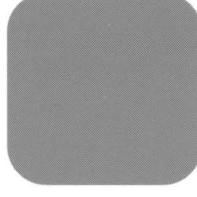
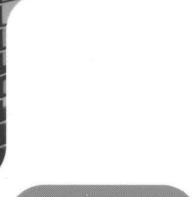
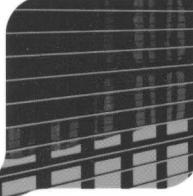
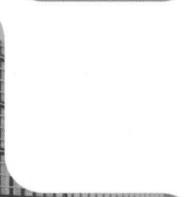
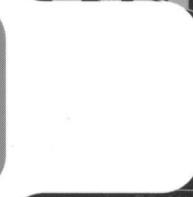
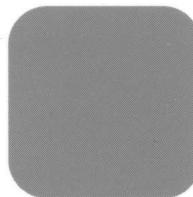
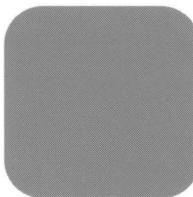
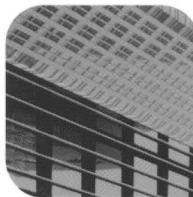


Foreword by Craig Mundie, Chief Research and Strategy Officer, Microsoft



Concurrent Programming on Windows



Joe Duffy

Praise for *Concurrent Programming on Windows*

"I have been fascinated with concurrency ever since I added threading support to the Common Language Runtime a decade ago. That's also where I met Joe, who is a world expert on this topic. These days, concurrency is a first-order concern for practically all developers. Thank goodness for Joe's book. It is a tour de force and I shall rely on it for many years to come."

—*Chris Brumme, Distinguished Engineer, Microsoft*

"I first met Joe when we were both working with the Microsoft CLR team. At that time, we had several discussions about threading and it was apparent that he was as passionate about this subject as I was. Later, Joe transitioned to Microsoft's Parallel Computing Platform team where a lot of his good ideas about threading could come to fruition. Most threading and concurrency books that I have come across contain information that is incorrect and explains how to solve contrived problems that good architecture would never get you into in the first place. Joe's book is one of the very few books that I respect on the matter, and this respect comes from knowing Joe's knowledge, experience, and his ability to explain concepts."

—*Jeffrey Richter, Wintellect*

"There are few areas in computing that are as important, or shrouded in mystery, as concurrency. It's not simple, and Duffy doesn't claim to make it so—but armed with the right information and excellent advice, creating correct and highly scalable systems is at least possible. Every self-respecting Windows developer should read this book."

—*Jonathan Skeet, Software Engineer, Clearswift*

"What I love about this book is that it is both comprehensive in its coverage of concurrency on the Windows platform, as well as very practical in its presentation of techniques immediately applicable to real-world software development. Joe's book is a 'must have' resource for anyone building native or managed code Windows applications that leverage concurrency!"

—*Steve Teixeira, Product Unit Manager,
Parallel Computing Platform, Microsoft Corporation*

“This book is a fabulous compendium of both theoretical knowledge and practical guidance on writing effective concurrent applications. Joe Duffy is not only a preeminent expert in the art of developing parallel applications for Windows, he’s also a true student of the art of writing. For this book, he has combined those two skill sets to create what deserves and is destined to be a long-standing classic in developers’ hands everywhere.”

—Stephen Toub, Program Manager Lead, Parallel Computing Platform, Microsoft

“As chip designers run out of ways to make the individual chip faster, they have moved towards adding parallel compute capacity instead. Consumer PCs with multiple cores are now commonplace. We are at an inflection point where improved performance will no longer come from faster chips but rather from our ability as software developers to exploit concurrency. Understanding the concepts of concurrent programming and how to write concurrent code has therefore become a crucial part of writing successful software. With *Concurrent Programming on Windows*, Joe Duffy has done a great job explaining concurrent concepts from the fundamentals through advanced techniques. The detailed descriptions of algorithms and their interaction with the underlying hardware turn a complicated subject into something very approachable. This book is the perfect companion to have at your side while writing concurrent software for Windows.”

—Jason Zander, General Manager, Visual Studio, Microsoft

Concurrent Programming on Windows

Microsoft .NET Development Series

John Montgomery, *Series Advisor*

Don Box, *Series Advisor*

Brad Abrams, *Series Advisor*

The award-winning Microsoft .NET Development Series was established in 2002 to provide professional developers with the most comprehensive and practical coverage of the latest .NET technologies. It is supported and developed by the leaders and experts of Microsoft development technologies, including Microsoft architects, MVPs, and leading industry luminaries. Books in this series provide a core resource of information and understanding every developer needs to write effective applications.

Titles in the Series

Brad Abrams, *.NET Framework Standard Library*

Annotated Reference Volume 1: Base Class Library and Extended Numerics Library, 978-0-321-15489-7

Brad Abrams and Tamara Abrams, *.NET Framework Standard Library Annotated Reference, Volume 2: Networking Library, Reflection Library, and XML Library*, 978-0-321-19445-9

Chris Anderson, *Essential Windows Presentation Foundation (WPF)*, 978-0-321-37447-9

Bob Beauchemin and Dan Sullivan, *A Developer's Guide to SQL Server 2005*, 978-0-321-38218-4

Adam Calderon, Joel Rumerman, *Advanced ASP.NET AJAX Server Controls: For .NET Framework 3.5*, 978-0-321-51444-8

Eric Carter and Eric Lippert, *Visual Studio Tools for Office: Using C# with Excel, Word, Outlook, and InfoPath*, 978-0-321-33488-6

Eric Carter and Eric Lippert, *Visual Studio Tools for Office: Using Visual Basic 2005 with Excel, Word, Outlook, and InfoPath*, 978-0-321-41175-4

Steve Cook, Gareth Jones, Stuart Kent, Alan Cameron Wills, *Domain-Specific Development with Visual Studio DSL Tools*, 978-0-321-39820-8

Krzysztof Cwalina and Brad Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*, Second Edition, 978-0-321-54561-9

Joe Duffy, *Concurrent Programming on Windows*, 978-0-321-43482-1

Sam Guckenheimer and Juan J. Perez, *Software Engineering with Microsoft Visual Studio Team System*, 978-0-321-27872-2

Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, Peter Golde, *The C# Programming Language*, Third Edition, 978-0-321-56299-9

Alex Homer and Dave Sussman, *ASP.NET 2.0 Illustrated*, 978-0-321-41834-0

Joe Kaplan and Ryan Dunn, *The .NET Developer's Guide to Directory Services Programming*, 978-0-321-35017-6

Mark Michaelis, *Essential C# 3.0: For .NET Framework 3.5*, 978-0-321-53392-0

James S. Miller and Susann Ragsdale,

The Common Language Infrastructure Annotated Standard, 978-0-321-15493-4

Christian Nagel, *Enterprise Services with the .NET Framework: Developing Distributed Business Solutions with .NET Enterprise Services*, 978-0-321-24673-8

Brian Noyes, *Data Binding with Windows Forms 2.0: Programming Smart Client Data Applications with .NET*, 978-0-321-26892-1

Brian Noyes, *Smart Client Deployment with ClickOnce: Deploying Windows Forms Applications with ClickOnce*, 978-0-321-19769-6

Fritz Onion with Keith Brown, *Essential ASP.NET 2.0*, 978-0-321-23770-5

Steve Resnick, Richard Crane, Chris Bowen, *Essential Windows Communication Foundation: For .NET Framework 3.5*, 978-0-321-44006-8

Scott Roberts and Hagen Green, *Designing Forms for Microsoft Office InfoPath and Forms Services 2007*, 978-0-321-41059-7

Neil Roodyn, *eXtreme .NET: Introducing eXtreme Programming Techniques to .NET Developers*, 978-0-321-30363-9

Chris Sells and Michael Weinhardt, *Windows Forms 2.0 Programming*, 978-0-321-26796-2

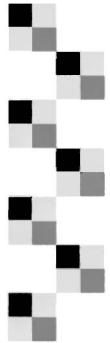
Dharma Shukla and Bob Schmidt, *Essential Windows Workflow Foundation*, 978-0-321-39983-0

Guy Smith-Ferrier, *.NET Internationalization: The Developer's Guide to Building Global Windows and Web Applications*, 978-0-321-34138-9

Will Stott and James Newkirk, *Visual Studio Team System: Better Software Development for Agile Teams*, 978-0-321-41850-0

Paul Yao and David Durant, *.NET Compact Framework Programming with C#*, 978-0-321-17403-1

Paul Yao and David Durant, *.NET Compact Framework Programming with Visual Basic .NET*, 978-0-321-17404-8



Concurrent Programming on Windows

■ Joe Duffy

◆ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The .NET logo is either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries and is used under license from Microsoft.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales

(800) 382-3419

corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales

international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Duffy, Joe, 1980-

Concurrent programming on Windows / Joe Duffy.

p. cm.

Includes bibliographical references and index.

ISBN 978-0-321-43482-1 (pbk. : alk. paper) 1. Parallel programming (Computer science)

2. Electronic data processing—Distributed processing. 3. Multitasking (Computer science)

4. Microsoft Windows (Computer file) I. Title.

QA76.642D84 2008

005.2'75—dc22

2008033911

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc

Rights and Contracts Department

501 Boylston Street, Suite 900

Boston, MA 02116

Fax (617) 671-3447

ISBN-13: 978-0-321-43482-1

ISBN-10: 0-321-43482-X

Text printed in the United States on recycled paper at Edwards Brothers in Ann Arbor, Michigan.

First printing, October 2008

For Mom & Dad



Contents at a Glance

Contents xi

Foreword xix

Preface xxiii

Acknowledgments xxvii

About the Author xxix

PART I Concepts 1

1 Introduction 3

2 Synchronization and Time 13

PART II Mechanisms 77

3 Threads 79

4 Advanced Threads 127

5 Windows Kernel Synchronization 183

6 Data and Control Synchronization 253

7 Thread Pools 315

8 Asynchronous Programming Models 399

9 Fibers 429

PART III Techniques 475

10 Memory Models and Lock Freedom 477

11 Concurrency Hazards 545

- 12 Parallel Containers 613**
- 13 Data and Task Parallelism 657**
- 14 Performance and Scalability 735**

PART IV Systems 783

- 15 Input and Output 785**
- 16 Graphical User Interfaces 829**

PART V Appendices 863

- A Designing Reusable Libraries for Concurrent .NET Programs 865**
- B Parallel Extensions to .NET 887**

Index 931

Contents

<i>Foreword</i>	<i>xix</i>
<i>Preface</i>	<i>xxiii</i>
<i>Acknowledgments</i>	<i>xxvii</i>
<i>About the Author</i>	<i>xxix</i>

PART I Concepts 1

1 Introduction	3
<i>Why Concurrency?</i>	3
Program Architecture and Concurrency	6
Layers of Parallelism	8
<i>Why Not Concurrency?</i>	10
Where Are We?	11
2 Synchronization and Time	13
Managing Program State	14
<i>Identifying Shared vs. Private State</i>	15
<i>State Machines and Time</i>	19
<i>Isolation</i>	31
<i>Immutability</i>	34
Synchronization: Kinds and Techniques	38
<i>Data Synchronization</i>	40
<i>Coordination and Control Synchronization</i>	60
Where Are We?	73

PART II Mechanisms 77**3 Threads 79**

Threading from 10,001 Feet	80
<i>What Is a Windows Thread?</i>	81
<i>What Is a CLR Thread?</i>	85
<i>Explicit Threading and Alternatives</i>	87
The Life and Death of Threads	89
<i>Thread Creation</i>	89
<i>Thread Termination</i>	101
<i>DllMain</i>	115
<i>Thread Local Storage</i>	117
Where Are We?	124

4 Advanced Threads 127

Thread State	127
<i>User-Mode Thread Stacks</i>	127
<i>Internal Data Structures (KTHREAD, ETHREAD, TEB)</i>	145
<i>Contexts</i>	151
Inside Thread Creation and Termination	152
<i>Thread Creation Details</i>	152
<i>Thread Termination Details</i>	153
Thread Scheduling	154
<i>Thread States</i>	155
<i>Priorities</i>	159
<i>Quotients</i>	163
<i>Priority and Quotient Adjustments</i>	164
<i>Sleeping and Yielding</i>	167
<i>Suspension</i>	168
<i>Affinity: Preference for Running on a Particular CPU</i>	170
Where Are We?	180

5 Windows Kernel Synchronization 183

The Basics: Signaling and Waiting	184
<i>Why Use Kernel Objects?</i>	186
<i>Waiting in Native Code</i>	189
<i>Managed Code</i>	204
<i>Asynchronous Procedure Calls (APCs)</i>	208

Using the Kernel Objects	211
<i>Mutex</i>	211
<i>Semaphore</i>	219
<i>A Mutex/Semaphore Example: Blocking/Bounded Queue</i>	224
<i>Auto- and Manual-Reset Events</i>	226
<i>Waitable Timers</i>	234
<i>Signaling an Object and Waiting Atomically</i>	241
<i>Debugging Kernel Objects</i>	250
Where Are We?	251
6 Data and Control Synchronization	253
Mutual Exclusion	255
<i>Win32 Critical Sections</i>	256
<i>CLR Locks</i>	272
Reader/Writer Locks (RWLs)	287
<i>Windows Vista Slim Reader/Writer Lock</i>	289
<i>.NET Framework Slim Reader/Writer Lock (3.5)</i>	293
<i>.NET Framework Legacy Reader/Writer Lock</i>	300
Condition Variables	304
<i>Windows Vista Condition Variables</i>	304
<i>.NET Framework Monitors</i>	309
<i>Guarded Regions</i>	311
Where Are We?	312
7 Thread Pools	315
Thread Pools 101	316
<i>Three Ways: Windows Vista, Windows Legacy, and CLR</i>	317
<i>Common Features</i>	319
Windows Thread Pools	323
<i>Windows Vista Thread Pool</i>	323
<i>Legacy Win32 Thread Pool</i>	353
CLR Thread Pool	364
<i>Work Items</i>	364
<i>I/O Completion Ports</i>	368
<i>Timers</i>	371
<i>Registered Waits</i>	374
<i>Remember (Again): You Don't Own the Threads</i>	377

