId(hWnd, &dwProcessId);
}

```

```
        if (pWindowInformation->dwProcessId == dwProcessId &&
            IsMainWindow(hWnd))
        {
            pWindowInformation->hWnd = hWnd;
            return FALSE;
        }

        return TRUE;
    }

HWND FindMainWindow()
{
    WINDOW_INFORMATION wndInfo = { GetCurrentProcessId(), 0 };

    EnumWindows(EnumWindowsCallback, (LPARAM)&wndInfo);

    return wndInfo.hWnd;
}

void ChangeWindowTitle()
{
    HWND hWnd = FindMainWindow();
    if (hWnd)
    {
        SetWindowText(hWnd,
            _T("Remotely started thread inside calculator!"));
    }
}

BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID
lpReserved)
{
    switch (dwReason)
    {
        case DLL_PROCESS_ATTACH:
        {
            HANDLE hMapping = OpenFileMapping(FILE_MAP_ALL_ACCESS,
FALSE, MAPPING_NAME);
            if (hMapping)
            {
                PCOMM_OBJECT pCommObject = (PCOMM_OBJECT)
MapViewOfFile(hMapping, FILE_MAP_ALL_ACCESS, 0, 0, 0);
                if (pCommObject)
```

```

        {
            pCommObject->hLibrary = hInstance;
        }

        CloseHandle(hMapping);
    }

    ChangeWindowTitle();

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE,
        EVENT_NAME);
    if (hEvent)
    {
        SetEvent(hEvent);
        CloseHandle(hEvent);
    }
    break;
}

case DLL_PROCESS_DETACH:
{
    //
    break;
}
}
return TRUE;
}

```

How it works...

First, we have created `common.h`, in order to define the structure, event, and mapping name that both projects will use. We must divide our program in two parts: **exe** and **dll**. We must create **exe** in order to run an application. We must create a separate **dll** because the remote thread start address will reside in that dll. In our previous examples, all thread start addresses were in the same project as the application itself, because the thread was executing in the address space of the current process. In this example, the thread will execute in the address space of another process, making all the memory addresses from current process useless.

When we are creating a thread, we must provide a start address or the address of the function where thread will start. If we define a function inside the current process and try to start a remote thread using the address of that function, a call to `CreateRemoteThread` will fail because the addresses inside the current process are private to it and the external process can't use them. That's why it is important to define the function inside a dll and load that dll into the target process context. We can achieve dll loading using the `LoadLibrary` API.

Now, let's explain the dll itself. Just like the application (exe) the dynamic link library must have its own entry point. Its prototype must match the following signature:

```
BOOL WINAPI DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID  
lpReserved);
```

When the dll is loaded, its execution starts from this method. You could distinguish reasons as to why this module was loaded. Those can be as follows:

- ▶ The `DLL_PROCESS_ATTACH` method notifying that the application just loaded the dll
- ▶ The `DLL_PROCESS_DETACH` method notifying that the application just unloaded the dll
- ▶ The `DLL_THREAD_ATTACH` method notifying that the application where the dll is loaded just created a new thread
- ▶ The `DLL_THREAD_DETACH` method notifying that the thread is exiting and some cleanup can be done

We do all our work inside `DLL_PROCESS_ATTACH`. Inside our main application, we create a file mapping necessary for communication between the application and the dll, which we'll explain later. Inside the dll, we first open the file mapping, using the `OpenFileMapping` API. If the mapping is opened, we write the handle to the dll into the `hLibrary` attribute. We need this handle later to unload the library from the target process, when we finish our work. Next, we call the `ChangeWindowTitle` method. This function tries to find the main window handle using `FindMainWindow`. If a handle to the window is found, we can alter the window title.

The `FindMainWindow` method uses the `EnumWindows` API to enumerate all top-level windows. User must provide a pointer to a user-defined function (called `callback`) as the first argument to `EnumWindows`. The second parameter is a user-defined object that will be passed inside a callback. This callback is called by the operating system for each window handle that the OS enumerates. We can use this feature to query each window handle for owning the thread (process). We can achieve this using the `GetWindowThreadProcessId` API.

If we find a window that belongs to the target process, we should also check if that is the main window. For that task, we use the `IsMainWidow` method. This method calls `GetWindow` with `GW_OWNER` as the second parameter and if the value returned is 0 (zero), it will query the window visibility using the `IsWindowVisible` API, making sure that, if visible, it is the main application window from the target process. After `ChangeWindowTitle` returns, we open an event that we use for notifying our application that the job inside the target process is done and we can unload the library. We set the event to the `signaled` state using the `SetEvent` API.

Now, let's explain the main application. The `RemoteLoadLibrary` method is used to create a remote thread inside the target process. First, we need to obtain the address of the `LoadLibrary` API (`LoadLibraryA`—multibyte version used for this example). This API will be used as a remote thread start address with our dll name specified as a parameter to load. In this way, we will load the specified dll into the target process context and perform our execution inside it. After we've obtained the address of `LoadLibrary`, we must allocate enough virtual memory, using the `VirtualAllocEx` API, inside target process in order to write the dll name inside. Pay attention that specifying only the dll name isn't correct because the address of an array of characters inside the current process isn't useful in the target process.

We must write the dll name into the target process, making the address of the array of characters valid inside the target process. After allocating virtual memory, we use the `WriteProcessMemory` API to perform the write operation. If the write operation is successful, we create a remote thread using the `CreateRemoteThread` API. We must provide the target process handle with the correct privileges. We didn't pay much attention to handle privileges because our application creates the target process, creating the returned handle with all possible privileges that we would need.

In situations where we could not create a target process, we would have to open an existing process, specifying the `PROCESS_ALL_ACCESS` flag as a first parameter for the `OpenProcess` API. Only the handle returned with such privileges can be used for the `CreateRemoteThread` API. After successful creation of a remote thread, we use the `WaitForSingleObject` API to wait for thread completion and return. The `RemoteFreeLibrary` method is used for dll unload. First, we must obtain the address of the `FreeLibrary` API in order to start a remote thread whose task will be to run `FreeLibrary` with the handle to the dll passed as a parameter. In that way, a remote thread will unload the specified dll with the handle passed.

Inside the application entry point, first we must set the physical path to the compiled dll. This is important because the remote thread inside the target process will try to load the dll from the given path. This path must be absolute for successful loading. Next, we create a file mapping that we use to communicate with the dll. This is important because when we want to unload the library, we must provide a handle to the dll to the `FreeLibrary` API. A handle to the loaded dll inside the target process is known only after successful loading. That's why we're writing to the file mapping from the target process only after the dll is attached. Next, we create an event that we'll use inside the dll to signal the main application that the handle is written to the file mapping and the dll can be unloaded. Then we'll start the calculator using the `CreateProcess` API. If a process is created we wait for its initialization specifying 0.5 second in the `WaitForSingleObject` API. Then we try to create a remote thread and wait for an event object. After the event is set to the signaled state, we can unload the library using `RemoteFreeLibrary` and close all used handles.

See also...

Where do we go from here? Well, we tried to cover most of themes related to concurrency in programming using the C++ language. Of course, there still remain a lot of details that everyone who wants to use parallelism must learn but our intention is to give you a good foundation. In order to manage concurrent execution, one must have thorough knowledge of operating systems, compilers, object-oriented programming, execution, runtime, and of course the C++ language if that is the developer's choice. As we said so many times, practice is the best teacher. So, get ready, and start creating!