<i>Thread Pool Thread Management</i>	377
<i>Debugging</i>	386
<i>A Case Study: Layering Priorities and Isolation on Top of the Thread Pool</i>	387
Performance When Using the Thread Pools	391
Where Are We?	398
8 Asynchronous Programming Models 399	
Asynchronous Programming Model (APM)	400
<i>Rendezvousing: Four Ways</i>	403
<i>Implementing IAsyncResult</i>	413
<i>Where the APM Is Used in the .NET Framework</i>	418
<i>ASP.NET Asynchronous Pages</i>	420
Event-Based Asynchronous Pattern	421
<i>The Basics</i>	421
<i>Supporting Cancellation</i>	425
<i>Supporting Progress Reporting and Incremental Results</i>	425
<i>Where the EAP Is Used in the .NET Framework</i>	426
Where Are We?	427
9 Fibers 429	
An Overview of Fibers	430
<i>Upsides and Downsides</i>	431
Using Fibers	435
<i>Creating New Fibers</i>	435
<i>Converting a Thread into a Fiber</i>	438
<i>Determining Whether a Thread Is a Fiber</i>	439
<i>Switching Between Fibers</i>	440
<i>Deleting Fibers</i>	441
<i>An Example of Switching the Current Thread</i>	442
Additional Fiber-Related Topics	445
<i>Fiber Local Storage (FLS)</i>	445
<i>Thread Affinity</i>	447
<i>A Case Study: Fibers and the CLR</i>	449
Building a User-Mode Scheduler	453
<i>The Implementation</i>	455
<i>A Word on Stack vs. Stackless Blocking</i>	472
Where Are We?	473

PART III Techniques 475

10 Memory Models and Lock Freedom 477

Memory Load and Store Reordering 478

What Runs Isn't Always What You Wrote 481

Critical Regions as Fences 484

Data Dependence and Its Impact on Reordering 485

Hardware Atomicity 486

The Atomicity of Ordinary Loads and Stores 487

Interlocked Operations 492

Memory Consistency Models 506

Hardware Memory Models 509

Memory Fences 511

.NET Memory Models 516

Lock Free Programming 518

Examples of Low-Lock Code 520

Lazy Initialization and Double-Checked Locking 520

A Nonblocking Stack and the ABA Problem 534

Dekker's Algorithm Revisited 540

Where Are We? 541

11 Concurrency Hazards 545

Correctness Hazards 546

Data Races 546

Recursion and Reentrancy 555

Locks and Process Shutdown 561

Liveness Hazards 572

Deadlock 572

Missed Wake-Ups (a.k.a. Missed Pulses) 597

Livelocks 601

Lock Convoys 603

Stampeding 605

Two-Step Dance 606

Priority Inversion and Starvation 608

Where Are We? 609

12 Parallel Containers 613

Fine-Grained Locking	616
<i>Arrays</i>	616
<i>FIFO Queue</i>	617
<i>Linked Lists</i>	621
<i>Dictionary (Hashtable)</i>	626
Lock Free	632
<i>General-Purpose Lock Free FIFO Queue</i>	632
<i>Work Stealing Queue</i>	636
Coordination Containers	640
<i>Producer/Consumer Data Structures</i>	641
<i>Phased Computations with Barriers</i>	650
Where Are We?	654

13 Data and Task Parallelism 657

Data Parallelism	659
<i>Loops and Iteration</i>	660
Task Parallelism	684
<i>Fork/Join Parallelism</i>	685
<i>Dataflow Parallelism (Futures and Promises)</i>	689
<i>Recursion</i>	702
<i>Pipelines</i>	709
<i>Search</i>	718
Message-Based Parallelism	719
Cross-Cutting Concerns	720
<i>Concurrent Exceptions</i>	721
<i>Cancellation</i>	729
Where Are We?	732

14 Performance and Scalability 735

Parallel Hardware Architecture	736
<i>SMP, CMP, and HT</i>	736
<i>Superscalar Execution</i>	738
<i>The Memory Hierarchy</i>	739
<i>A Brief Word on Profiling in Visual Studio</i>	754
Speedup: Parallel vs. Sequential Code	756
<i>Deciding to “Go Parallel”</i>	756

<i>Measuring Improvements Due to Parallelism</i>	758
<i>Amdahl's Law</i>	762
<i>Critical Paths and Load Imbalance</i>	764
<i>Garbage Collection and Scalability</i>	766
Spin Waiting	767
<i>How to Properly Spin on Windows</i>	769
<i>A Spin-Only Lock</i>	772
<i>Mellor-Crummey-Scott (MCS) Locks</i>	778
Where Are We?	781

PART IV Systems 783

15 Input and Output 785

Overlapped I/O	786
<i>Overlapped Objects</i>	788
<i>Win32 Asynchronous I/O</i>	792
<i>.NET Framework Asynchronous I/O</i>	817
I/O Cancellation	822
<i>Asynchronous I/O Cancellation for the Current Thread</i>	823
<i>Synchronous I/O Cancellation for Another Thread</i>	824
<i>Asynchronous I/O Cancellation for Any Thread</i>	825
Where Are We?	826

16 Graphical User Interfaces 829

GUI Threading Models	830
<i>Single Threaded Apartments (STAs)</i>	833
<i>Responsiveness: What Is It, Anyway?</i>	836
.NET Asynchronous GUI Features	837
<i>.NET GUI Frameworks</i>	837
<i>Synchronization Contexts</i>	847
<i>Asynchronous Operations</i>	855
<i>A Convenient Package: BackgroundWorker</i>	856
Where Are We?	860

PART V Appendices 863

A Designing Reusable Libraries for Concurrent .NET Programs 865

The 20,000-Foot View	866
-----------------------------	-----

The Details	867
<i>Locking Models</i>	867
<i>Using Locks</i>	870
<i>Reliability</i>	875
<i>Scheduling and Threads</i>	879
<i>Scalability and Performance</i>	881
<i>Blocking</i>	884
B Parallel Extensions to .NET	887
Task Parallel Library	888
<i>Unhandled Exceptions</i>	893
<i>Parents and Children</i>	895
<i>Cancellation</i>	897
<i>Futures</i>	898
<i>Continuations</i>	900
<i>Task Managers</i>	902
<i>Putting it All Together: A Helpful Parallel Class</i>	904
<i>Self-Replicating Tasks</i>	909
Parallel LINQ	910
<i>Buffering and Merging</i>	912
<i>Order Preservation</i>	914
Synchronization Primitives	915
<i>ISupportsCancelation</i>	915
<i>CountdownEvent</i>	915
<i>LazyInit<T></i>	917
<i>ManualResetEventSlim</i>	919
<i>SemaphoreSlim</i>	920
<i>SpinLock</i>	921
<i>SpinWait</i>	923
Concurrent Collections	924
<i>BlockingCollection<T></i>	925
<i>ConcurrentQueue<T></i>	928
<i>ConcurrentStack<T></i>	929
Index	931

Foreword

THE COMPUTER INDUSTRY is once again at a crossroads. Hardware concurrency, in the form of new *manycore processors*, together with growing software complexity, will require that the technology industry fundamentally rethink both the architecture of modern computers and the resulting software development paradigms.

For the past few decades, the computer has progressed comfortably along the path of exponential performance and capacity growth without any fundamental changes in the underlying computation model. Hardware followed Moore's Law, clock rates increased, and software was written to exploit this relentless growth in performance, often ahead of the hardware curve. That symbiotic hardware-software relationship continued unabated until very recently. Moore's Law is still in effect, but gone is the unnamed law that said clock rates would continue to increase commensurately.

The reasons for this change in hardware direction can be summarized by a simple equation, formulated by David Patterson of the University of California at Berkeley:

$$\text{Power Wall} + \text{Memory Wall} + \text{ILP Wall} = \text{A Brick Wall for Serial Performance}$$

Power dissipation in the CPU increases proportionally with clock frequency, imposing a practical limit on clock rates. Today, the ability to dissipate heat has reached a practical physical limit. As a result, a significant

increase in clock speed without heroic (and expensive) cooling (or materials technology breakthroughs) is not possible. This is the “Power Wall” part of the equation. Improvements in memory performance increasingly lag behind gains in processor performance, causing the number of CPU cycles required to access main memory to grow continuously. This is the “Memory Wall.” Finally, hardware engineers have improved the performance of sequential software by speculatively executing instructions before the results of current instructions are known, a technique called instruction level parallelism (ILP). ILP improvements are difficult to forecast, and their complexity raises power consumption. As a result, ILP improvements have also stalled, resulting in the “ILP Wall.”

We have, therefore, arrived at an inflection point. The software ecosystem must evolve to better support manycore systems, and this evolution will take time. To benefit from rapidly improving computer performance and to retain the “write once, run faster on new hardware” paradigm, the programming community must learn to construct concurrent applications. Broader adoption of concurrency will also enable *Software + Services* through asynchrony and loose-coupling, client-side parallelism, and server-side cloud computing.

The Windows and .NET Framework platforms offer rich support for concurrency. This support has evolved over more than a decade, since the introduction of multiprocessor support in Windows NT. Continued improvements in thread scheduling performance, synchronization APIs, and memory hierarchy awareness—particularly those added in Windows Vista—make Windows the operating system of choice for maximizing the use of hardware concurrency. This book covers all of these areas. When you begin using multithreading throughout an application, the importance of clean architecture and design is critical to reducing software complexity and improving maintainability. This places an emphasis on understanding not only the platform’s capabilities but also emerging best practices. Joe does a great job interspersing best practice alongside mechanism throughout this book.

Manycore provides improved performance for the kinds of applications we already create. But it also offers an opportunity to think completely differently about what computers should be able to do for people. The

continued increase in compute power will qualitatively change the applications that we can create in ways that make them a lot more interesting and helpful to people, and able to do new things that have never been possible in the past. Through this evolution, software will enable more personalized and humanistic ways for us to interact with computers. So enjoy this book. It offers a lot of great information that will guide you as you take your first steps toward writing concurrent, manycore aware software on the Windows platform.

Craig Mundie

Chief Research and Strategy Officer

Microsoft Corporation

June 2008

Preface

I BEGAN WRITING this book toward the end of 2005. At the time, dual-core processors were becoming standard on the mainstream PCs that ordinary (nonprogrammer) consumers were buying, and a small number of people in industry had begun to make noise about the impending concurrency problem. (Herb Sutter's, *The Free Lunch is Over*, paper immediately comes to mind.) The problem people were worried about, of course, was that the software of the past was not written in a way that would allow it to naturally exploit that additional compute power. Contrast that with the never-ending increase in clock speeds. No more free lunch, indeed.

It seemed to me that concurrency was going to eventually be an important part of every software developer's job and that a book such as this one would be important and useful. Just two years later, the impact is beginning to ripple up from the OS, through the libraries, and on up to applications themselves.

This was about the same time I had just wrapped up prototyping a small side project I had been working on for six months, called Parallel Language Integrated Query (PLINQ). The PLINQ project was a conduit for me to explore the intricacies of concurrency, multicore, and specifically how parallelism might be used in real-world, everyday programs. I used it as a tool to figure out where the platform was lacking. This was in addition to spending my day job at Microsoft focused on software transactional memory (STM), a technology that in the intervening two years has become somewhat of an industry buzzword. Needless to say, I had become pretty

entrenched in all topics concurrency. What better way to get entrenched even further than to write a book on the subject?

As I worked on all of these projects, and eventually PLINQ grew into the Parallel Extensions to the .NET Framework technology, I was amazed at how few good books on Windows concurrency were available. I remember time and time again being astonished or amazed at some intricate and esoteric bit of concurrency-related information, jotting it down, and earmarking it for inclusion in this book. I only wished somebody had written it down before me, so that I didn't need to scour it from numerous sources: hallway conversations, long nights of pouring over Windows and CLR source code, and reading and rereading countless Microsoft employee blogs. But the best books on the topic dated back to the early '90s and, while still really good, focused too much on the mechanics and not enough on how to structure parallel programs, implement parallel algorithms, deal with concurrency hazards, and all those important concepts. Everything else targeted academics and researchers, rather than application, system, and library developers.

I set out to write a book that I'd have found fascinating and a useful way to shortcut all of the random bits of information I had to learn throughout. Although it took me a surprisingly long two-and-a-half years to finish this book, the state of the art has evolved slowly, and the state of good books on the topic hasn't changed much either. The result of my efforts, I hope, is a new book that is down to earth and useful, but still full of very deep technical information. It is for any Windows or .NET developer who believes that concurrency is going to be a fundamental requirement of all software somewhere down the road, as all industry trends seem to imply.

I look forward to kicking back and enjoying this book. And I sincerely hope you do too.

Book Structure

I've structured the book into four major sections. The first, Concepts, introduces concurrency at a high level without going too deep into any one topic. The next section, Mechanisms, focuses squarely on the fundamental platform features, inner workings, and API details. After that, the Techniques

section describes common patterns, best practices, algorithms, and data structures that emerge while writing concurrent software. The fourth section, Systems, covers many of the system-wide architectural and process concerns that frequently arise. There is a progression here. Concepts is first because it develops a basic understanding of concurrency in general. Understanding the content in Techniques would be difficult without a solid understanding of the Mechanisms, and similarly, building real Systems would be impossible without understanding the rest. There are also two appendices at the end.

Code Requirements

To run code found in this book, you'll need to download some free pieces of software.

- **Microsoft Windows SDK.** This includes the Microsoft C++ compiler and relevant platform headers and libraries. The latest versions as of this writing are the Windows Vista and Server 2008 SDKs.
- **Microsoft .NET Framework SDK.** This includes the Microsoft C# and Visual Basic compilers, and relevant framework libraries. The latest version as of this writing is the .NET Framework 3.5 SDK.

Both can be found on MSDN: <http://msdn.microsoft.com>.

In addition, it's highly recommended that you consider using Visual Studio. This is not required—and in fact, much of the code in this book was written in emacs—but provides for a more seamless development and debugging experience. Visual Studio 2008 Express Edition can be downloaded for free, although it lacks many useful capabilities such as performance profiling.

Finally, the debugging tools for Windows package, which includes the popular WINDBG debugging utility—can also come in handy, particularly if you don't have Visual Studio. It is freely downloadable from <http://www.microsoft.com>. Similarly, the Sysinternals utilities available from <http://technet.microsoft.com/sysinternals> are quite useful for inspecting aspects of the Windows OS.

A companion website is available at:
<http://www.bluebytesoftware.com/books>

Joe Duffy
June 2008

joe@bluebytesoftware.com
http://www.bluebytesoftware.com

Acknowledgments

MANY PEOPLE HAVE helped with the creation of this book, both directly and indirectly.

First, I have to sincerely thank Chris Brumme and Jan Gray for inspiring me to get the concurrency bug several years ago. You've both been incredibly supportive and have helped me at every turn in the road. This has led to not only this book but a never-ending stream of career, technical, and personal growth opportunities. I'm still not sure how I'll ever repay you guys.

Also, thanks to Herb Sutter, who was instrumental in getting this book's contract in the first place. And also to Craig Mundie for writing a terrific Foreword and, of course, leading Microsoft and the industry as a whole into our manycore future.

Vance Morrison deserves special thanks for not only being a great mentor along the way, but also for being the toughest technical reviewer I've ever had. His feedback pushed me really hard to keep things concise and relevant. I haven't even come close to attaining his vision of what this book could have been, but I hope I'm not too far afield from it.

Next, in alphabetical order, many people helped by reviewing the book, discussing ideas along the way, or answering questions about how things work (or were supposed to work): David Callahan, Neill Clift, Dave Detlefs, Yves Dolce, Patrick Dussud, Can Erten, Eric Eilebrecht, Ed Essey, Kang Su Gatlin, Goetz Graefe, Kim Greenlee, Vinod Grover, Brian Grunkemeyer, Niklas Gustafsson, Tim Harris, Anders Hejlsberg, Jim Larus, Eric Li, Weiwen Liu, Mike Magruder, Jim Miller, Igor Ostrovsky,

Joel Pobar, Jeff Richter, Paul Ringseth, Burton Smith, Stephen Toub, Roger Wolff, and Keith Yedlin. For those reviewers who were constantly promised drafts of chapters that never actually materialized on time, well, I sincerely appreciate the patience.

Infinite thanks also go out to the staff from Addison-Wesley. In particular, I'd like to give a big thanks to Joan Murray. You've been the only constant throughout the whole project and have to be the most patient person I've ever worked with. When I originally said the book would only take eight months, I wasn't lying intentionally. Hey, a 22-month underestimate isn't too bad, right? Only a true software developer would say that.

About the Author

Joe Duffy is the development lead, architect, and founder of the Parallel Extensions to the .NET Framework team at Microsoft, in the Visual Studio division. In addition to hacking code and managing a team of amazing developers, he defines the team's long-term vision and strategy. His current interests include functional programming, first-class concurrency safety in the type system and creating programming models that will enable everyday people to exploit GPUs and SIMD style processors. Joe had previous positions at Microsoft as the developer for Parallel LINQ (PLINQ) and the concurrency program manager for the Common Language Runtime (CLR). Before joining Microsoft, he had seven years of professional programming experience, including four years at EMC. He was born in Massachusetts, and currently lives in Washington. While not indulging in technical excursions, Joe spends his time playing guitar, studying music theory, listening to and writing music, and feeding his wine obsession.

PART I

Concepts

1

Introduction

CONCURRENCY IS EVERYWHERE. No matter whether you're doing server-side programming for the web or cloud computing, building a responsive graphical user interface, or creating a new interactive client application that uses parallelism to attain better performance, concurrency is ever present. Learning how to deal with concurrency when it surfaces and how to exploit it to deliver more capable and scalable software is necessary for a large category of software developers and is the main focus of this book.

Before jumping straight into the technical details of how to use concurrency when developing software, we'll begin with a conceptual overview of concurrency, some of the reasons it can be important to particular kinds of software, the role it plays in software architecture, and how concurrency will fit progressively into layers of software in the future.

Everything in this chapter, and indeed most of the content in this book, applies equally to programs written in native C++ as it does to programs written in the .NET Framework.

Why Concurrency?

There are many reasons why concurrency may be interesting to you.

- You are programming in an environment where concurrency is already pervasive. This is common in real-time systems,

OS programming, and server-side programming. It is the reason, for example, that most database programmers must become deeply familiar with the notion of a transaction before they can truly be effective at their jobs.

- You need to maintain a responsive user interface (UI) while performing some compute- or I/O-intensive activity in response to some user input. In such cases, running this work on the UI thread will lead to poor responsiveness and frustrated end users. Instead, concurrency can be used to move work elsewhere, dramatically improving the responsiveness and user experience.
- You'd like to exploit the asynchrony that already exists in the relationship between the CPU running your program and other hardware devices. (They are, after all, separately operating and independent pieces of hardware.) Windows and many device drivers cooperate to ensure that large I/O latencies do not severely impact program performance. Using these capabilities requires that you rewrite code to deal with concurrent orchestration of events.
- Some problems are more naturally modeled using concurrency. Games, AI, and scientific simulations often need to model interactions among many agents that operate mostly independently of one another, much like objects in the real world. These interactions are inherently concurrent. Stream processing of real-time data feeds, where the data is being generated in the physical world, typically requires the use of concurrency. Telephony switches are inherently massively concurrent, leading to special purpose languages, such as Erlang, that deal specifically with concurrency as a first class concept.
- You'd like to utilize the processing power made available by multiprocessor architectures, such as multicore, which requires a form of concurrency called parallelism to be used. This requires individual operations to be decomposed into independent parts that can run on separate processors.

In summary, many problem domains are ripe with inherent concurrency. If you're building a server application, for example, many requests

may arrive concurrently via the network and must be dealt with simultaneously. If you're writing a Web request handler and need to access shared state, concurrency is suddenly thrust to the forefront.

While it's true that concurrency can *sometimes* help express problems more naturally, this is rare in practice. Human beings tend to have a difficult time reasoning about large amounts of asynchrony due to the combinatorial explosion of possible interactions. Nevertheless, it is becoming increasingly more common to use concurrency in instances where it feels unnatural. The reason for this is that microprocessor architecture has fundamentally changed; parallel processors are now widespread on all sorts of mainstream computers. Multicore has already pervaded the PC and mobile markets, and highly parallel graphics processing units (GPUs) are everywhere and sometimes used for general purpose computing. In order to fully maximize use of these newer generation processors, programs must be written in a **naturally scalable** manner. That means applications must contain sufficient *latent concurrency* so that, as newer machines are adopted, program performance automatically improves alongside by realizing that latent concurrency as *actual concurrency*.

In fact, although many of us program in a mostly sequential manner, our code often has a lot of inherent latent concurrency already by virtue of the way operations have been described in our language of choice. Data and control dependence among loops, if-branches, and memory moves can constrain this, but, in a surprisingly large number of cases, these are artificial constraints that are placed on code out of stylistic habit common to C-style programming.

This shift is a change from the past, particularly for client-side programs. **Parallelism** is the use of concurrency to decompose an operation into finer grained constituent parts so that independent parts can run on separate processors on the target machine. This idea is not new. Parallelism has been used in scientific computing and supercomputing for decades as a way to scale across tens, hundreds, and, in some cases, thousands of processors. But mainstream commercial and Web software generally has been authored with sequential techniques based on the assumption that clock speed will increase 40 to 50 percent year over year, indefinitely, and that corresponding improvements in performance would follow "for free."

Program Architecture and Concurrency

Concurrency begins with architecture. It is also possible to retrofit concurrency into an existing application, but the number of common pitfalls is vastly decreased with careful planning. The following taxonomy is a useful way to think about the structure of concurrent programs, which will help during the initial planning and architecture phases of your project:

- **Agents.** Most programs are already coarsely decomposed into independent agents. An agent in this context is a very abstract term, but the key attributes are: (1) state is mostly *isolated* within it from the outset, (2) its interactions with the world around it are *asynchronous*, and (3) it is generally loosely coupled with respect to peer agents. There are many manifestations of agents in real-world systems, ranging from individual Web requests, a Windows Communication Foundation (WCF) service request, COM component call, some asynchronous activity a program has farmed off onto another thread, and so forth. Moreover, some programs have just one agent: the program's entry point.
- **Tasks.** Individual agents often need to perform a set of operations at once. We'll call these tasks. Although a task shares many ideas with agents—such as being asynchronous and somewhat independent—tasks are unique in that they typically share state intimately. Many sequential client-side programs fail to recognize tasks are first class concepts, but doing so will become increasingly important as fine-grained parallelism is necessary for multicore. Many server-side programs also do not have a concept of tasks, because they already use large numbers of agents in order to expose enough latent concurrency to utilize the hardware. This is OK so long as the number of active agents exceeds the number of available processors; as processor counts and the workloads a single agent is responsible for grow, this can become increasingly difficult to ensure.
- **Data.** Operations on data are often naturally parallel, so long as they are programmed such that the system is made available of latent concurrency. This is called **data parallelism**. Such operations might

include transformations of data in one format into another, business intelligence analysis, encryption, compression, sorting, searching data for elements with certain characteristics, summarizing data for reporting purposes, rendering images, etc. The more data there is, the more compute- and time-intensive these operations are. They are typically leaf level, very fine grained, and, if expressed properly, help to ensure future scaling. Many programs spend a large portion of their execution time working with data; thus, these operations are likely to grow in size and complexity as a program's requirements and data input evolves over time.

This taxonomy forms a nice hierarchy of concurrency, shown in Figure 1.1. While it's true that the clean hierarchy must be strictly broken in some cases (e.g., a data parallel task may need to communicate with an agent), a clean separation is a worthy goal.

State isolation also is crucial to think about while architecting concurrent programs. For example, it is imperative to strive for designs that lead to agents having state entirely isolated from one another such that they can remain loosely coupled and to ease the *synchronization* burden. As finer grained concurrency is used, state is often shared, but functional concepts

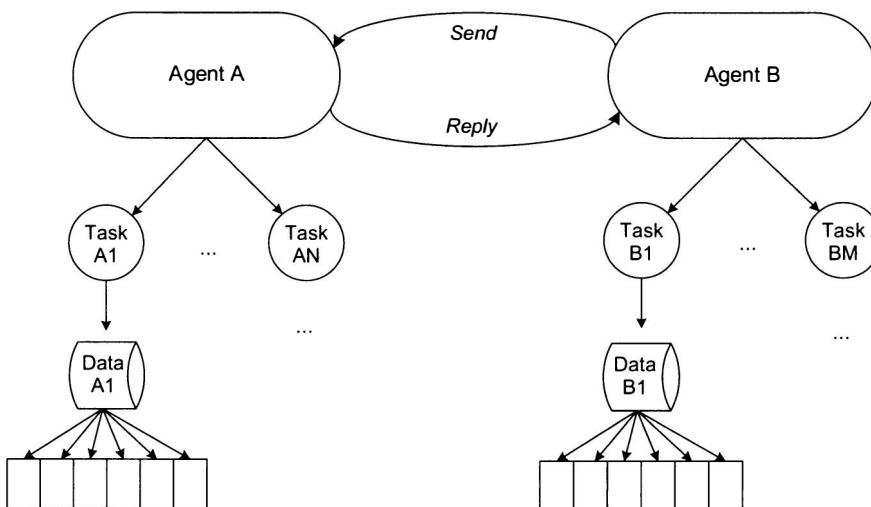


FIGURE 1.1: A taxonomy of concurrent program structure

such as *immutability* and *purity* become important: these disciplines help to eliminate concurrency bugs that can be extraordinarily difficult to track down and fix later. The topics of state and synchronization are discussed at length in Chapter 2, Synchronization and Time.

What you'll find as you read the subsequent chapters in this book is that these terms and concepts are merely guidelines on how to create structured architecture in your program, rather than being concrete technologies that you will find in Windows and the .NET Framework. Several examples of agents were already given, and both task and data parallelism may take one of many forms today. These ideas often map to work items executed in dedicated threads or a *thread pool* (see Chapter 7, Thread Pools), but this varies from one program to the next.

Layers of Parallelism

It is not the case that all programs can be highly parallel, nor is it the case that this should be a goal of most software developers. At least over the next half decade, much of multicore's success will undoubtedly be in the realm of *embarrassingly parallel* problems, where real parallel hardware is used to attain impressive speedups. These are the kinds of problems where parallelism is inherent and easily exploitable, such as compute-intensive image manipulation, financial analysis, and AI algorithms. Because parallelism is more natural in these domains, there is often less friction in getting code correct and performing well. Race conditions and other concurrency hazards are simply easier to avoid with these kinds of programs, and, when it comes to observing a parallel speedup, the ratio of success to failure is far higher.

Other compute-intensive kernels of computations will use parallelism but will require more effort. For example, math libraries, sort routines, report generation, XML manipulation, and stream processing algorithms may all use parallelism to speed up result generation. In addition, domain specific languages (DSLs) may arise that are inherently parallel. C#'s Language Integrated Query (LINQ) is one example of an embedded DSL within an otherwise imperative language, and MATLAB is yet another. Both are amenable to parallel execution. As libraries adopt parallelism, those programs that use them will receive some amount of scalability for

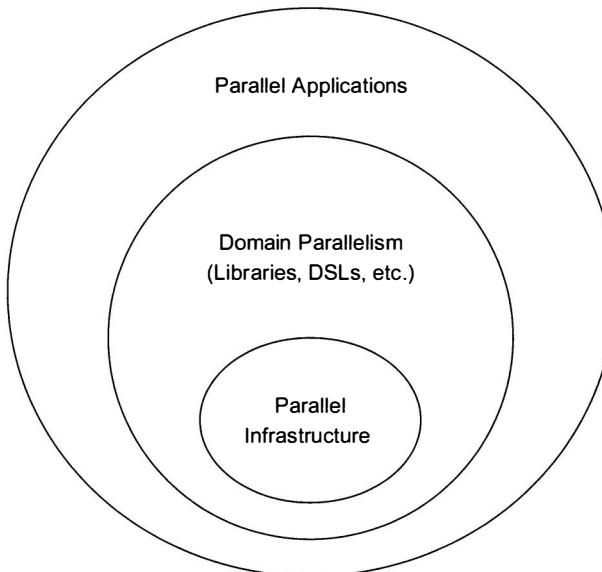


FIGURE 1.2: A taxonomy of concurrent program structure

free, particularly if a large portion of time is spent executing that library code. This is attractive because the parallelism is reusable in a variety of contexts.

The resulting landscape of parallelism is visualized in Figure 1.2. If you stop to think about it, this picture is not very different from what we are accustomed to seeing for sequential software. Software developers creating libraries focus on ensuring that their performance meets customer expectations, and they spend a fair bit of time on optimization and enabling future scalability. Parallelism is similar; the techniques used are different, but the primary motivating factor—that of improving performance—is shared among them.