Appendix

In this appendix, we will cover the following topics:

- ▶ Installing MySQL Connector/C
- ▶ Installing WinDDK – Driver Development Kit
- ▶ Setting up a Visual Studio project for driver compilation
- ▶ Using the DebugView application
- ▶ Setting up a Visual Studio project for OpenMP compilation

Installing MySQL Connector/C

In order to use **MySQL Connector/C**, you need to perform the following operations:

1. Open your web browser and navigate to <http://dev.mysql.com/downloads/connector/c/>.

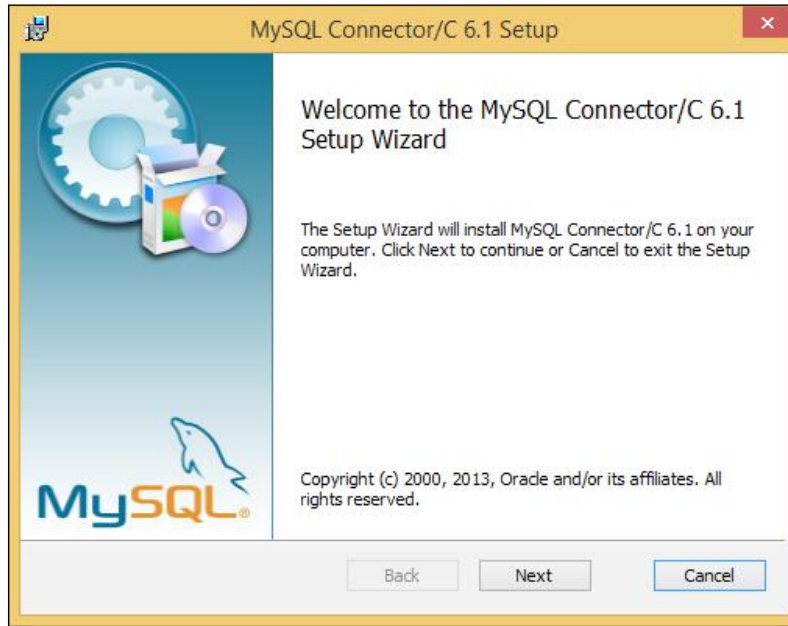
Generally Available (GA) Releases

Connector/C 6.1.3

Select Platform:
Microsoft Windows

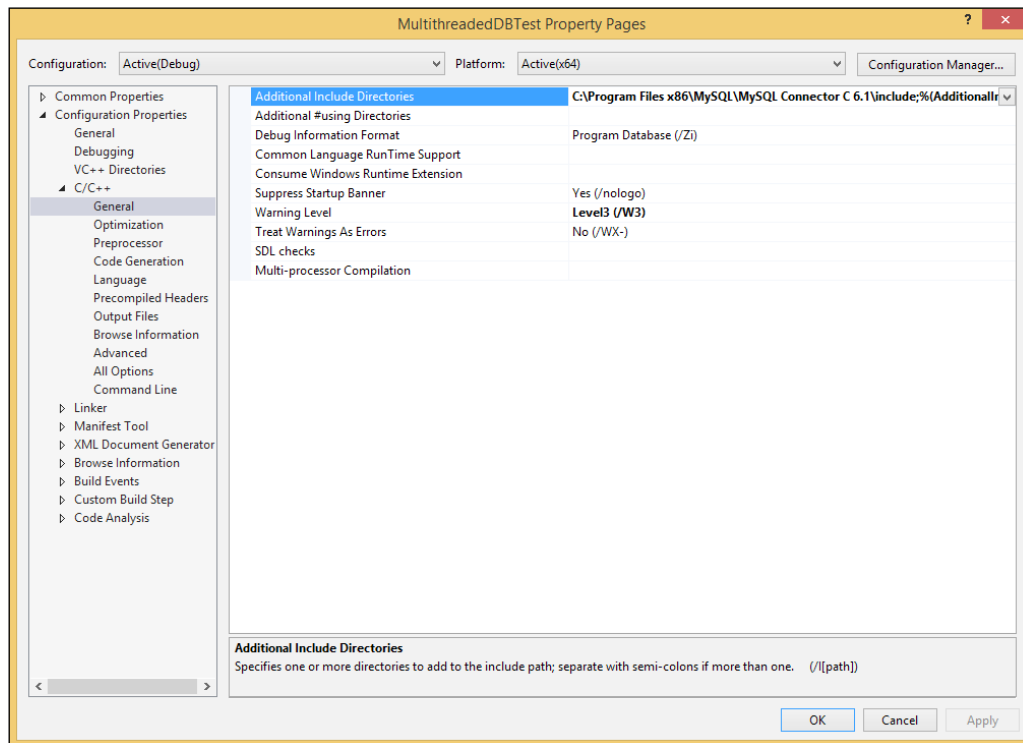
Windows (x86, 32-bit), MSI Installer	6.1.3	21.8M	Download
(mysql-connector-c-6.1.3-win32.msi) MD5: 5984eda023ff990b24aee939ff91e23e Signature			
Windows (x86, 64-bit), MSI Installer	6.1.3	23.3M	Download
(mysql-connector-c-6.1.3-winx64.msi) MD5: 122da8ec5cdd3a70384d301c13747834 Signature			
Windows (x86, 32-bit), ZIP Archive	6.1.3	24.9M	Download
(mysql-connector-c-6.1.3-win32.zip) MD5: 86c5885e535db01197bd982974b0be23 Signature			
Windows (x86, 64-bit), ZIP Archive	6.1.3	26.3M	Download
(mysql-connector-c-6.1.3-winx64.zip) MD5: 41f648e42f35bb99f4d91dd03a78bf93 Signature			

2. Download the `mysql-connector-c-6.1.3-win32.msi` or `mysql-connector-c-6.1.3-win64.msi` file, depending on your architecture and project configuration, and then start the installation as shown in the following screenshot:



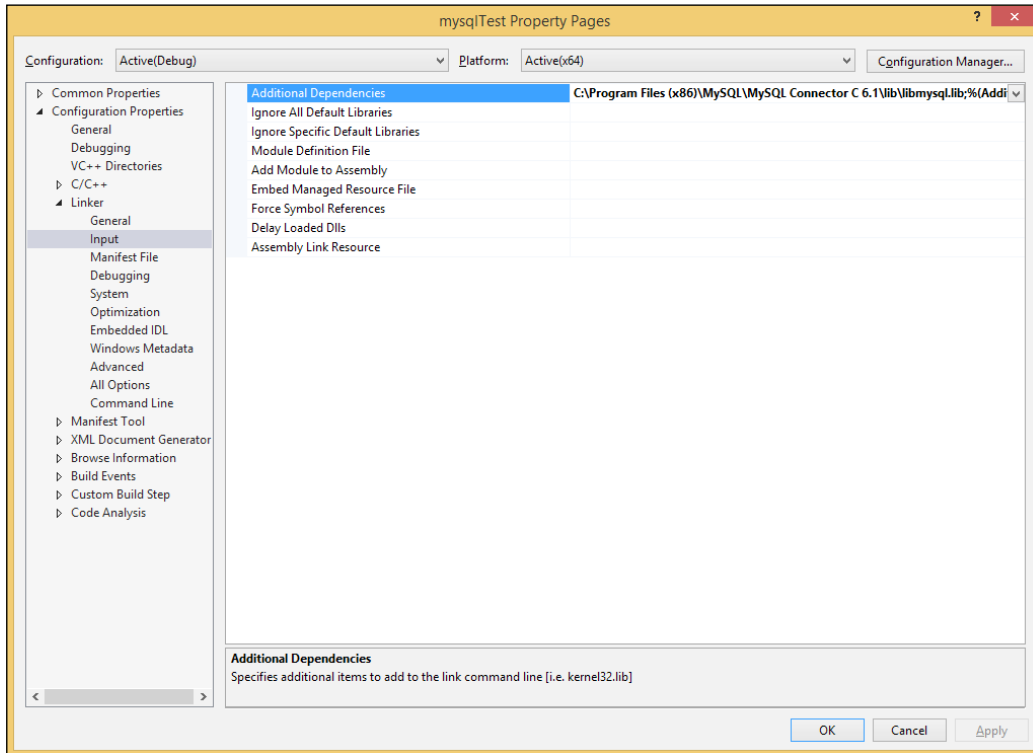
3. Accept the terms in the license agreement and select the **Complete** type of installation. Click on **Finish** and open Visual Studio, and then open the **MultithreadedDBTest** solution.

4. Open **Solution explorer** and right-click on the **MultithreadedDBTest** project. Select **Properties**. Under **Configuration Properties**, expand the **General** node. Open the combobox next to **Additional Include Directories**, and using the Windows folder browser, navigate to `C:\Program Files\MySQL\MySQL Connector C 6.1\include`, as shown in the following screenshot:



5. Click on **Select Folder**.

- Under **Configuration Properties**, expand the **Linker** node. Open the combobox next to **Additional Dependencies** and enter `C:\Program Files\MySQL\MySQL Connector C 6.1\lib\libmysql.lib`, as shown in the following screenshot:



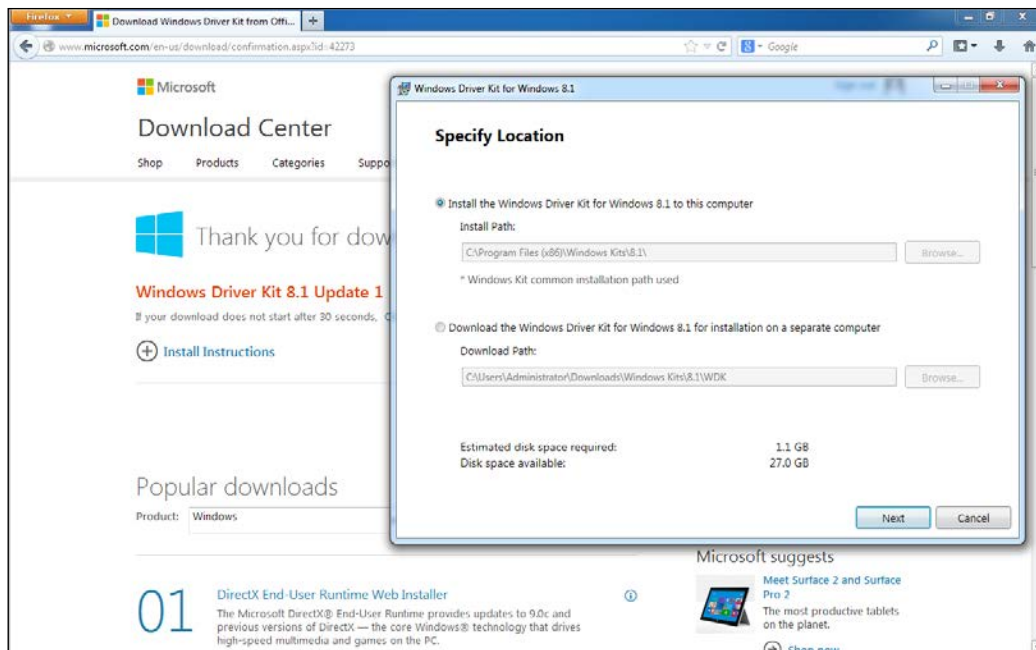
- Using the Windows file explorer, navigate to `C:\Program Files\MySQL\MySQL Connector C 6.1\lib`. Copy the `libmysql.dll` file to `C:\Windows\System32`.

Now, you can compile and run the **MultithreadedDBTest** project or any other project that uses MySQL Connector/C.

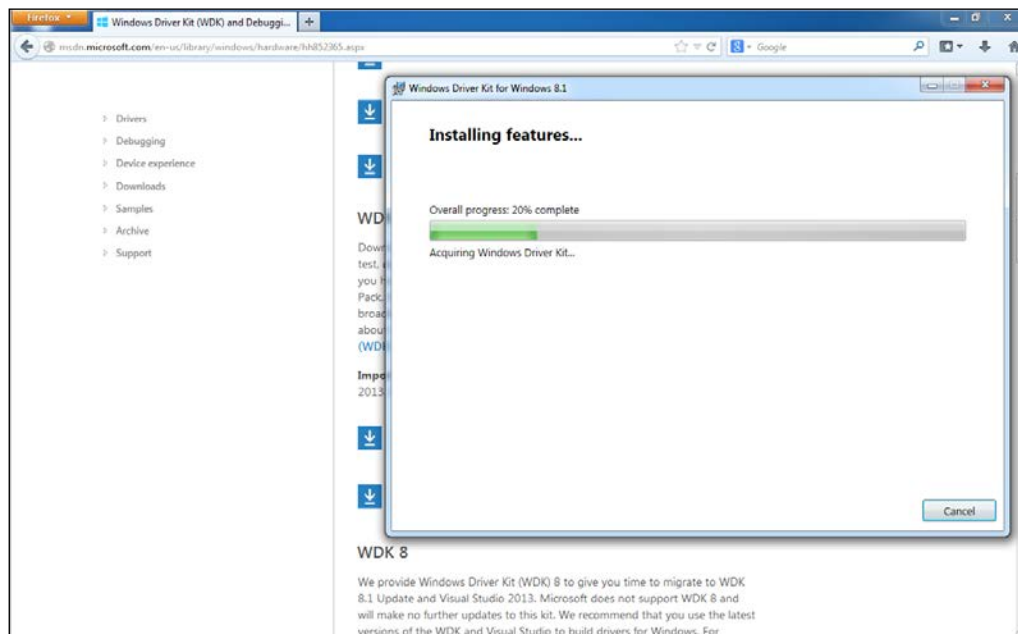
Installing WinDDK – Driver Development Kit

In order to use Windows Driver Kit, you need to perform the following operations:

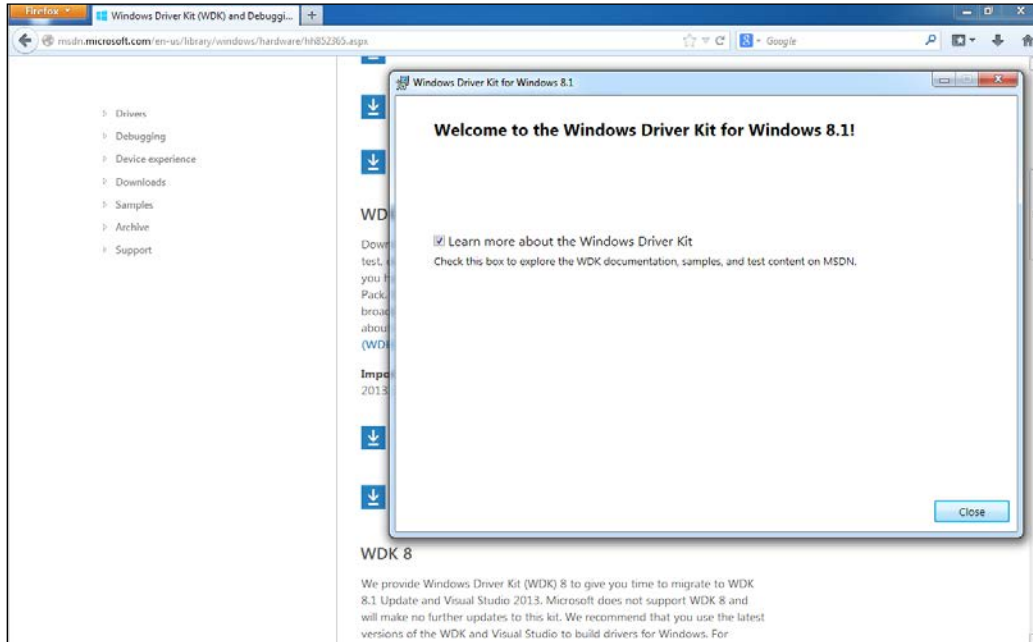
- Open your web browser and navigate to <http://msdn.microsoft.com/en-us/library/windows/hardware/hh852365.aspx>. Since we are using Visual Studio 2013, you'll need to download WDK 8.1 or use the following address: <http://go.microsoft.com/fwlink/p/?LinkId=393659>. The following is a screenshot of the WDK download page:



2. Select **Next** and choose whether you want to participate in **Customer Experience Improvement Program**, and then accept **License Agreement** to start the installation. You'll see a window as shown in the following screenshot:



3. Wait for the installation to complete and close the wizard.

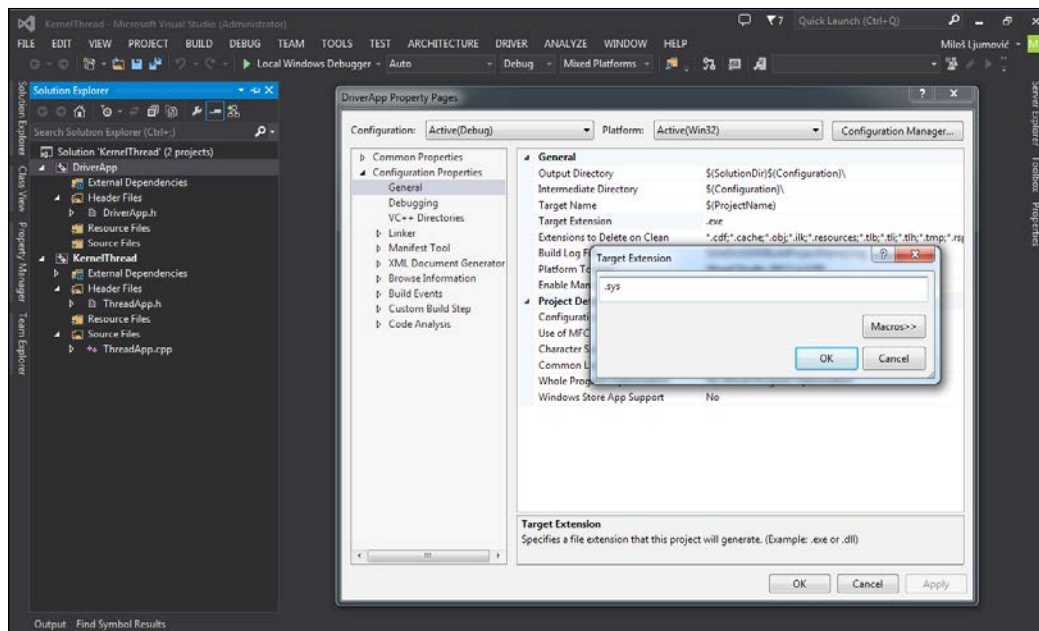


Now, Windows Driver Kit is successfully installed.