Aside from embarrassingly parallel algorithms and libraries, some applications will still use concurrency specifically. Many of these use cases will be in representing coarse-grained independent operations as agents. In fact, many programs already are structured this way; utilizing the benefits of multicore in these cases often requires minimal restructuring, although the scalability tends to be fixed to a small number of agents and, hence, cores. Most developers of mostly sequential applications also can use

profilers (such as the one in Visual Studio) to identify CPU-bound hotspots in programs to identify opportunities for fine-grained parallelism.

Why Not Concurrency?

Concurrency is not for everyone. The fact that a whole book has been written about concurrency alone should tell you that it's a somewhat dense topic. It is relatively easy to get started with concurrency—thanks to the fact that creating threads, queuing work to thread pools, and the like, are all very simple (and indeed automated by some commonly used programming models such as ASP.NET)—but there are many subtle consequences.

Concurrency is a fundamental cross-cutting property of software. Once you've got many threads actively calling into a shared data structure that you've written, for example, the number of concerns you must have considered and proactively safeguarded yourself against when writing that data structure is often daunting. Indeed it will often only be evident after you've been programming with concurrency for a while or until you've read a book about it.

Here is a quick list of some examples of such problems. Chapter 2, Synchronization and Time, and later, Chapter 11, Concurrency Hazards, will provide more detail on each.

- State management decisions, as noted above, often lead to synchronization. Most often this means some form of **locking**. Locking is difficult to get right and can have a negative impact on performance. Verifying that you've implemented some locking policy correctly tends to be vastly more difficult than typical unit-test-style verification. And getting it wrong will lead to **race conditions**, which are bugs that depend on intricate timing and machine architecture and are very difficult to reproduce.
- **Deadlock** can arise when synchronization is used, leading to a program that suddenly stops making progress indefinitely. The result of this can range anywhere from annoying (e.g., a hung user interface) to disastrous (e.g., a semi-real-time system fails to respond to a

critical event in time). When optimistic concurrency is used, a similar phenomenon, *livelock*, can occur.

- Data structure invariants are significantly more important to reason about and solidify when concurrency is involved. **Reentrancy** can break them and so, too, can incorrect synchronization granularity. A common source of the latter problem is releasing a lock before invariants have been restored. Yet at the same time, our current languages and tools do not encourage any kind of invariant capture or verification, complicating the task of ensuring correctness.
- The current generation of tools—including Visual Studio 2008 and Debugging Tools for Windows—do not tailor the debugging experience to concurrency. Thus debugging all of the above mentioned problems tends to be more of a black art than a science and requires deep knowledge of OS and threading internals.

Concurrency is a double-edged sword. It can be used to do amazing new things and to enable new compute-intensive experiences that will only become possible with the amount of computing power available in the next generation of microprocessor architecture. And in some situations concurrency is unavoidable. But it must also be used responsibly so as not to negatively impact software robustness and reliability. This book’s aim is to help you decide when it is appropriate, in what ways it is appropriate, and, once you’ve answered those questions for your situation, to aid you in developing, testing, and maintaining concurrent software.

Where Are We?

This introductory chapter painted a high-level picture of concurrency’s place in modern software. We began by explaining why you might be interested in using concurrency and then moved on to a couple brief explorations of taxonomies that can be useful in organizing your thoughts and structuring your programs. Sadly, we haven’t seen any code yet! The next chapter, and all of the remaining ones, will change that by focusing on specifics and details.

FURTHER READING

- K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yellick. *The Landscape of Parallel Computing Research: A View from Berkeley*, EECS Technical Report EECS-2006-183 (University of California, 2006).
- J. Larus, H. Sutter. Software and the Concurrency Revolution. *ACM Queue*, Vol. 3, No. 7 (2005).
- J. Larus. Spending Moore's Dividend. *Microsoft Technical Report*, MSR-TR-2008-69 (May 2008).
- H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3) (2005).

2

Synchronization and Time

STATE IS AN important part of any computer system. This point seems so obvious that it sounds silly to say it explicitly. But state within even a single computer program is seldom a simple thing, and, in fact, is often scattered throughout the program, involving complex interrelationships and different components responsible for managing state transitions, persistence, and so on. Some of this state may reside inside a process's memory—whether that means memory allocated dynamically in the heap (e.g., objects) or on thread stacks—as well as files on-disk, data stored remotely in database systems, spread across one or more remote systems accessed over a network, and so on. The relationships between related parts may be protected by transactions, handcrafted semitransactional systems, or nothing at all.

The broad problems associated with state management, such as keeping all sources of state in-synch, and architecting consistency and recoverability plans all grow in complexity as the system itself grows and are all traditionally very tricky problems. If one part of the system fails, either state must have been protected so as to avoid corruption entirely (which is generally not possible) or some means of recovering from a known safe point must be put into place.

While state management is primarily outside of the scope of this book, state “in-the-small” is fundamental to building concurrent programs. Most Windows systems are built with a strong dependency on *shared memory* due to the way in which many threads inside a process share access to the same

virtual memory address space. The introduction of concurrent access to such state introduces some tough challenges. With concurrency, many parts of the program may simultaneously try to read or write to the same shared memory locations, which, if left uncontrolled, will quickly wreak havoc. This is due to a fundamental concurrency problem called a *data race* or often just *race condition*. Because such things manifest only during certain interactions between concurrent parts of the system, it's all too easy to be given a false sense of security—that the possibility of havoc does not exist.

In this chapter, we'll take a look at state and synchronization at a fairly high level. We'll review the three general approaches to managing state in a concurrent system:

1. **Isolation**, ensuring each concurrent part of the system has its own copy of state.
2. **Immutability**, meaning that shared state is read-only and never modified, and
3. **Synchronization**, which ensures multiple concurrent parts that wish to access the same shared state simultaneously cooperate to do so in a safe way.

We won't explore the real mechanisms offered by Windows and the .NET Framework yet. The aim is to understand the fundamental principles first, leaving many important details for subsequent chapters, though pseudo-code will be used often for illustration.

We also will look at the relationship between state, control flow, and the impact on coordination among concurrent threads in this chapter. This brings about a different kind of synchronization that helps to coordinate state dependencies between threads. This usually requires some form of waiting and notification. We use the term **control synchronization** to differentiate this from the kind of synchronization described above, which we will term **data synchronization**.

Managing Program State

Before discussing the three techniques mentioned above, let's first be very precise about what the terminology **shared state** means. In short, it's any

state that is accessible by more than one thread at a time. It's surprisingly difficult to pin down more precisely, and the programming languages commonly in use on the platform are not of help.

Identifying Shared vs. Private State

In object oriented systems, state in the system is primarily instance and static (a.k.a. class) fields. In procedural systems, or in languages like C++ that support a mixture of object oriented and procedural constructs, state is also held in global variables. In thread based programming systems, state may also take the form of local variables and arguments on thread stacks used during the execution and invocation of functions. There are also several other subtle sources of state distributed throughout many layers in the overall infrastructure: code, DLLs, thread local storage (TLS), runtime and OS resources, and even state that spans multiple processes (such as memory mapped files and even many OS resources).

Now the question is "What constitutes 'shared state' versus 'private state'?" The answer depends on the precise mechanisms you are using to introduce concurrency into the system. Stated generally, shared state is any state that may, at any point in time, be accessed by multiple threads concurrently. In the systems we care about, that means:

- All state pointed to by a global or static field is shared.
- Any state passed during thread creation (from creator to createe) is shared.
- Any state reachable through references in said state is also shared, transitively.

As a programmer, it's important to be very conscious of these points, particularly the last. The transitive nature of sharing and the fact that, given any arbitrary pointer, you cannot tell whether the state it refers to has been shared or not, cause tremendous difficulty in building concurrent systems on Windows. Once something becomes shared, it can be difficult to track its ownership in the system, particularly to determine precisely at what point it becomes shared and at what point it becomes unshared in the future (if at all). These can be referred to as **data publication** and **privatization**,

respectively. Certain programming patterns such as producer/consumer use consistent sharing and transfer of ownership patterns, making the points of publication and privatization more apparent. Even then it's easy to trip up and make a mistake, such as treating something private although it is still shared, causing race conditions.

It's also important to note that the above definitions depend to some degree on modern type safety. In the .NET Framework this is generally not negotiable, whereas in systems like C++ it is highly encouraged but can be circumvented. When any part of the program can manufacture a pointer to any arbitrary address in the process's address space, all data in the entire address space is shared state. We will ignore this loophole. But when pointer arithmetic is involved in your system, know that many of the same problems we'll look at in this chapter can manifest. They can be even more nondeterministic and hard to debug, however.

To illustrate some of the challenges in identifying shared state, here's a class definition in C++. It has one simple method, `f`, and two fields, one static (`s_f`) and the other instance (`m_f`). Despite the use of C++ here, the same principles clearly apply to managed code too.

```
class C
{
    static int s_f;
    int m_f;
public:
    void f(int * py)
    {
        int x;
        x++;      // local variable
        s_f++;   // static class member
        m_f++;   // class member
        (*py)++; // pointer to something
    }
};
```

The method contains four read/increment/write operations (via C++'s `++` unary operator). In a concurrent system, it is possible that multiple threads could be invoking `f` on the same instance of `C` concurrently with one another. Some of these increments will be safe to perform while others are not. Others still might only be safe if `f` is called in certain ways. We'll see many detailed examples of what can go wrong with this example. Simply

put, any increments of shared data are problematic. This is not strictly true because higher level programming conventions and constructs may actually prevent problematic shared interactions, but given the information above, we have no choice but to assume the worst.

By simply looking at the class definition above, how do we determine what state is shared? Unfortunately we can't. We need more information. The answer to this question depends on how instances of C are used in addition to where the py pointer came from.

We can quickly label the operations that do not act on shared state because there are so few (just one). The only memory location not shared with other threads is the x variable, so the x++ statement doesn't modify shared state. (Similar to the statement above about type safety, we are relying on the fact that we haven't previously shared the address of x on the thread's stack with another thread. Of course, another thread might have found an address to the stack through some other means and could perform address arithmetic to access x indirectly, but this is a remote possibility. Again, we will assume some reasonable degree of type safety.) Though it doesn't appear in this example, if there was a statement to increment the value of py, i.e., py++, it would not affect shared state because py is passed by value.

The s_f++ statement affects shared state because, by the definition of static variables, the class's static memory is visible to multiple threads running at once. Had we used a static local variable in f in the above example, it would fall into this category too.

Here's where it becomes complicated. The m_f++ line might, at first glance, appear to act on private memory, but we don't have enough information to know. Whether it modifies shared state or not depends on if the caller has shared the instance of C across multiple threads (or itself received the pointer from a caller that has shared the instance). Remember, m_f++ is a pointer dereference internally, (this->m_f)++. The this pointer might refer to an object allocated on the current thread's stack or allocated dynamically on the heap and may or may not be shared among threads in either case.

```
class D
{
    static C s_c; // initialized elsewhere...
    C m_c;        // also initialized elsewhere...
```

```
void g()
{
    int x = 0;

    C c1(); // stack-alloc
    c1.f(&x);

    C c2 = new C(); // heap-alloc
    c2.f(&x);
    s_c.f(&x);
    m_c.f(&x);
}
```

In the case of the `c1.f(&x)` function call, the object is private because it was allocated on the stack. Similarly, with `c2.f(&x)` the object is probably private because, although allocated on the heap, the instance is not shared with other threads. (Neither case is simple: C's constructor could publish a reference to itself to a shared location, making the object shared before the call to f happens.) When called through `s_c`, clearly the object is shared because it is stored in a shared static variable. And the answer for the call through `m_c` is "it depends." What does it depend on? It depends on the allocation of the instance of D through which g has been invoked. Is it referred to by a static variable elsewhere, another shared object, and so forth? This illustrates how quickly the process of identifying shared state is transitive and often depends on complex, dynamically composed object graphs.

Because the member variable and explicit pointer dereference are similar in nature, you can probably guess why "it depends" for `(*py)++` too. The caller of f might be passing a pointer to a private or shared piece of memory. We really have no way of telling.

Determining all of this statically is impossible without some form of type system support (which is not offered by VC++ or any mainstream .NET languages). The process of calculating the set of shared objects dynamically also is even difficult but possible. The process can be modeled much in the same way garbage collection works: by defining the set of shared roots as those objects referenced directly by static variables, we could then traverse the entire reachable set of objects beginning with only those roots, marking all objects as we encounter them (avoiding cycles). At the end, we know that all marked objects are shared. But this approach is

too naïve. An object can also become shared at thread creation time by passing a pointer to it as an argument to thread creation routines. The same goes for thread pool APIs, among others. Some objects are special, such as the one global shared `OutOfMemoryException` object that the CLR throws when memory is very low. Some degree of compiler analysis could help. A technique called escape analysis determines when private memory “escapes” into the shared memory space, but its application is limited mostly to academic papers (see Further Reading, Choi, Gupta, Serrano, Sreedhar, Midkiff). In practice, complications, such as late bound method calls, pointer aliasing, and hidden sources of cross-thread sharing, make static analysis generally infeasible and subject to false negatives without restrictions in the programming model. There is research exploring such ideas, such as ownership types, but it is probably years from mainstream use (see Further Reading, Boyapati, Liskov, Shrira).

In the end, logically separating memory that is shared from memory that is private is of utmost importance. This is perhaps the most fundamental and crucial skill to develop when building concurrent systems in modern programming environments: accurately identifying and properly managing shared state. And, more often than not, shared state must be managed carefully and with a great eye for detail. This is also why understanding and debugging concurrent code that someone else wrote is often very difficult.

State Machines and Time

All programs are state machines. Not all people think of their programs this way, but it turns out to be a convenient mental model for concurrent programs. Even if you don’t think about your program as a state machine proper, you probably at least think about your program in terms of time and the sequence of program events on a sequential timeline: the order in which reads from and writes to variables occur, the time distance between two such events, and so on. A unique problem with concurrency thus arises. We are accustomed to reasoning about the code we write on the screen in sequential order, which is necessarily written in a sequential layout. We form mental models and conclusions about the state transitions possible with these assumptions firmly in mind. However, concurrency invalidates many such assumptions.

When state is shared, multiple concurrent threads, each of which may have been constructed with a set of sequential execution assumptions, may end up overlapping in time. And when they overlap in time, their operations become interleaved. If these operations access common memory locations, they may possibly violate the legal set of state transitions that the program's state machine was planned and written to accommodate. Once this happens, the program may veer wildly off course, doing strange and inexplicable things that the author never intended, including performing bogus operations, corrupting memory, or crashing.

Broken Invariants and Invalid States

As an illustration, let's say on your first day at a new programming job you were assigned the task of implementing a reusable, dynamically resizing queue data structure. You'd probably start out with a sketch of the algorithms and outline some storage alternatives. You'd end up with some fields and methods and some basic decisions having been made, perhaps such as using an array to store elements versus a linked list. If you're really methodical, you might write down the state invariants and transitions and write them down as asserts in the code or even use a formal specification system to capture (and later verify) them. But even if you didn't go to these lengths, those invariants still exist. Break any one of them during development, or worse after code has been embedded into a system, and you've got a bug.

Let's consider a really simple invariant. The count of the queue must be less than or equal to the length of the array used to store the individual elements. (There are of course several others: the head and tail indices must be within the legal range, and so on.) If this queue was meant only to be used by sequential programs, then preserving the invariant at the entrance and exit of all public methods would be sufficient as a correctness condition. It would be trivial: only those methods that modify the fields need to be written to carefully respect the invariant. The most difficult aspect of attaining this would be dealing with failures, such as an inability to allocate memory when needed.

Things become much more difficult as soon as concurrency is added to the system. Unless another approach is used, you would have to ensure invariants held at every single line of code in your implementation. And

even that might not be sufficient if some lines of code (in whatever higher level language you are programming in) were compiled into multiple instructions in the machine language. Moreover, this task becomes impossible when there are multiple variables involved in the operation (as is probably the case with our queue), leading to the requirement of some extra form of state management: i.e., isolation, immutability, or synchronization.

The fact is that it's very easy to accidentally expose invalid program states as a result of subtle interactions between threads. These states might not exist on any legal state machine diagram we would have drawn for our data structure, but interleaving can cause them. Such problems frequently differ in symptom from one execution of your code to the next—causing new exceptions, data corruption, and so forth and depend on timing in order to manifest. The constant change in symptom and dependence on timing makes it difficult to anticipate the types of failures you will experience when more concurrency is added to the system and makes such failures incredibly hard to debug and fix.

The various solutions hinted at above can solve this problem. The simplest solutions are to avoid sharing data or to avoid updating data completely. Unfortunately, taking such an approach does not completely eliminate the need to synchronize. For instance, you must keep intermediate state changes confined within one thread until they are all complete and then, once the changes are suitable to become visible, you must use some mechanism to publish state updates to the globally visible set of memory as a single, indivisible operation (i.e., atomic operation). All other threads must cooperate by reading such state from the global memory space as a single, indivisible atomic operation.

This is not simple to achieve. Because reading and writing an arbitrary number of memory locations atomically at once are not supported by current hardware, software must simulate this effect using **critical regions**. A critical region ensures that only one thread executes a certain piece of code at once, eliminating problematic interleaved operations and forcing one after the other timing. This implies some threads in the system will have to wait for others to finish work before doing their own. We will discuss critical regions later. But first, let's look at a motivating example where data synchronization is direly needed.

A Simple Data Race

Consider this deceptively simple program statement.

```
int * a = ...;
(*a)++;
```

(Forgive the C++-isms for those managed programmers reading this. `(*a)++` is used instead of `a++`, just to make it obvious that `a` points to some shared memory location.)

When translated into machine code by the compiler this seemingly simple, high-level, single-line statement involves multiple machine instructions:

```
MOV EAX, [a]
INC EAX
MOV [a], EAX
```

Notice that, as a first step, the machine code dereferences `a` to get some virtual memory address and copies 4 bytes' worth of memory starting at that address into the processor local `EAX` register. The code then increments the value of its private copy in `EAX`, and, lastly, makes yet another copy of the value, this time to copy the incremented value held in its private register back to the shared memory location referred to by `a`.

The multiple steps and copies involved in the `++` operator weren't apparent in the source file at all. If you were manipulating multiple variables explicitly, the fact that there are multiple steps would be a little more apparent. In fact, it's as though we had written:

```
int * a = ...;
int tmp = *a;
tmp++;
*a = tmp;
```

Any software operation that requires multiple hardware instructions is nonatomic. And thus we've now established that `++` is nonatomic (as is `--`), meaning we will have to take extra steps to ensure concurrency safety. There are some other nonobvious sources of nonatomic operations. Modern processors guarantee that single reads from and writes to memory in increments of the natural word size of the machine will be carried out atomically covering 32-bit values on 32-bit machines and 64-bit values on 64-bit machines.

Conversely, reading or writing data with a size larger than the addressable unit of memory on your CPU is nonatomic. For instance, if you wrote a 64-bit value on a 32-bit machine, it will entail two move instructions from processor private to shared memory, each to copy a 4-byte segment. Similarly, reading from or writing to unaligned addresses (i.e., address ranges that span an addressable unit of memory) also require multiple memory operations in addition to some bit masking and shifting, even if the size of the value is less than or equal to the machine's addressable memory size. Alignment is a tricky subject and is discussed in much more detail in Chapter 10, Memory Models and Lock Freedom.

So why is all of this a problem?

An increment statement is meant to monotonically increase the value held in some memory location by a delta of 1. If three increments were made to a counter with an original value 0, you'd expect the final result to be 3. It should never be possible (overflow aside) for the value of the counter to decrease from one read to the next; therefore, if a thread executes two `(*a)++` operations, one after the other, you would expect that the second update always yields a higher value than the first. These are some very basic correctness conditions for our simple `(*a)++` program. (Note: You shouldn't be expecting that the two values will differ by precisely 1, however, since another thread might have snuck in and run between them.)

There's a problem. While the actual loads and stores execute atomically by themselves, the three operation sequence of load, increment, and store is nonatomic, as we've already established. Imagine three threads, t1, t2, and t3, are running the compiled program instructions simultaneously.

t1	t2	t3
<code>t1(0): MOV EAX, [a]</code>	<code>t2(0): MOV EAX, [a]</code>	<code>t3(0): MOV EAX, [a]</code>
<code>t1(1): INC, EAX</code>	<code>t2(1): INC, EAX</code>	<code>t3(1): INC, EAX</code>
<code>t1(2): MOV [a], EAX</code>	<code>t2(2): MOV [a], EAX</code>	<code>t3(2): MOV [a], EAX</code>

Each thread is running on a separate processor. Of course, this means that each processor has its own private EAX register, but all threads see the same value in `a` and therefore access the same shared memory. This is where time becomes a very useful tool for explaining the behavior of our concurrent programs. Each of these steps won't really happen "simultaneously." Although separate processors can certainly execute instructions

simultaneously, there is only one central, shared memory system with a cache coherency system that ensures a globally consistent view of memory. We can therefore describe the execution history of our program in terms of a simple, sequential time scale.

In the following time scale, the y-axis (labeled T) represents time, and the abscissa, in addition to a label of the form thread (sequence#) and the instruction itself, depicts a value in the form #n, where n is the value in the memory target of the move after the instruction has been executed.

T	t1	t2	t3
0	t1(0): MOV EAX,[a] #0		
1	t1(1): INC,EAX #1		
2	t1(2): MOV [a],EAX #1		
3		t2(0): MOV EAX,[a] #1	
4		t2(1): INC,EAX #2	
5		t2(2): MOV [a],EAX #2	
6			t3(0): MOV EAX,[a] #2
7			t3(1): INC,EAX #3
8			t3(2): MOV [a],EAX #3

If a is an integer that begins with a value of 0 at time step 0, then after three (*a)++ operations have executed, we expect the value to be $0 + 3 = 3$. Indeed, we see that this is true for this particular history: t1 runs to completion, leaving value 1 in *a, and then t2, leaving value 2, and finally, after executing the instruction at time 8 in our timeline, t3 has finished and *a contains the expected value 3.

We can compress program histories into more concise representations so that they fit on one line instead of needing a table like this. Because only one instruction executes at any time step, this is simple to accomplish. We'll write each event in sequence, each with a thread (sequence#) label, using the notation a → b to denote that event a happens before b. A sequence of operations is written from left to right, with the time advancing as we move from one operation to the next. Using this scheme, the above history could be written instead as follows.

`t1(0)->t1(1)->t1(2)->t2(0)->t2(1)->t2(2)->t3(0)->t3(1)->t3(2)`

We'll use one form or the other depending on the level of scrutiny in which we're interested for that particular example. The longhand form is

often clearer to illustrate specific values and is better at visualizing subtle timing issues, particularly for larger numbers of threads.

No matter the notation, examining timing like this is a great way of reasoning about the execution of concurrent programs. Programmers are accustomed to thinking about programs as a sequence of individual steps. As you develop your own algorithms, writing out the concurrent threads and exploring various legal interleavings and what they mean to the state of your program, it is imperative to understanding the behavior of your concurrent programs. When you think you might have a problematic timing issue, going to the whiteboard and trying to devise some problematic history, perhaps in front of a colleague, is often an effective way to uncover concurrency hazards (or determine their absence).

Simple, noninterleaved histories pose no problems for our example. The following histories are also safe with our algorithm as written.

```
t1(0)->t1(1)->t1(2)->t3(0)->t3(1)->t3(2)->t2(0)->t2(1)->t2(2)
t2(0)->t2(1)->t2(2)->t1(0)->t1(1)->t1(2)->t3(0)->t3(1)->t3(2)
t2(0)->t2(1)->t2(2)->t3(0)->t3(1)->t3(2)->t1(0)->t1(1)->t1(2)
t3(0)->t3(1)->t3(2)->t1(0)->t1(1)->t1(2)->t2(0)->t2(1)->t2(2)
t3(0)->t3(1)->t3(2)->t2(0)->t2(1)->t2(2)->t1(0)->t1(1)->t1(2)
```

These histories yield correct results because none results in one thread's statements interleaving amongst another's. In each scenario, the first thread runs to completion, then another, and then the last one. In these histories, the threads are **serialized** with respect to one another (or the history is **serializable**).

But this example is working properly by virtue of sheer luck. There is nothing to prevent the other interleaved histories from occurring at runtime, where two (or more) threads overlap in time, leading to an interleaved timing and resulting race conditions. Omitting t3 from the example for a moment, consider this simple timing, written out longhand so we can emphasize the state transitions from one time step to the next.

T t1	t2
0 t1(0): MOV EAX,[a] #0	t2(0): MOV EAX,[a] #0
1	t2(1): INC,EAX #1
2	t2(2): MOV [a],EAX #1
4 t1(1): INC,EAX #1	
5 t1(2): MOV [a],EAX #1	

The value of `*a` starts at 0. Because two increments happen, we would expect the resulting value to be $0 + 2 = 2$. However, `*a` ends up at 1. This clearly violates the first correctness condition of our algorithm as stated initially: for each thread that invokes the increment operator, the global counter increments by exactly 1.

This is a classic race condition, or more precisely, a data race, because, in this case, our problems are caused by a lack of data synchronization. It is called a “race” because the correctness of our code depends squarely on the outcome of multiple threads racing with one another. It’s as if each is trying to get to the finish line first, and, depending on which gets there first, the program will yield different results, sometimes correct and sometimes not. Races are just one of many issues that can arise when shared state is involved and can be a serious threat to program correctness. A thorough exploration of concurrency hazards, including races, is presented in Chapter 11, Concurrency Hazards.

Why did this race manifest? It happened because `t1` and `t2` each made a copy of the shared memory value in their own processor local register, one after the other, both observing the same value of 0, and then incremented their own private copies. Then both copied their new values back into the shared memory without any validation or synchronization that would prevent one from overwriting the other’s value. Both threads calculate the value 1 in their private registers, without knowledge of each other, and, in this particular case, `t1` just overwrites `t2`’s earlier write of 1 to the shared location with the same value.

One might question how likely this is to occur. (Note that the likelihood matters very little. The mere fact that it can occur means that it is a very serious bug. Depending on the statistical improbability of such things is seriously discouraged. A program is not correct unless all possible sources of data races have been eliminated.) This interleaved history can happen quite easily, for obvious reasons, if `t1` and `t2` were running on separate processors. The frequency depends on the frequency with which the routine is accessed, among other things. This problem can also arise on a single processor machine, if a context switch occurred—because `t1`’s quantum had expired, because `t2` was running at a higher priority, and so forth—right after `t1` had moved the contents of `a` into its `EAX` register or after it had

incremented its private value. The probability of this happening is higher on a machine with multiple processors, but just having multiple threads running on a single processor machine is enough. The only way this may be impossible is if code accessing the same shared state is never called from multiple threads simultaneously.

Other execution histories exhibit the same problem.

```
t1(0)->t2(0)->t1(1)->t1(2)->t2(1)->t2(2)
t1(0)->t1(1)->t2(0)->t1(2)->t2(1)->t2(2)
t2(0)->t1(0)->t1(1)->t1(2)->t2(1)->t2(2)
...and so on
```

If we add the t3 thread back into the picture, we can violate the second correctness condition of our simple increment statement, in addition to the first, all at once.

T t1	t2	t3
0		
1 t1(0): MOV EAX,[a] #0		
2 t1(1): INC,EAX #1		
3 t1(2): MOV [a],EAX #1		
4	t2(0): MOV EAX,[a] #1	
5	t2(1): INC,EAX #2	
6	t2(2): MOV [a],EAX #2	
7		t3(1): INC,EAX #1
8		t3(2): MOV [a],EAX #1

In this program history, the global counter is updated to 1 by t1, and then to 2 by t2. Everything looks fine from the perspective of other threads in the system at this point in time. But as soon as t3 resumes, it wipes out t1's and t2's updates, "losing" two values from the counter and going backward to a value of 1. This is because t3 made its private copy of the shared value of *a before t1 and t2 even ran. The second correctness condition was that the value only ever increases; but if t2 runs again, it will see a value smaller than the one it previously published. This is clearly a problem that is apt to break whatever algorithm is involved. As we add more and more threads that are frequently running close together in time, we increase the probability of such problematic timings accordingly.

All of these histories demonstrate different kinds of hazards.