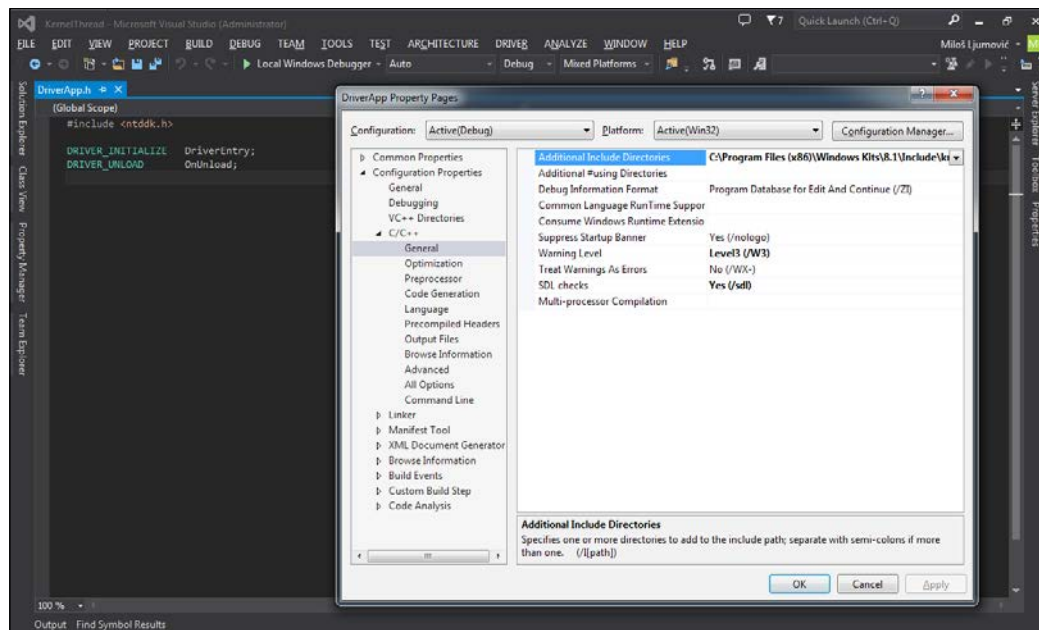
Setting up a Visual Studio project for driver compilation

In order to use the features from Windows Driver Kit, such as compilers and libraries, you need to set up the Visual Studio project properties. To do this, perform the following operations:

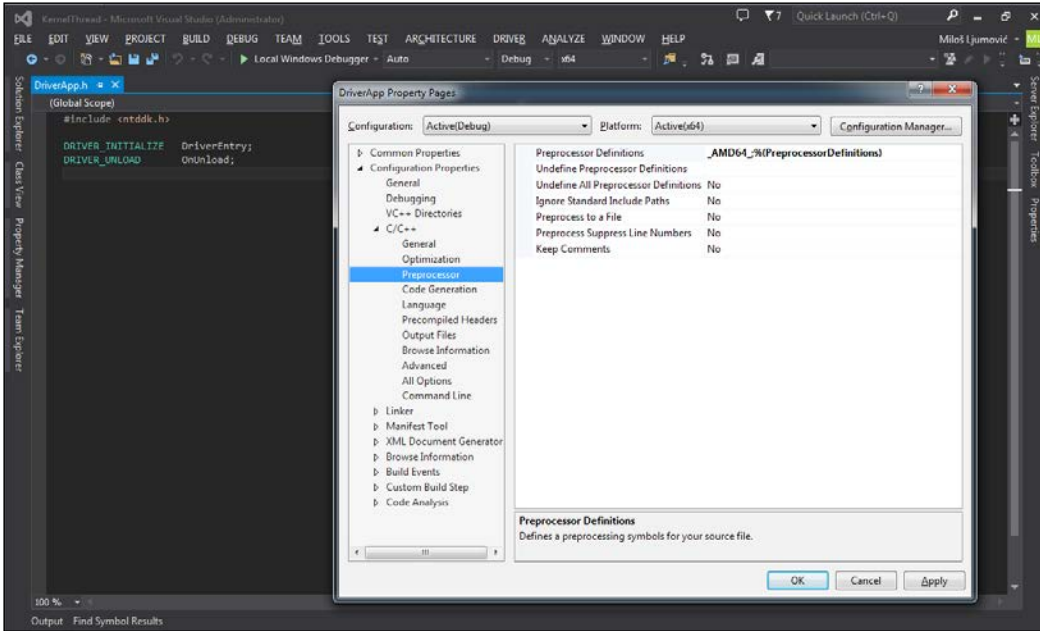
1. If you have followed the instructions from *Chapter 2, Thread Concept*, the example *Implementing threads in kernel* – open the solution **KernelThread**. In **Solution Explorer**, right-click on the **DriverApp** project and select **Properties**. Under the **General** tab, select **Target Extension** and choose **Edit**. Rename **.exe** to **.sys**. Click on **Apply**, as shown in the following screenshot:



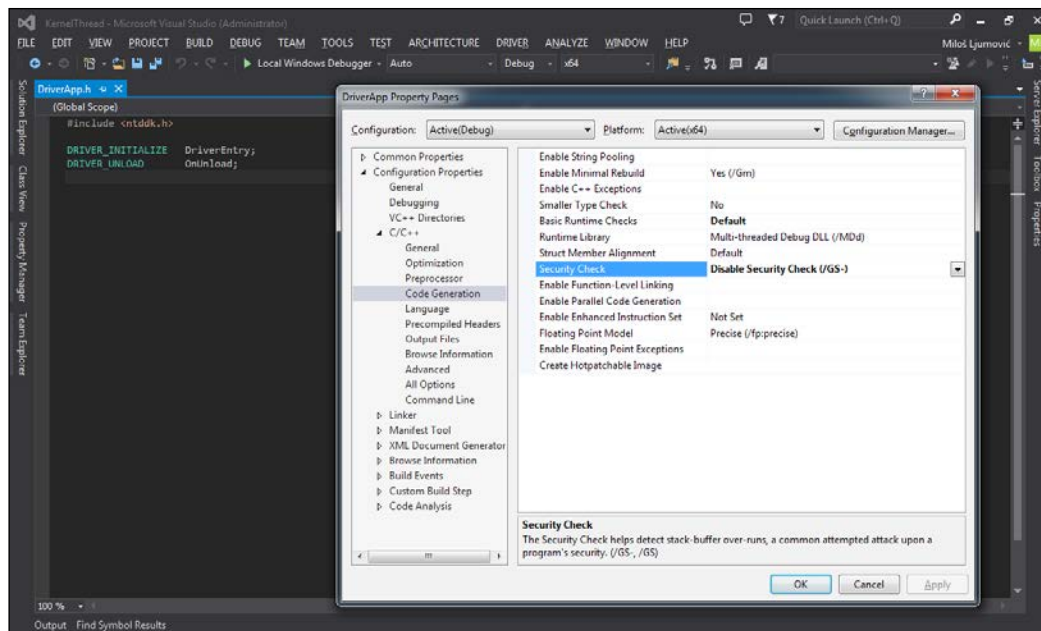
- Now, expand the **C++** node, and under **General**, select **Additional Include Directories**. Click on **Edit** and enter the following Windows path: `C:\Program Files (x86)\Windows Kits\8.1\Include\km`. Click on **OK** and then on **Apply**.