- **Read/write hazard.** A thread, t1, reads from a location, t2, then writes to that location, and t1 subsequently makes a decision based on its (now invalid) read of t1. This also can be referred to as a **stale read**.
- **Write/write hazard.** A thread, t1, writes to the same location as t2 in a concurrency unsafe way, leading to lost updates, as in the example given above.
- **Write/read hazard.** A thread, t1, writes to a location and then t2 reads from it before it is safe to do so. In some cases, t1 may decide to undo its partial update to state due to a subsequent failure, leading t2 to make decisions on an invalid snapshot of state that should have never been witnessed. This also can be referred to as an **unrepeatable read**.
- **Read/read hazard.** There is no problem with multiple concurrent threads reading the same shared data simultaneously. This property can be exploited to build a critical region variant called a **reader/ writer lock** to provide better performance for read/read conflicts; this idea is explored more in Chapter 6, Data and Control Synchronization.

(This last point is a simplification. Normally read/read conflicts are safe in the case of simple shared memory, but there are some cases in which they are not: when a read has a side effect, like reading a stack's guard page, or when reading some data associated with a physical device, it may be necessary to ensure no two threads try to do it concurrently.)

Very little of this discussion is specific to the `++` operator itself. It just turns out to be a convenient example because it intrinsically exhibits all of the problematic conditions that lead to these timing issues.

1. Multiple threads make private copies of data from a shared location.
2. Threads publish results back to shared memory, overwriting existing values.
3. Compound updates may be made with the intent of establishing or preserving invariants between multiple independent shared locations.
4. Threads run concurrently such that their timing overlaps and operations interleave.

There is no greater skill that differentiates great concurrent programmers from the rest than the ability to innately predict and consider various timings to some reasonable depth of complexity. With experience comes the ability to see several steps ahead and proactively identify the timings that can lead to race conditions and other hazards. This is especially important when writing sophisticated **lock free** algorithms, which eschew isolation, immutability, and synchronization in favor of strict discipline and reliance on hardware guarantees, which we'll review in Chapter 10, Memory Models and Lock Freedom.

On Atomicity, Serializability, and Linearizability

A fundamental problem is that many program operations are not truly atomic because an operation consists of multiple logical steps, a certain logical step is comprised of many physical steps, or both. **Atomicity**, quite simply, is the property that a single operation or set of operations appear as if they happened at once. Any state modifications and side effects performed are completely instantaneous, and no other thread in the system can observe intermediary (and invalid) states that occur in the midst of such an atomic operation. Similarly, the atomic operation must not be permitted to fail part way through the update, or if it does so, there must be a corresponding roll back of state updates to the previous state.

By this definition, atomicity would seldom be practical to achieve, at least physically. Although processors guarantee single writes to aligned words of memory are truly atomic, higher level logical operations—like the execution of a single method call on an object, consisting of several statements, function calls, and reads and writes—are not so simple. In fact, sometimes the operations we'd like to make atomic can even span physical machines, perhaps interacting with a Web service or database, at which point the difficulty of ensuring atomicity is greater. System wide control mechanisms must be used to achieve atomicity except for very simple read and write operations. As already noted, critical regions can simulate atomicity for in-memory updates. Transactions, of the ilk found in databases, COM+, and the `System.Transactions` namespace in .NET, are also attractive solutions when multiple or persistent durable resources are involved.

When two operations are atomic, they do not appear to overlap in time. If we were to plot several atomic operations on a timeline, then we could place one before or after the other without worrying about having to interleave them. We did this earlier for individual reads and writes, and it was possible because of the guarantees made by the hardware that they are atomic. Object oriented programs are typically built from higher level atomic methods, however, and reasoning about concurrency at this level (like “puts an element in the queue,” “writes data to disk,” and so forth), and not about the individual memory reads and writes involved, is often more useful.

Serializability is when two operations happen one after the other; if a happens before b , then a *serializes before* b . Building your program out of atomic operations achieves serializability. It’s as though your program was executed sequentially, by a single processor, by executing each atomic operation in the sequence as it appeared in the resulting serializable order. But serializability on its own is insufficient for correctness; and it’s often in the eye of the beholder—remember that even individual reads and writes are themselves atomic. For a concurrent program to be correct, all possible serialization orders must be legal. Techniques like critical regions can be used to constrain legal serialization orders.

Linearizability is a property related to serializability and also is used to describe correctness of atomic operations (see Further Reading, Herlihy, Wing): a *linearization point* is a place when a batch of atomic updates becomes visible to other threads. This commonly corresponds to exiting a critical region where the updates made within suddenly become visible. It is typically easier to reason about linearization points instead of the more abstract serialization property.

Atomic operations also must be reorderable, such that having one start completely before the other still leads to a correct program schedule. That’s not to say that subsequently initiated operations will not change behavior based on the changed order of commutative operations, due to causality, but this reordering should not fundamentally alter the correctness of a program.

As software developers, we like to think of serializable schedules and atomic operations. But we’d also like to use concurrency for the reasons

identified earlier in this book, for performance, responsiveness, and so on. For this reason, the Win32 and .NET Framework platforms give you a set of tools to achieve atomicity via data synchronization constructs, as implied earlier. Those familiar with relational databases will recognize a similarity: databases employ transactions to achieve serializable operations, giving the programmer an interface with atomicity, consistency, isolation, and durability (a.k.a. ACID). You will notice many similarities, but you will also notice that these properties must be achieved “by hand” in general purpose concurrent programming environments.

Isolation

An obvious approach to eliminating problematic shared state interactions is to avoid sharing state in the first place. We described how concurrent systems are typically formed out of higher level components that eschew sharing in favor of isolation, and that lower level components typically do share data for purposes of fine-grained, performance sensitive operations. This is a middle ground, but the two extremes are certainly possible: on one hand, all components in the system may share state, while, on the other hand, no components share state and instead communicate only via loosely coupled messages. And there are certainly situations in which the architecture is less clearly defined: i.e., some lower level components will use isolation, while some higher level components will share state for efficiency reasons.

When it comes to employing isolation, there are three basic techniques from which to choose.

- **Process isolation.** Each Windows process has a separate memory address space, ensuring that one process cannot read or write memory used by another. Hardware protection is used to absolutely guarantee that there is no chance of accidental sharing by bleeding memory references. Note that processes do share some things, like machine-wide kernel objects, the file system, memory mapped files, and so on, so even rigid process isolation can be broken. An even more extreme technique is isolating components on separate machines or inside virtualized partitions on a single machine.

- **Intraprocess isolation.** If you are using managed code, CLR Application Domains (AppDomains) can be used to isolate objects so that code running in one AppDomain cannot read or write an object running in another AppDomain. While hardware protection is not used to enforce this isolation, the verifiable type safety employed by the CLR ensures that no sharing will occur. There are some specific ways to circumvent this broadly stated policy, but they are generally opt-in and rare.
- **By convention.** When some code allocates a piece of memory or an object, either dynamically from the heap or on the stack, this data begins life as unshared, and, hence, is in effect isolated. This data remains isolated so long as care is taken to not share the data (as described previously), that is, by not storing a reference to the data in a shared location (like a static variable or object reachable through a static variable). This is the trickiest of the three approaches to implement safely because it is entirely based on programming convention and care, is not checkable in any way and has no infrastructure regulated support such as hardware isolation or type system verification.

It's common to use isolated state as a form of cache. In other words, though some state is physically isolated, it is merely a copy of some master copy that is not isolated. Such designs require that the master copy is periodically refreshed (if updates are made to the cache) and that caches are refreshed as the master copy changes. Depending on the requirements, a more sophisticated cache coherency mechanism may be needed, to guarantee that refreshes happen in a safe and serializable way, requiring a combination of isolation and synchronization techniques.

The last mechanism, enforcement by convention, requires that programs follow some strict disciplines, each of which is cause for concern because they are informal and inherently brittle. It can be useful to think of state in terms of being “owned” by particular “agents” at any point in time. Thinking this way allows you to very clearly articulate where ownership changes for a particular piece of data, including at what point data transitions between isolated and shared.

Data Ownership

At any point in time, a particular piece of isolated data can be said to be owned by one agent in the system. Ownership is used in this context to mean that the agent understands what other components or agents may concurrently access that piece of data, and what this means for the read and write safety of its own operations. If, at any time, multiple agents believe they own the same piece of data, it is likely that the data is no longer truly isolated. Clearly there are many kinds of ownership patterns a system might employ, including shared ownership, but let's stick to the idea of single agent ownership for a moment.

An agent may transfer ownership, but it must do so with care. For example, some agent may allocate and initialize some interesting object, but then insert it into a global shared list. This is called publication. Publication transfers ownership from the initializing agent to the global namespace; at some point in the future, an agent may remove the data from the shared list, at which point the ownership transfers from the global namespace to that agent. This is called privatization. Publication must be done such that the agent doing the transferring no longer tries to access the state as though it is the sole owner: the data is no longer confined (or isolated) within the agent. Similarly, privatization must be done such that other agents do not subsequently try to access the privatized data.

One of the more difficult aspects of ownership is that a piece of data may move between isolation and shared status over the course of its life. These publication and privatization points must be managed with care. A slight misstep, such as erroneously believing an object is private and no longer shared when in reality other threads still have outstanding references to it that they might use, can introduce all of the same kinds of race condition problems noted earlier.

Another challenge with isolation is determining where the points of escape in the program might be. Publication is not always such a clear-cut point in the program's execution. This requires that agents attempting to control ownership of data only ever share references to this data with trusted agents. The agent is trusting that the other agents will not publish the reference so that the data becomes shared, either directly or indirectly (e.g., by passing the reference along to another untrusted agent).

Similarly, an agent that receives a reference to data from an outside source must assume the worst—that the data is shared—unless an alternative arrangement is known, such as only ever being called by an agent that guarantees the data is isolated. Again, the lack of type system and verification support makes this convention notoriously tricky to implement and manage in real programs, particularly when multiple developers are involved.

Immutability

As noted earlier, read / read “hazards” are not really hazardous at all. Many threads can safely read from some shared memory location concurrently without cause for concern. Therefore, if some piece of shared state is guaranteed to be immutable—that is, read-only—then accessing it from many threads inside a concurrent system will be safe.

Proving that a piece of complex data is immutable is not terribly difficult with some discipline. Both C++ and .NET offer constructs to help make immutable types. If each of an object’s fields never changes during its lifetime, it is **shallow immutable**. If the object’s fields also only refer to objects whose state does not change over time, the object is **deeply immutable**. An entire object graph can be transitively immutable if all objects within it are themselves deeply immutable.

In the case that data transitions between private and shared throughout its lifetime, as discussed above in the context of isolation, it is sometimes useful to have a conditionally-immutable type, in which it remains immutable so long as it is shared but can be mutated while private. So, for example, a thread may remove a piece of shared state from public view, making it temporarily private, mutate it, and then later share the state again to public view.

Single Assignment

A popular technique for enforcing the immutability of data is to use **single assignment variables**. Many programming systems offer static verification that certain data is indeed assigned a value only once, leading to the term **static single assignment**, or **SSA**.

The CLR offers limited support for single assignment variables in its common type system through the `initonly` field modifier, surfaced in C#.

through the `readonly` keyword. And C++ offers the `const` modifier to achieve a similar effect, though it is far more powerful: pointers may be marked as being `const`, ensuring (statically) that the instance referred to is not modified by the user of such a pointer (though unlike `readonly` C++ programmers can explicitly “cast away the `const`-ness” of a reference, bypassing the safety guarantees). Using these constructs can be tremendously useful because it avoids having to depend on brittle and subtle programming convention and rules. Let’s look at each briefly.

CLR `initonly` Fields (a.k.a. C# `readonly` Fields). When you mark a field as `readonly` in C#, the compiler emits a field with the `initonly` modifier in the resulting IL. The only writes to such variables that will pass the type system’s verification process are those that occur inside that type’s constructor or field initializers. This ensures that the value of such a field cannot change after the object has been constructed. While it is not a true single assignment variable, as it can be written multiple times during initialization, it is similar in spirit.

Another subtle issue can arise if a reference to an object with `readonly` fields escapes from its constructor. Fields are not guaranteed to have been initialized with the permanent immutable values until after the constructor has finished running and could be assigned multiple values during the construction process. If an object’s constructor shares itself before finishing initialization, then other concurrent threads in the system cannot safely depend on the `readonly` nature of the fields. Letting the object’s `this` reference escape before the object is fully constructed like this is a bad practice anyway, and is easily avoided. When a field is marked `readonly`, it simply means the field’s value cannot change. In other words, a type with only `readonly` fields is shallow immutable but not necessarily deeply immutable. If an object depends on the state of the objects it references, then those objects should be immutable also. Unfortunately, the CLR offers no type system support for building deeply immutable types. Of course they may be immutable by convention, `readonly` fields, or a combination of both.

There are some cases where the mutability of referenced objects does not matter. For example, if we had an immutable pair class that refers to two mutable objects but never accesses the state of those objects, then is the pair itself immutable?

```

class ImmutablePair<T, U>
{
    private readonly T m_first;
    private readonly U m_second;

    public ImmutablePair(T first, U second)
    {
        m_first = first;
        m_second = second;
    }

    public T First { get { return m_first; } }
    public U Second { get { return m_second; } }
}

```

From one perspective, the answer is yes. The `ImmutablePair<T, U>` implementation itself cannot tell whether the `m_first` or `m_second` objects have been mutated, since it never accesses their internal state. If it relied on a stable `ToString` value, then it might matter. Those who instantiate `ImmutablePair<T, U>` may or may not care about deep immutability, depending on whether they access the pair's fields; they control this by the arguments they supply for `T` and `U`. So it seems shallow immutability here is sufficient. That said, if a developer desires deep immutability, they need only supply immutable types for `T` and `U`.

C++ Const. C++ `const` is a very powerful and feature rich-programming language construct, extending well beyond simple single assignment variable capabilities, and encompassing variables, pointers, and class members. A complete overview of the feature is outside of the scope of this book. Please refer instead to a book such as *The C++ Programming Language*, Third Edition (see Further Reading, Stroustrup), for a detailed overview.

Briefly, the `const` modifier can be a useful and effective way of relying on the C++ compiler to guarantee a certain level of immutability in your data structures, including single assignment variables. In summary:

- Class fields may be marked `const`, which enforces that their value is assigned at initialization time in the constructor's field initialization list and may not subsequently change. This effectively turns a field into a single assignment variable, though it may still be modified by a pointer that has been cast a certain way (as we'll see soon).

The value of static `const` fields cannot depend on runtime evaluation, unlike class member fields that can involve arbitrary runtime computation to generate a value, much like CLR `initonly` fields. This means they are limited to compiler constants, statically known addresses, and inline allocated arrays of such things.

- Member functions may be marked `const`, which means that the function body must not modify any fields and ensures that other non-`const` member functions cannot be invoked (since they may modify fields).
- Pointers can be marked as “pointing to a constant,” via the prefix `const` modifier. For instance, the following declaration states that `d` points to a constant object of type `C`: `const C * d`. When a pointer refers to a constant, only `const` member functions may be called on it, and the pointer may not be passed where an ordinary non-`const` pointer is expected. A `const` pointer to an integral type cannot be written through. A non-`const` pointer can be supplied where a `const` is expected. Constant references are also possible.

As noted earlier, a pointer to a constant can be cast to a non-`const` pointer, which violates most of what was mentioned above. For example, the C++ compiler enforces that a pointer to a `const` member field also must be a pointer to `const`; but you can cast this to a non-`const` pointer and completely subvert the `const` guarantees protecting the field. For example, given the following class declaration, pointers may be manufactured and used in certain ways.

```
class C
{
public:
    const int d;
    C(int x) : d(x) {}
};

// ... elsewhere ...

C * pC = ...;
const int * pCd1 = &pC->d; // ok!
*pCd1 = 42; // compiler error: cannot write to const
int * pCd2 = &pC->d; // compiler error: non-const pointer to const field
int * pCd3 = const_cast<int *>(&pC->d); // succeeds!
```

Casting away `const` is a generally frowned upon practice, but is sometimes necessary. And, a `const` member function can actually modify state, but only if those fields have been marked with the `mutable` modifier. Using this modifier is favored over casting. Despite these limitations, liberal and structured use of `const` can help build up a stronger and more formally checked notion of immutability in your programs. Some of the best code bases I have ever worked on have used `const` pervasively, and in each case, I have found it to help tremendously with the maintainability of the system, even with concurrency set aside.

Dynamic Single Assignment Verification. In most concurrent systems, single assignment has been statically enforced, and C# and C++ have both taken similar approaches. It's possible to dynamically enforce single assignment too. You would just have to reject all subsequent attempts to set the variable after the first (perhaps via an exception), and handle the case where threads attempt to use an uninitialized variable. Implementing this does require some understanding of the synchronization topics about to be discussed, particularly if you wish the end result to be efficient; some sample implementation approaches can be found in research papers (see Further Reading, Drejhammar, Schulte).

Synchronization: Kinds and Techniques

When shared mutable state is present, synchronization is the only remaining technique for ensuring correctness. As you might guess, given that there's an entire chapter in this book dedicated to this topic—Chapter 11, Concurrency Hazards—implementing a properly synchronized system is complicated. In addition to ensuring correctness, synchronization often is necessary for behavioral reasons: threads in a concurrent system often depend on or communicate with other threads in order to accomplish useful functionality.

The term synchronization is admittedly overloaded and too vague on its own to be very useful. Let's be careful to distinguish between two different, but closely related, categories of synchronization, which we'll explore in this book:

1. **Data synchronization.** Shared resources, including memory, must be protected so that threads using the same resource in parallel do

not interfere with one another. Such interference could cause problems ranging from crashes to data corruption, and worse, could occur seemingly at random: the program might produce correct results one time but not the next. A piece of code meant to move money from one bank account to another, written with the assumption of sequential execution, for instance, would likely fail if concurrency were naively added. This includes the possibility of reaching a state in which the transferred money is in neither account! Fixing this problem often requires using mutual exclusion to ensure no two threads access data at the same time.

2. **Control synchronization.** Threads can depend on each others' traversal through the program's flow of control and state space. One thread often needs to wait until another thread or set of threads have reached a specific point in the program's execution, perhaps to rendezvous and exchange data after finishing one step in a cooperative algorithm, or maybe because one thread has assumed the role of orchestrating a set of other threads and they need to be told what to do next. In either case, this is called control synchronization.

The two techniques are not mutually exclusive, and it is quite common to use a combination of the two. For instance, we might want a producer thread to notify a consumer that some data has been made available in a shared buffer, with control synchronization, but we also have to make sure both the producer and consumer access the data safely, using data synchronization.

Although all synchronization can be logically placed into the two general categories mentioned previously, the reality is that there are many ways to implement data and control synchronization in your programs on Windows and the .NET Framework. The choice is often fundamental to your success with concurrency, mostly because of performance. Many design forces come into play during this choice: from correctness—that is, whether the choice leads to correct code—to performance—that is, the impact to the sequential performance of your algorithm—to liveness and scalability—that is, the ability of your program

to ensure that, given the addition of more and more processors, the throughput of the system improves commensurately (or at least doesn't do the inverse of this).

Because these are such large topics, we will tease them apart and review them in several subsequent chapters. In this chapter, we stick to the general ideas, providing motivating examples as we go. In Chapter 5, Windows Kernel Synchronization, we look at the foundational Windows kernel support used for synchronization, and then in Chapter 6, Data and Control Synchronization, we will explore higher level primitives available in Win32 and the .NET Framework. We won't discuss performance and scalability in great depth until Chapter 14, Performance and Scalability, although it's a recurring theme throughout the entire book.

Data Synchronization

The solution to the general problem of data races is to serialize concurrent access to shared state. Mutual exclusion is the most popular technique used to guarantee no two threads can be executing the sensitive region of instructions concurrently. The sequence of operations that must be serialized with respect to all other concurrent executions of that same sequence of operations is called a **critical region**.

Critical regions can be denoted using many mechanisms in today's systems, ranging from language keywords to API calls, and involving such terminology as *locks*, *mutexes*, *critical sections*, *monitors*, *binary semaphores*, and, recently, *transactions* (see Further Reading, Shavit, Touitou). Each has its own subtle semantic differences. The desired effect, however, is usually roughly the same. So long as all threads use critical regions consistently to access certain data, they can be used to avoid data races.

Some regions support shared modes, for example reader/writer locks, when it is safe for many threads to be reading shared data concurrently. We'll look at examples of this in Chapter 6, Data and Control Synchronization. We will assume strict mutual exclusion for the discussion below.

What happens if multiple threads attempt to enter the same critical region at once? If one thread wants to enter the critical region while another

is already executing code inside, it must either wait until the thread leaves or it must occupy itself elsewhere in the meantime, perhaps checking back again sometime later to see if the critical region has become available. The kind of waiting used differs from one implementation to the next, ranging from busy waiting to relying on Windows' support for waiting and signaling. We will return to this topic later.

Let's take a brief example. Given some statement or compound statement of code, S , that depends on shared state and may run concurrently on separate threads, we can make use of a critical region to eliminate the possibility of data races.

```
EnterCriticalSection();
S;
LeaveCriticalSection();
```

(Note that these APIs are completely fake and simply used for illustration.)

The semantics of the faux `EnterCriticalSection` API are rather simple: only one thread may enter the region at a time and must otherwise wait for the thread currently inside the region to issue a call to `LeaveCriticalSection`. This ensures that only one thread may be executing the statement S at once in the entire process and, hence, serializes all executions. It appears as if all executions of S happen atomically—provided there is no possibility of concurrent access to the state accessed in S outside of critical regions, and that S may not fail part-way through—although clearly S is not really atomic in the most literal sense of the word.

Using critical regions can solve both data invariant violations illustrated earlier, that is when S is $(*a)++$, as shown earlier. Here is the first problematic interleaving we saw, with critical regions added into the picture.

T	t1	t2
0	<code>t1(E): EnterCriticalSection();</code>	
1	<code>t1(0): MOV EAX,[a] #0</code>	
2		<code>t2(0): EnterCriticalSection();</code>
3	<code>t1(1): INC,EAX #1</code>	
4	<code>t1(2): MOV [a],EAX #1</code>	
5	<code>t1(L): LeaveCriticalSection();</code>	
6		<code>t2(0): MOV EAX,[a] #1</code>
7		<code>t2(1): INC,EAX #2</code>
8		<code>t2(2): MOV [a],EAX #3</code>
9		<code>t2(L): LeaveCriticalSection();</code>

In this example, t2 attempts to enter the critical region at time 2. But the thread is not permitted to proceed because t1 is already inside the region and it must wait until time 5 when t1 leaves. The result is that no two threads may be operating on a simultaneously.

As alluded to earlier, any other accesses to `a` in the program must also be done under the protection of a critical region to preserve atomicity and correctness across the whole program. Should one thread forget to enter the critical region before writing to `a`, shared state can become corrupted, causing cascading failures throughout the program. For better or for worse, critical regions in today's programming systems are very code-centric rather than being associated with the data accessed inside those regions.

A Generalization of the Idea: Semaphores

The semaphore was invented by E. W. Dijkstra in 1965 as a generalization of the general critical region idea. It permits more sophisticated patterns of data synchronization in which a fixed number of threads are permitted to be inside the critical region simultaneously.

The concept is simple. A semaphore is assigned an **initial count** when created, and, so long as the count remains above 0, threads may continue to decrement the count without waiting. Once the count reaches 0, however, any threads that attempt to decrement the semaphore further must wait until another thread releases the semaphore, increasing the count back above 0. The names Dijkstra invented for these operations are **P**, for the fictitious word **prolaag**, meaning to try to take, and **V**, for the Dutch word **verhoog**, meaning to increase. Since these words are meaningless to those of us who don't speak Dutch, we'll refer to these activities as **taking** and **releasing**, respectively.

A critical region (a.k.a. **mutex**) is therefore just a specialization of the semaphore in which its current count is always either 0 or 1, which is also why critical regions are often called **binary semaphores**. Semaphores with maximum counts of more than 1 are typically called **counting semaphores**. Windows and .NET both offer intrinsic support for semaphore objects. We will explore this support further in Chapter 6, Data and Control Synchronization.

Patterns of Critical Region Usage

The faux syntax shown earlier for entering and leaving critical regions maps closely to real primitives and syntax. We'll generally interchange the terminology enter/leave, enter/exit, acquire/release, and begin/end to mean the same thing. In any case, there is a pair of operations for the critical region: one to enter and one to exit. This syntax might appear to suggest there is only one critical region for the entire program, which is almost never true. In real programs, we will deal with multiple critical regions, protecting different disjoint sets of data, and therefore, we often will have to instantiate, manage, and enter and leave specific critical regions, either by name, object reference, or some combination of both, during execution.

A thread wishing to enter some region 1 does not interfere with a separate region 2 and vice versa. Therefore, we must ensure that all threads consistently enter the correct region when accessing certain data. As an illustration, imagine we have two separate `CriticalRegion` objects, each with `Enter` and `Leave` methods. If two threads tried to increment a shared variable `s_a`, they *must* acquire the same `CriticalRegion` first. If they acquire separate regions, mutual exclusion is not guaranteed and the program has a race.

Here is an example of such a broken program.

```
static int a;
static CriticalRegion cr1, cr2; // initialized elsewhere
void f() { cr1.Enter(); s_a++; cr1.Leave(); }
void g() { cr2.Enter(); s_a++; cr2.Leave(); }
```

This example is flawed because `f` acquires critical region `cr1` and `g` acquires critical region `cr2`. But there are no mutual exclusion guarantees between these separate regions. If one thread runs `f` concurrently with another thread that is running `g`, we will see data races.

Critical regions are most often—but not always—associated with some static lexical scope, in the programming language sense, as shown above. The program enters the region, performs the critical operation, and exits, all occurring on the same stack frame, much like a block scope in C based languages. Keep in mind that this is just a common way to group

synchronization sensitive operations under the protection of a critical region and not necessarily a restriction imposed by the mechanisms you will be using. (Many encourage it, however, like C# and VB, which offer keyword support.) It's possible, although often more difficult and much more error prone, to write a critical region that is more dynamic about entering and leaving regions.

```
BOOL f()
{
    if (...)

    {
        EnterCriticalSection();
        S0; // some critical work
        return TRUE;
    }
    return FALSE;
}

void g()
{
    if (f())
    {
        S1; // more critical work
        LeaveCriticalSection();
    }
}
```

This style of critical region use is more difficult for a number of reasons, some of which are subtle. First, it is important to write programs that spend as little time as possible in critical regions, for performance reasons. This example inserts some unknown length of instructions into the region (i.e., the function return epilogue of `f` and whatever the caller decides to do before leaving). Synchronization is also difficult enough, and spreading a single region out over multiple functional units adds difficulty where it is not needed.

But perhaps the most notable problem with the more dynamic approach is reacting to an exception from within the region. Normally, programs will want to guarantee the critical region is exited, even if the region is terminated under exceptional circumstances (although not always, as this failure can indicate data corruption). Using a statically scoped block allows you to use things like `try/catch` blocks to ensure this.

```
EnterCriticalSection();
__try
{
    S0; S1; // critical work
}
__finally
{
    LeaveCriticalSection();
}
```

Achieving this control flow for failure and success becomes more difficult with more dynamism. Why might we care so much about guaranteeing release? Well, if we don't always guarantee the lock is released, another thread may subsequently attempt to enter the region and wait indefinitely. This is called an **orphaned lock** and leads to deadlock.

Simply releasing the lock in the face of failure is seldom sufficient, however. Recall that our definition of atomicity specifies two things: that the effects appear instantaneously and that they happen either completely or not at all. If we release the lock immediately when a failure occurs, we may be opening up data corruption to the rest of the program. For example, say we had two shared variables *x* and *y* with some known relationship based invariant; if a region modified *x* but failed before it had a chance to modify *y*, releasing the region would expose the corrupt data and likely lead to additional failure in other parts of the program. Deadlock is generally more debuggable than data corruption, so if the code cannot be written to revert the update to *x* in the face of such a failure, it's often a better idea to leave the region in an acquired state. That said we will use a try/finally type of scheme in examples to ensure the region is exited properly.

Coarse- vs. Fine-Grained Regions

When using a critical region, you must decide what data is to be protected by which critical regions. Coarse- and fine-grained regions are two extreme ends of the spectrum. At one extreme, a single critical region could be used to protect all data in the program; this would force the program to run single-threaded because only one thread could make forward progress at once. At the other extreme, every byte in the heap could be protected by its own critical region; this might alleviate scalability bottlenecks, but would be ridiculously expensive to implement, not to mention impossible to

understand, ensure deadlock freedom, and so on. Most systems must strike a careful balance between these two extremes.