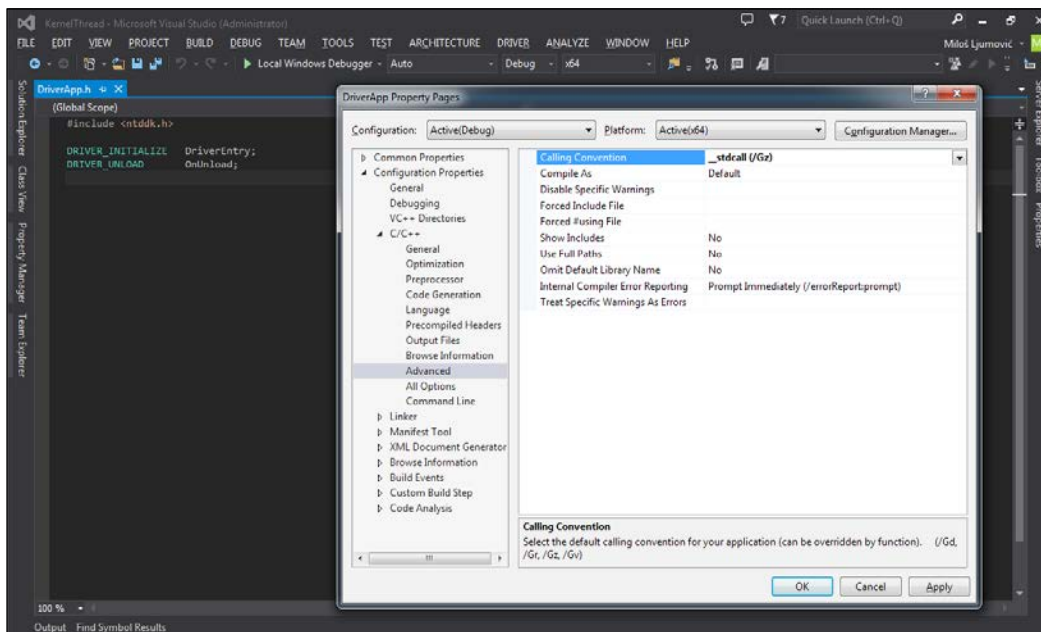
- Now, under **C++**, select **Preprocessor**, and under **Preprocessor Definitions**, choose **Edit**. Remove all the lines and enter `_x86_` if the **Win32** configuration is active, or `_AMD64_` if the **x64** configuration is active, as follows:



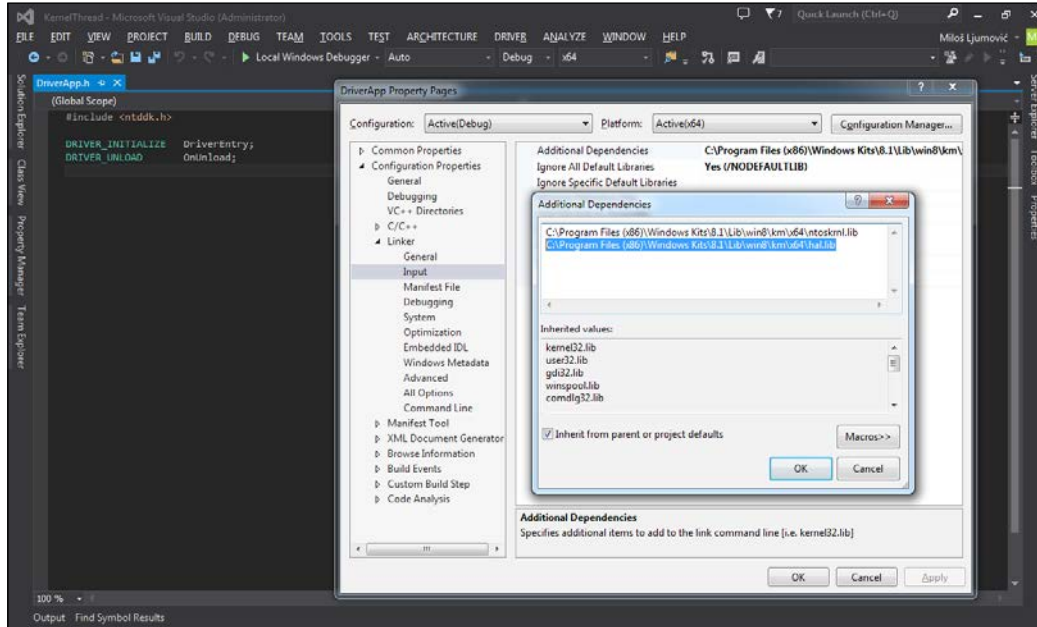
- Click on **OK** and then on **Apply**. Now, under **Code Generation**, select **Enable C++ Exceptions** and delete all the text and leave it blank. Click on **Apply**. Under **Basic Runtimes Check**, select **Default** and then click on **Apply**. Under **Security Check**, select **Disable Security Check (/GS-)** and click on **Apply**.



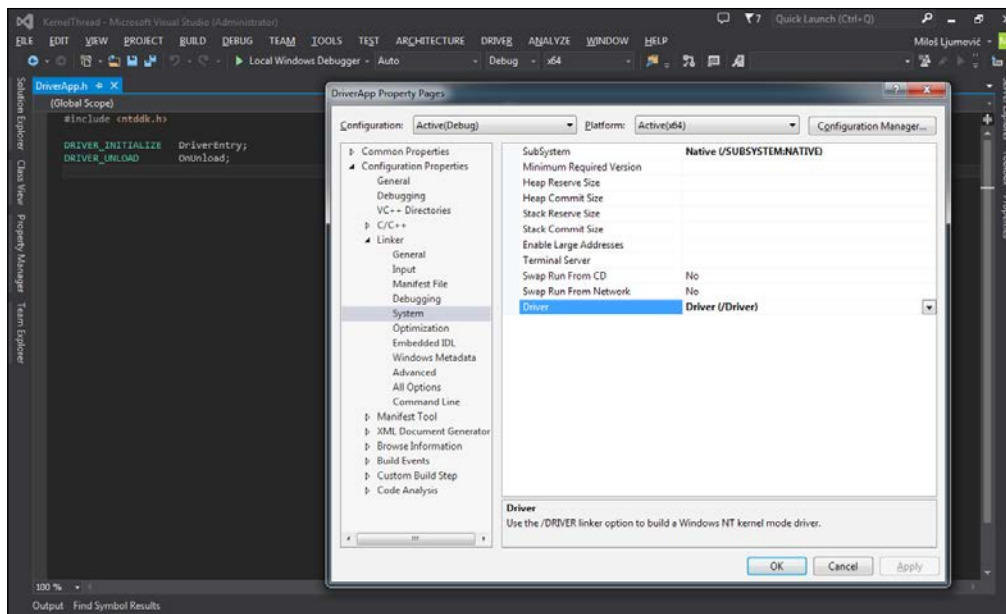
- Now, under **Advanced**, select **Calling convention** and choose **__stdcall (/Gz)** as follows:



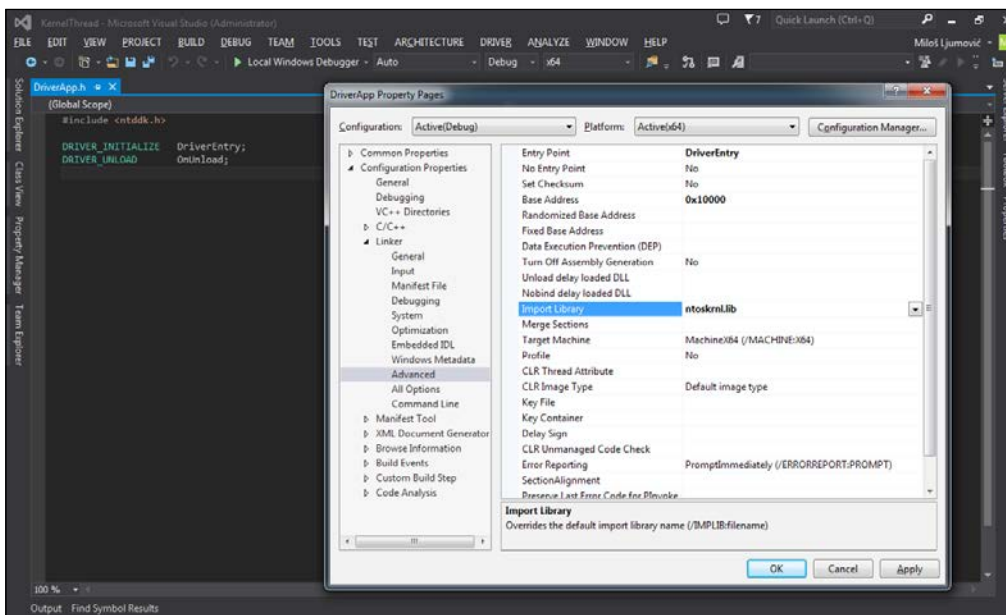
- Now, expand the **Linker** node, and under **General**, select **Enable Incremental Linking** and choose **No (INCREMENTAL:NO)**. Now, under **Input**, select **Additional Dependencies** and click on **Edit**. Enter the two following lines: `C:\Program Files (x86)\Windows Kits\8.1\Lib\win8\km\x64\ntoskrnl.lib` and `C:\Program Files (x86)\Windows Kits\8.1\Lib\win8\km\x64\hal.lib`. Pay attention to change **x64** to **x86**, depending on your active solution configuration:



- Click on **OK** and select **Apply**. Now, expand the **Manifest File** node and select **Generate Manifest**. From the drop-down list, select **No (/MANIFEST:NO)**. Click on **Apply**. Now, expand the **System** node and select **SubSystem**. From the drop-down list, choose **Native (/SUBSYSTEM:NATIVE)**. Select **Driver** and from the drop-down list, choose **Driver(/Driver)**. Click on **Apply**.



8. Finally, select the **Advanced** node and then select **Entry Point**. Insert the text `DriverEntry`. Under **Base Address**, insert the text `0x10000`. Then, select **Randomize Base Address** and delete all the text and leave it blank. Then, select **Data Execution Prevention** and delete all the text and leave it blank. Select **Import Library** and insert the text `ntoskrnl.lib`.

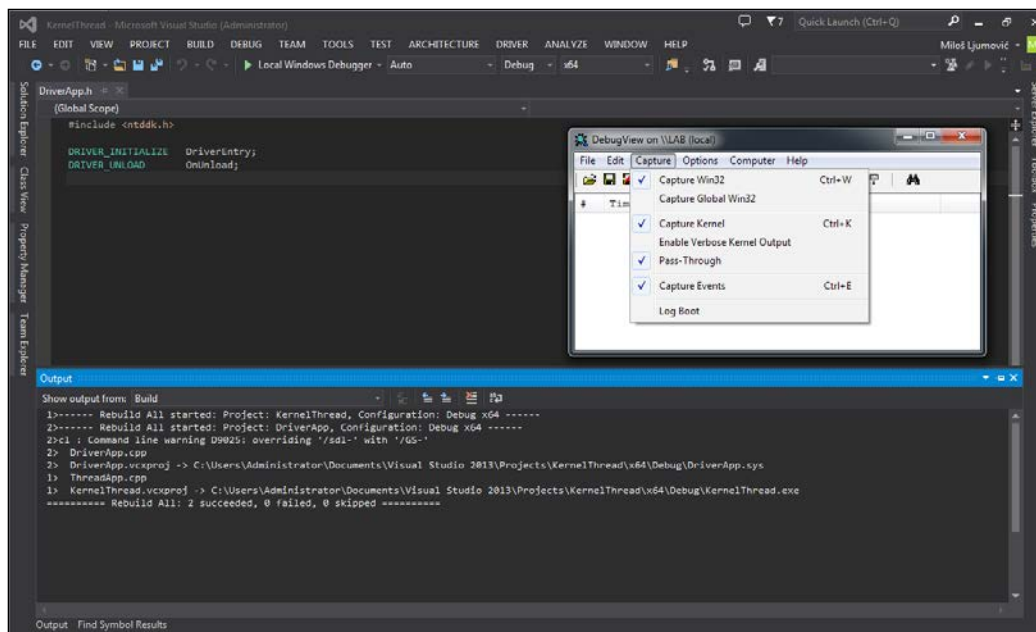


Now, you can compile and run the project **DriverApp**. Note that if you're using a 64-bit operating system, starting a digitally-signed driver is not possible, until you make changes to **Advanced Boot Options**; Vista and the later versions of Windows support the F8 **Advanced Boot Option**, that is, **Disable Driver Signature Enforcement**, which disables the load-time signature enforcement for a kernel-mode driver only for the current system session. This setting does not persist across system restarts (from MSDN). For more information on this, visit [http://msdn.microsoft.com/en-us/library/windows/hardware/ff547565\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff547565(v=vs.85).aspx).

Using the DebugView application

In order to display the output from the kernel driver, you'll need to install the DebugView application. Perform the following operations:

1. Open your web browser and navigate to <http://technet.microsoft.com/en-us/sysinternals/bb896647.aspx>. Download the DebugView ZIP file and extract it to a folder. Run the DebugView application and, from the menu, select **Capture**. Add a tick to **Capture Kernel** or press **Ctrl + K**.

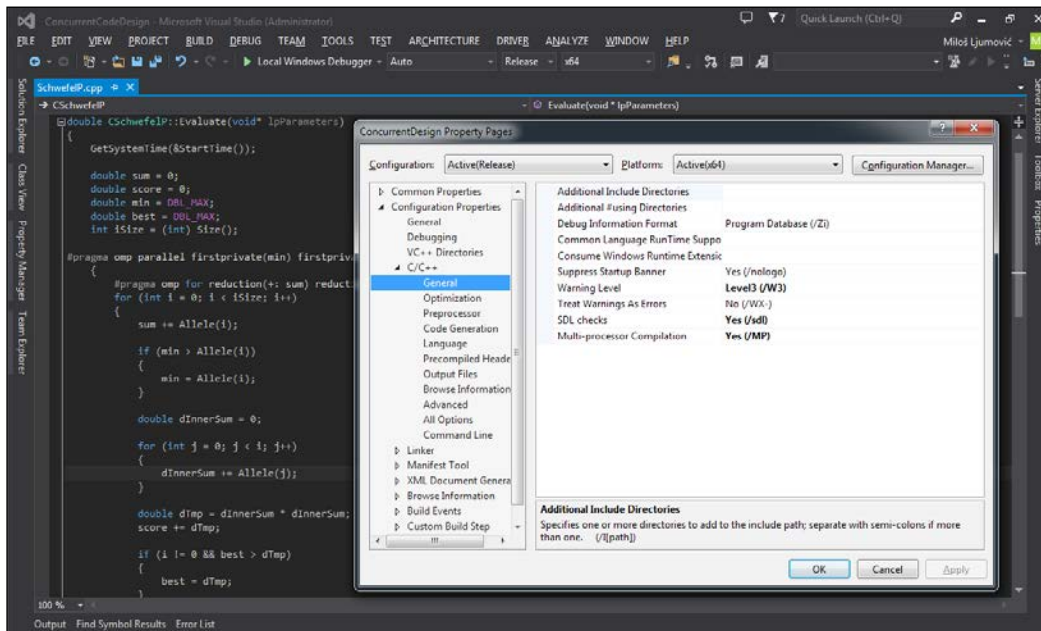


2. Now, if you run your **KernelDriver** project, you will see the output from the driver, using DebugView.

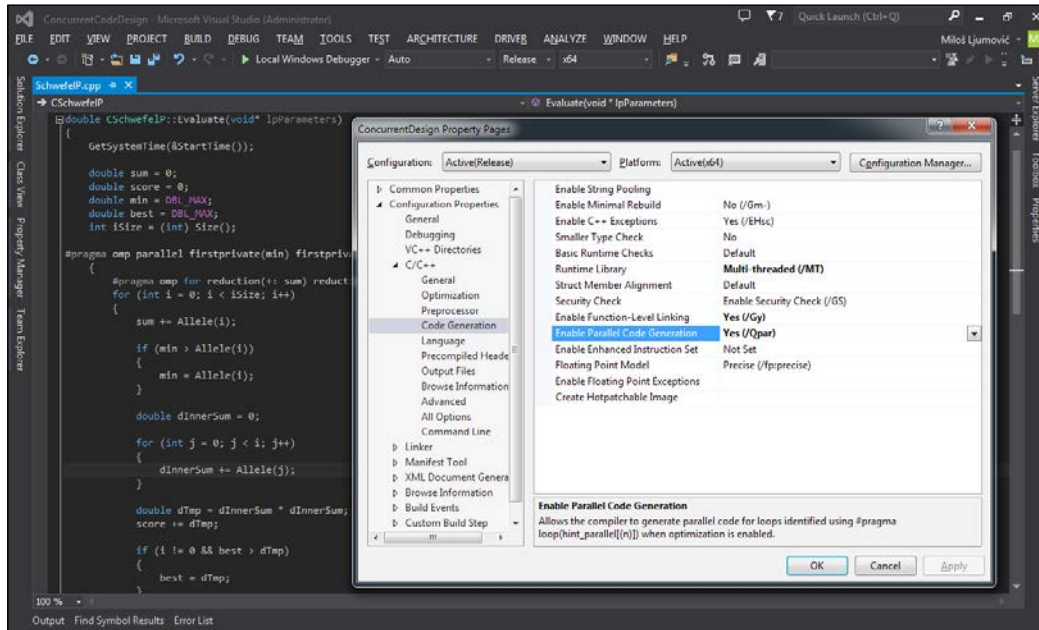
Setting up a Visual Studio project for OpenMP compilation

In order to use OpenMP, you need to perform the following operations:

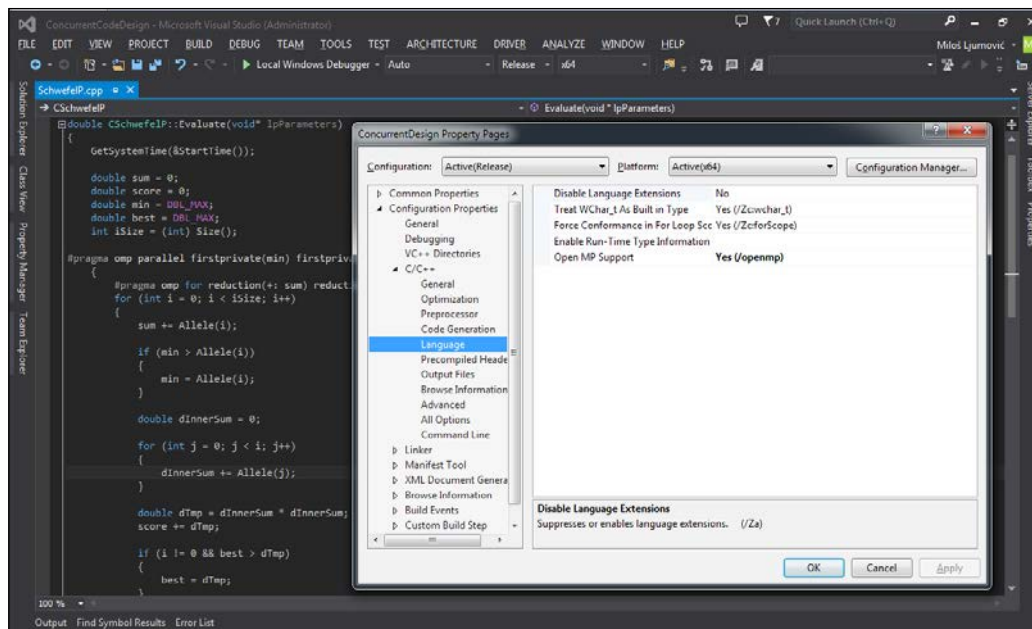
1. If you have followed the instructions in *Chapter 7, Concurrent Code Design*, the example *Improvement of Performance Factors* – open the solution **KernelIT ConcurrentDesign**. In **Solution Explorer**, right-click on the **ConcurrentDesign** project and select **Properties**. Under the **Configuration Properties** tab, expand the **C++** node and select **General**, and then select **Multi-processor Compilation**. Select **Yes (/MP)** and click on **Apply**.



- Now, under **Code Generation**, select **Enable Parallel Code Generation** and then choose **Yes (/Qpar)**. Click on **Apply**.



- Now, under **Language**, select **Open MP Support** and choose **Yes (/openmp)**. Click on **Apply**.



Now, you can compile and run the **ConcurrentDesign** project or any other project that uses the OpenMP support.

Index

Symbols

(int _tmain(int, char)) prototype 12**

.NET

background thread 243

Barrier class 299

EAP 269

foreground thread 243

synchronization mechanisms 247

threading 235-242

thread safety 261, 262

^ operator 242

*** operator 242**

& operator 242

% operator 242

\WM_PLOTDATA message 154

A

Abort method 281

acquire fence 293

Alert method 121

Append method 42

application

creating, from scratch 316

Application::EnableVisualStyles method 242

AsyncDownloadDataCompleted method 276

B

background thread

differentiating, with foreground

thread 243-247

BackgroundWorker class

about 276

using 276

working 280

BackgroundWorker thread

aborting, with Cancellable

method 286-291

Banker's algorithm 370

Barrier::AddParticipant method 300

Barrier class 299

Barrier object

used, for synchronization 300-307

Barrier::RemoveParticipant method 300

Barrier::SignalAndWait method 300

base priority 169, 170

bitwise OR (|) 127

blocked state, process 50

Blue Screen Of Death (BSOD) 104

Boolean value 352

bounded-buffer problem. *See*

producer-consumer problem

button click 142

BUTTON_CLOSE macro 146

C

CalculateSomething function 14

Cancellable method

used, for aborting BackgroundWorker

thread 286-290

CBankAccount class 352

C++/CLI 234

CComplex class 19

child-control identifier 146

C++ language 10

client process

reviewing 231

CNode class 42

Common Language Infrastructure (CLI) 234

- Common Language Runtime (CLR) 234**
- compiled application (*.exe) 234**
- ComplexAdd function 16**
- Composite Actions scenarios 312**
- concurrency**
 - correctness problem 310
 - liveness problem 310
- constructor (ctor) 42**
- ContinueWith method 291**
- correctness problem**
 - about 310
 - occurring, instances 314
- Count method 42**
- cout object 12**
- CParameters class 352**
- C++ program**
 - #include <iostream> header 11
 - creating 11-14
 - example 13
- C++ project**
 - creating 9, 10
- CPU multitasking system 46**
- Create method 120**
- CreateNamedPipe API**
 - reference, URL 165
- CreateProcess function**
 - about 49
 - on MDN, URL 47
- CreateThread API**
 - on MDN, URL 81
- CreateWindow API 32**
- critical region 56**
- critical section**
 - about 56, 139, 206
 - using 206-215
- CSchwefelMT class**
 - creating 325-327
 - working 328
- CSchwefelP derived class**
 - working 337-340
- CThread class 120**
- CThreadPool class 353**
- CWorker class, CPerson 22**

D

- daemon tasks 216**
- Database Management System (DBMS) 88**
- Data Race scenarios 310**
- deadlock**
 - about 260
 - avoiding 260
- DebugView application**
 - installing 392
 - using 392
- DestroyWindow API 33**
- destructor (dtor) 42**
- dining philosophers problem**
 - solving 66-74
- dispatcher**
 - about 175, 176, 343
 - customizing 355-366
 - working 366-370
- dValue 11**
- dwMaxWaitTime attribute 367**
- dynamic binding 28**
- dynamic library (*.dll) 14**
- dynamic priority 169, 170**

E

- EAP**
 - about 269
 - advantage 275
 - down side 275
 - example 275, 276
 - program, creating 269-275
 - working 275
- Elapsed(TCHAR*) method 323**
- Elapsed(void) method 323**
- Evaluate method 328**
- Evaluate(void*) method 323**
- event**
 - using 197-206
- Event-based Asynchronous Pattern. *See* EAP**
- event-driven system 141**
- event handler**
 - about 29
 - for Windows 29

event object 136-138
event synchronization object 197
eventual problem solving feature 343
event wait handles
 signaling 263-268
ExitProcess function 50

F

Fibonacci sequence 339
file mapping 64
FindMainWindow method 378
Find method 42
First In First Out (FIFO) 34
foreground thread
 differentiating, with background
 thread 243-247
Fourier series 175
full fence 293

G

Garbage Collector (GC) 242
gcnew operator 242
GetAsyncState method 121
GetFirst method 42
GetHandle method 121
GetId method 121
GetLast method 42
GetMessage API 32
GetNext method 42
GetUserData method 121
GET_X_LPARAM macro 147
GET_Y_LPARAM macro 147
Graphical User Interface (GUI) 9
GWLP_USERDATA macro 155

H

half fences (volatile memory objects) 293
Hardware Abstraction Layer (HAL) 234
Head method 42
hPrev parameter 30
hThis parameter 30
hThread attribute 367
Hyper-Threading Technology 338

I

iCount 11
Inconsistent Synchronization scenarios 311
inheritance
 about 19
 example, reviewing 20-22
 working 22
InitializeApp method 240
Insert method 42
installation
 MySQL Connector/C 381-384
 WinDDK 384-386
IntelliSense engine 9
Interprocess Communication. *See* **IPC**
Interrupt method 281
int _tmain(int argc, TCHAR* argv[]) prototype
 argc argument 12
 argv argument 12
int _tmain(void) prototype 12
IPC
 about 55, 230
 example 55-59
 working 63-65
IPCDemo 64
IPC problems
 dining philosophers, solving 65-67
IPCWorker 63

J

Java Runtime Environment (JRE) 235
Join method 121

K

kernel
 threads, implementing in 99-104

L

LABEL_TEXT macro 146
Last In First Out (LIFO) 34
linear linked list
 using 35-41
linear list 34

listener thread
 creating 230
list view 121
liveness problem 310, 314
LoadLibrary API 379
lock
 implementing 254-260
Lock class 254
LockName method 352
IParam parameter 155

M

main function
 prototypes 12
main thread 8, 105
managed code
 advantages 234
 disadvantages 235
 versus unmanaged code 234, 235
ManualResetEventSlim event 268
MemoryBarrier class 291
message 141
message passing interface
 demonstration 142-147
Message Passing Interface (MPI) 141, 142
message queue
 about 147
 implementing 148-155
Microsoft .NET. *See* .NET
Microsoft .NET framework 234
Monitor object
 Monitor::Pulse method 294
 Monitor::Wait method 294
Monitor::Pulse method
 about 294
 signaling with 294-299
Monitor::Wait method
 about 294
 signaling with 294-299
MSDN reference
 URL 9
MSI
 demonstrating 142-147
multitasking 46
multithreading 8

mutex
 about 67, 134, 177
 feature 134
 using 177-187
mutual exclusion 56
MySQL Connector/C
 installing 381-384
MySQL database
 used, for demonstrating
 thread 81-88

N

namespaces 12
NewId method 352
non-blocking synchronization 291-293

O

Object Manager 111
OOP (Object-oriented programming)
 about 17
 example, reviewing 17
 inheritance 19
 working 19
OpenMP (open multiprocessor)
 about 332
 URL 340
OpenThread function 49
OVERLAPPED object 166
overriding feature 24

P

parallel applications
 designing 310-316
parallel approach
 turning on 324-331
parallelism
 adding, to code design 323, 324
 deciding, factors 332
parallel programming 309
PCB
 about 51, 52
 categories 51
PeekMessage API 161

- performance factors**
 - improving 332-336
- permissive multitasking**
 - versus preemptive multitasking 110
- Philosopher application 74**
- pipe client 161**
- pipe object**
 - about 161
 - creating 165
 - used, for process communication 161-166
- pipes**
 - about 216
 - using 216-230
- pipe server 161**
- PLOTDATA object 155**
- polymorphism**
 - about 24
 - performing 24-28
- PostMessage API 155**
- postPhaseAction method delegate 307**
- PostQuitMessage API 33**
- PostThreadMessage API 142**
- preemptive multitasking**
 - versus permissive multitasking 111
- primary thread 8**
- ProblemSolver method 368**
- process and thread priority**
 - usage 170-176
- process communication**
 - pipe object, used for 161-166
- Process Control Block. *See* PCB**
- processes**
 - about 46
 - implementing 52-54
 - versus threads 107-110
- process model**
 - blocked state 50
 - events 47
 - program, creating 48-51
 - ready state 50
 - running state 50
- producer-consumer problem 98**
- program**
 - creating, for Schwefel's function
 - calculation 317-322

- programming paradigm**
 - declarative 10
 - functional(structural) 10
 - imperative 10
 - object-oriented 10
- pseudoparallelism 8, 168**
- pure virtual functions 27**

Q

- quantum 177**
- quasi-parallel address space concept 8**
- queue 34**

R

- race conditions 56**
- Rand function 16**
- ready state, process 50**
- ready state, thread 176**
- release fence 293**
- ReleaseThread method 353**
- RemoteFreeLibrary method 379**
- RemoteLoadLibrary method 379**
- remote threading**
 - about 371
 - using 371-375
 - working 377-379
- RemoveAll method 353**
- RemoveThread method 353, 368**
- request processing 166**
- RequestThread method 353, 368**
- ResetEvent 138**
- Resume method 121**
- running state, process 50**
- running state, thread 176**

S

- Scheduler 46, 51**
- scheduling, operating system 98**
- Schwefel's function 317**
- semaphore**
 - about 135, 136, 187
 - using 187-196
- SendMessage API 155**

sequential programming approach

importance 310

SetEvent API 161

SetUserData method 121

signaling

with Monitor::Pulse method 294-299

with Monitor::Wait method 294-299

Single-Threaded Apartment (STA) 242

Sleep method 247, 248

socket 216

stack 34

Standby state 176

StartAddress method 241, 328

STATE_ALIVE macro 120

STATE_ASYNC macro 120

STATE_BLOCKED macro 120

STATE_CONTINUOUS macro 120

STATE_READY macro 120

STATE_RUNNING macro 120

STATE_SYNC macro 120

static library (*.lib) 14

StillAlive method 352

structural programming approach

about 14

example 15-17

Suspend method 121

synchronization 168

synchronization mechanisms, .NET

about 248

working 253

synchronization objects, Win32

critical section 133

event 133

mutex 133

synchronized threads

using 127-132

szCmdLine parameter 30

T

Task class

reviewing 290

Task::Wait method 247

TCHAR* ObjectName(void) method 323

T_ENDTASK message

defining 161

Terminated state 176

Thread::Abort method 281

thread execution

aborting 281-286

interrupting 281-286

safe canceling 281-290

thread object 8

threading, .NET

about 235-242

parallel tasks, executing 235

working 237-242

Thread::Interrupt method 281

Thread::Join method 247

thread message queue

using 156-161

thread model

implementing 75-80

thread object attributes

base priority 169

dynamic priority 169

thread pool

about 342

dispatcher, customizing 354-366

implementing 342, 343

using 343-350

working 352, 353

ThreadPool class 236

thread pool dispatcher. *See* dispatcher

threads

adding 47

demonstrating, with MySQL database 81-88

implementing, in kernel 99-104

implementing, in user space 89-98

implementing, without

synchronization 121-127

managing 112-121

terminating 106

time control 168

versus processes 107-109

thread safety

about 261

in .NET 261

Thread::Sleep method 247

ThreadStart method 81

time control, threads

priority 168

synchronization 168

T_MESSAGE message
 defining 161
Transfer method 352
Transition state 176
Translation Unit (TU) model 9
typedef, NtQueryInformationProcess
 function 54

U

unmanaged code
 about 234
 versus managed code 234
UpdateWindow function 32
user space
 threads, implementing in 89-98
using command 12

V

Visual Studio project
 setting up, for driver compilation 386-392
 setting up, for OpenMP compilation 393-395
void _tmain(int argc, TCHAR* argv[])
 prototype 12
void _tmain(void) prototype 12
volatile keyword 293

W

WaitAll method 353
Wait Chain Traversal (WCT) 369
WaitForInputIdle function 50

Waiting state 176
Wait method 247
WhoAml method
 overriding 24
Win32
 synchronization objects 133
 thread object 111
Win32 API CreateThread 105
Win32 thread object 111
WinDDK
 installing 384-386
Windows application
 creating 29-34
Windows dispatcher object 176, 341
Windows Driver Development
 Kit. *See* WinDDK
Windows NT HAL 234
Windows synchronization object
 critical section 206
 mutex 177
 semaphore 187
Windows Virtual Memory manager
 mechanism 341
WM_ENDTASK message 154
WM_USER message 154
WndProc 29
work-stealing feature 236



Thank you for buying C++ Multithreading 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.

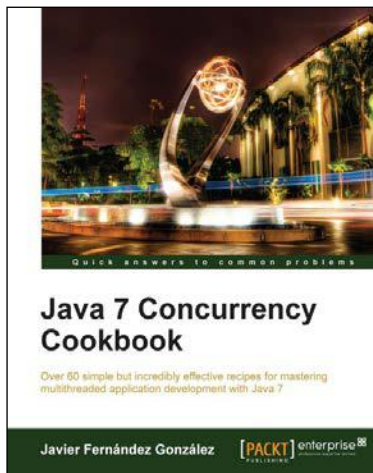
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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.



Java 7 Concurrency Cookbook

ISBN: 978-1-84968-788-1

Paperback: 364 pages

Over 60 simple but incredibly effective recipes for mastering multithreaded application development with Java 7

1. Master all that Java 7 has to offer for concurrent programming.
2. Get to grips with thread management, the Fork/Join framework, concurrency classes, and much more in this book and e-book.
3. A practical Cookbook packed with recipes for achieving the most important Java Concurrency tasks.



Multithreading in C# 5.0 Cookbook

ISBN: 978-1-84969-764-4

Paperback: 268 pages

Over 70 recipes to help you learn asynchronous and parallel programming with C# 5.0 quickly and efficiently

1. Delve deep into the .NET threading infrastructure and use Task Parallel Library for asynchronous programming.
2. Scale out your server applications effectively.
3. Master C# 5.0 asynchronous operations language support.

Please check www.PacktPub.com for information on our titles



C++ Application Development with Code::Blocks

ISBN: 978-1-78328-341-5 Paperback: 128 pages

Develop advanced applications with Code::Blocks quickly and efficiently with this concise, hands-on guide

1. Successfully install and configure Code::Blocks for C++ development.
2. Perform rapid application development with Code::Blocks.
3. Work with advanced C++ features including code debugging and GUI toolkits.



Boost C++ Application Development Cookbook

ISBN: 978-1-84951-488-0 Paperback: 348 pages

Over 80 practical, task-based recipes to create applications using Boost libraries

1. Explores how to write a program once and then use it on Linux, Windows, Mac OS, and Android operating systems.
2. Includes everyday use recipes for multithreading, networking, metaprogramming, and generic programming from a Boost library developer.
3. Take advantage of the real power of Boost and C++ to get a good grounding in using it in any project.

Please check www.PacktPub.com for information on our titles