The critical region mechanisms available today are defined by regions of program statements in which mutual exclusion is in effect, as shown above, rather than being defined by the data accessed within such regions. The data accessed is closely related to the program logic, but not directly: any given data can be manipulated by many regions of the program and similarly any given region of the program is apt to manipulate different data. This requires many design decisions and tradeoffs to be made around the organization of critical regions.

Programs are often organized as a collection subsystems and composite data structures whose state may be accessed concurrently by many threads at once. Two reasonable and useful approaches to organizing critical regions are as follows:

- **Coarse-grained.** A single lock is used to protect all constituent parts of some subsystem or composite data structure. This is the simplest scheme to get right. There is only one lock to manage and one lock to acquire and release: this reduces the space and time spent on synchronization, and the decision of what comprises a critical region is driven entirely by the need of threads to access some large, easy to identify thing. Much less work is required to ensure safety. This over conservative approach may have a negative impact to scalability due to false sharing, however. False sharing prevents concurrent access to some data unnecessarily, that is it is not necessary to guard access to ensure correctness.
- **Fine-grained.** As a way of improving scalability, we can use a unique lock per constituent piece of data (or some groupings of data), enabling many threads to access disjoint data objects simultaneously. This reduces or eliminates false sharing, allowing threads to achieve greater degrees of concurrency and, hence, better liveness and scalability. The down side to this approach is the increase of number of locks to manage and potentially multiple lock acquisitions needed if more than one data structure must be accessed at once, both of which are bad for space and time complexity. This

strategy also can lead to deadlocks if not used carefully. If there are complex invariant relationships between multiple data structures, it can also become more difficult to eliminate data races.

No single approach will be best for all scenarios. Programs will use a combination of techniques on this spectrum. But as a general rule of thumb, starting with coarse-grained locking to ensure correctness first and fine-tuning the approach to successively use finer-grained regions as scalability requirements demand is an approach that typically leads to a more maintainable, understandable, and bug-free program.

How Critical Regions Are Implemented

Before moving on, let's briefly explore how critical regions might be implemented. There are a series of requirements for any good critical region implementation.

1. The mutual exclusion property holds. That is, there can never be a circumstance in which more than one thread enters the critical region at once.
2. Liveness of entrance and exit of the region is guaranteed. The system as a whole will continue to make forward progress, meaning that the algorithm can cause neither deadlock nor livelock. More formally, given an infinite amount of time, each thread that arrives at the region is guaranteed to eventually enter the region, provided that no thread stays in the region indefinitely.
3. Some reasonable degree of fairness, such that a thread's arrival time at the region somehow gives it (statistical) preference over other threads, is desirable though not strictly required. This does not necessarily dictate that there is a deterministic fairness guarantee—such as first-in, first-out—but often regions strive to be reasonably fair, probabilistically speaking.
4. Low cost is yet another subjective criterion. It is important that entering and leaving the critical region be very inexpensive. Critical regions are often used pervasively in low-level systems software,

such as operating systems, and thus, there is a lot of pressure on the efficiency of the implementation.

As we'll see, there is a progression of approaches that can be taken. In the end, however, we'll see that all modern mutual exclusion mechanisms rely on a combination of atomic compare and swap (CAS) hardware instructions and operating system support. But before exploring that, let's see why hardware support is even necessary. In other words, shouldn't it be easy to implement `EnterCriticalSection` and `LeaveCriticalSection` using familiar sequential programming constructs?

The simplest, overly naïve approach won't work at all. We could have a single flag variable, initially 0, which is set to 1 when a thread enters the region and 0 when it leaves. Each thread attempting to enter the region first checks the flag and then, once it sees the flag at 0, sets it to 1.

```
int taken = 0;

void EnterCriticalSection()
{
    while (taken != 0) /* busy wait */ ;
    taken = 1; // Mark the region as taken.
}

void LeaveCriticalSection()
{
    taken = 0; // Mark the region as available.
}
```

This is fundamentally very broken. The reason is that the algorithm uses a sequence of reads and writes that aren't atomic. Imagine if two threads read `taken` as 0 and, based on this information, both decide to write 1 into it. Multiple threads would each think it owned the critical region, but both would be running code inside the critical region at once. This is precisely the thing we're trying to avoid with the use of critical regions in the first place!

Before reviewing the state of the art—that is, the techniques all modern critical regions use—we'll take a bit of a historical detour in order to better understand the evolution of solutions to mutual exclusion during the past 40+ years.

Strict Alternation. We might first try to solve this problem with a technique called **strict alternation**, granting ownership to thread 0, which then grants ownership to thread 1 when it is done, which then grants ownership to 2 when it is done, and so on, for N threads, finally returning ownership back to 0 after thread N – 1 has been given ownership and finished running inside the region. This might be implemented in the form of the following code snippet:

```
const int N = ...; // # of threads in the system.  
int turn = 0; // Thread 0 gets its turn first.  
  
void EnterCriticalSection(int i)  
{  
    while (turn != i) /* busy wait */ ;  
    // Someone gave us the turn... we own the region.  
}  
  
void LeaveCriticalSection(int i)  
{  
    // Give the turn to the next thread (possibly wrapping to 0).  
    turn = (i + 1) % N;  
}
```

This algorithm ensures mutual exclusion inside the critical region for precisely N concurrent threads. In this scheme, each thread is given a unique identifier in the range [0 . . . N), which is passed as the argument i to `EnterCriticalSection`. The turn variable indicates which thread is currently permitted to run inside the critical region, and when a thread tries to enter the critical region, it must wait for its turn to be granted by another thread, in this particular example by busy spinning. With this algorithm, we have to choose someone to be first, so we somewhat arbitrarily decide to give thread 0 its turn first by initializing turn to 0 at the outset. Upon leaving the region, each thread simply notifies the next thread that its turn has come up: it does this notification by setting turn, either wrapping it back around to 0, if we've reached the maximum number of threads, or by incrementing it by one otherwise.

There is one huge deal breaker with strict alternation: the decision to grant a thread entry to the critical region is not based in any part on the arrival of threads to the region. Instead, there is a predefined ordering: 0,

then 1, then . . . , then $N - 1$, then 0, and so on, which is nonnegotiable and always fixed. This is hardly fair and effectively means a thread that isn't currently in the critical region holds another thread from entering it. This can threaten the liveness of the system because threads must wait to enter the critical region even when there is no thread currently inside of it. This kind of "false contention" isn't a correctness problem per se, but reduces the performance and scalability of any use of it. This algorithm also only works if threads regularly enter and exit the region, since that's the only way to pass on the turn. Another problem, which we won't get to solving for another few pages, is that the critical region cannot accommodate a varying number of threads. It's quite rare to know *a priori* the number of threads a given region must serve, and even rarer for this number to stay fixed for the duration of a process's lifetime.

Dekker's and Dijkstra's Algorithms (1965). The first widely publicized general solution to the mutual exclusion problem, which did not require strict alternation, was a response submitted by a reader of a 1965 paper by E. W. Dijkstra in which he identified the mutual exclusion problem and called for solutions (see Further Reading, Dijkstra, 1965, Co-operating sequential processes). One particular reader, T. Dekker, submitted a response that met Dijkstra's criteria but that works only for two concurrent threads. It's referred to as "Dekker's algorithm" and was subsequently generalized in a paper by Dijkstra, also in 1965 (see Further Reading, Dijkstra, 1965, Solution of a problem in concurrent programming control), to accommodate N threads.

Dekker's solution works similar to strict alternation, in which turns are assigned, but extends this with the capability for each thread to note an interest in taking the critical region. If a thread desires the region but yet it isn't its turn to enter, it may "steal" the turn if the other thread has not also noted interest (i.e., isn't in the region).

In our sample implementation, we have a shared 2-element array of Booleans, `f1ags`, initialized to contain `false` values. A thread stores `true` into its respective element (index 0 for thread 0, 1 for thread 1) when it wishes to enter the region, and `false` as it exits. So long as only one thread wants to enter the region, it is permitted to do so. This works because a thread first writes into the shared `f1ags` array and then checks whether the

other thread has also stored into the `flags` array. We can be assured that if we write `true` into `flags` and then read `false` from the other thread's element that the other thread will see our `true` value. (Note that modern processors perform out of order reads and writes that actually break this assumption. We'll return to this topic later.)

We must deal with the case of both threads entering simultaneously. The tie is broken by using a shared `turn` variable, much like we saw earlier. Just as with strict alternation, when both threads wish to enter, a thread may only enter the critical region when it sees `turn` equal to its own index and that the other thread is no longer interested (i.e., its `flags` element is `false`). If a thread finds that both threads wish to enter but it's not its turn, the thread will "back off" and wait by setting its `flags` element to `false` and waiting for the `turn` to change. This lets the other thread enter the region. When a thread leaves the critical region, it just resets its `flags` element to `false` and changes the `turn`.

This entire algorithm is depicted in the following snippet.

```
static bool[] flags = new bool[2];
static int turn = 0;

void EnterCriticalSection(int i) // i will only ever be 0 or 1
{
    int j = 1 - i;           // the other thread's index
    flags[i] = true;        // note our interest
    while (flags[j])        // wait until the other is not interested
    {
        if (turn == j)      // not our turn, we must back off and wait
        {
            flags[i] = false;
            while (turn == j) /* busy wait */;
            flags[i] = true;
        }
    }
}

void LeaveCriticalSection(int i)
{
    turn = 1 - i;           // give away the turn
    flags[i] = false;       // and exit the region
}
```

Dijkstra's modification to this algorithm supports N threads. While it still requires N to be determined a priori, it does accommodate systems in

which fewer than N threads are active at any moment, which admittedly makes it much more practical.

The implementation is slightly different than Dekker's algorithm. We have a `flags` array of size N, but instead of Booleans it contains a tri-value. Each element can take on one of three values, and in our example, we will use an enumeration: `passive`, meaning the thread is uninterested in the region at this time; `requesting`, meaning the thread is attempting to enter the region; and `active`, which means the thread is currently executing inside of the region.

A thread, upon arriving at the region, notes interest by setting its flag to `requesting`. It then attempts to "steal" the current turn: if the current turn is assigned to a thread that isn't interested in the region, the arriving thread will set `turn` to its own index. Once the thread has stolen the turn, it notes that it is actively in the region. Before actually moving on, however, the thread must verify that no other thread has stolen the turn in the meantime and possibly already entered the region, or we could break mutual exclusion. This is verified by ensuring that no other thread's flag is `active`. If another active thread is found, the arriving thread will back off and go back to a `requesting` state, continuing the process until it is able to enter the region. When a thread leaves the region, it simply sets its flag to `passive`.

Here is a sample implementation in C#.

```
const int N = ...; // # of threads that can enter the region.

enum F : int
{
    Passive,
    Requesting,
    Active
}

F[] flags = new F[N]; // all initialized to passive
int turn = 0;

void EnterCriticalSection(int i)
{
    int j;
    do
    {
```

```

flags[i] = F.Requesting; // note our interest

while (turn != i)           // spin until it's our turn
    if (flags[turn] == F.Passive)
        turn = i;             // steal the turn

flags[i] = F.Active;        // announce we're entering

// Verify that no other thread has entered the region.
for (j = 0;
     j < N && (j == i || flags[j] != F.Active);
     j++);
}
while (j < N);
}

void LeaveCriticalSection(int i)
{
    flags[i] = F.Passive;      // just note we've left
}

```

Note that just as with Dekker's algorithm as written above this code will not work as written on modern compilers and processors due to the high likelihood of out of order execution. This code is meant to illustrate the logical sequence of steps only.

Peterson's Algorithm (1981). Some 16 years after the original Dekker algorithm was published, a simplified algorithm was developed by G. L. Peterson and detailed in his provocatively titled paper, "Myths about the Mutual Exclusion" (see Further Reading, Peterson). It is simply referred to as Peterson's algorithm. In fewer than two pages, he showed a two thread algorithm alongside a slightly more complicated N thread version of his algorithm, both of which were simpler than the 15 years of previous efforts to simplify Dekker and Dijkstra's original proposals.

For brevity's sake, we review just the two thread version here. The shared variables are the same, that is, a `flags` array and a `turn` variable, as in Dekker's algorithm. Unlike Dekker's algorithm, however, a requesting thread immediately gives away the turn to the other thread after setting its `flags` element to true. The requesting thread then waits until either the other thread is not in its critical region or until the turn has been given back to the requesting thread.

```

bool[] flags = new bool[2];
int turn = 0;

void EnterCriticalSection(int i)
{
    flags[i] = true; // note our interest in the region
    turn = 1 - i;    // give the turn away

    // Wait until the region is available or it's our turn.
    while (flags[1 - i] && turn != i) /* busy wait */ ;

}

void LeaveCriticalSection(int i)
{
    flags[i] = false; // just exit the region
}

```

Peterson's algorithm, just like Dekker's, also satisfies all of the basic mutual exclusion, fairness, and liveness properties outlined above. It is also much simpler, and so it tends to be used more frequently over Dekker's algorithm to teach mutual exclusion.

Lamport's Bakery Algorithm (1974). L. Lamport also proposed an alternative algorithm, and called it the Baker's algorithm (see Further Reading, Lamport, 1974). This algorithm nicely accommodates varying numbers of threads, but has the added benefit that the failure of one thread midway through executing the critical region entrance or exit code does not destroy liveness of the system, as is the case with the other algorithms seen so far. All that is required is the thread must reset its ticket number to 0 and move to its noncritical region. Lamport was interested in applying his algorithm to distributed systems in which such fault tolerance was obviously a critical component of any viable algorithm.

The algorithm is called the “bakery” algorithm because it works a bit like your neighborhood bakery. When a thread arrives, it takes a ticket number, and only when its ticket number is called (or more precisely, those threads with lower ticket numbers have been serviced) will it be permitted to enter the critical region. The implementation properly deals with the edge case in which multiple threads happen to be assigned the same ticket number by using an ordering among the threads themselves—for example, a unique thread identifier, name, or some other comparable property—to break the tie. Here is a sample implementation.

```

const int N = ...; // # of threads that can enter the region.
int[] choosing = new int[N];
int[] number   = new int[N];

void EnterCriticalSection(int i)
{
    // Let others know we are choosing a ticket number.
    // Then find the max current ticket number and add one.
    choosing[i] = 1;
    int m = 0;
    for (int j = 0; j < N; j++)
    {
        int jn = number[j];
        m = jn > m ? jn : m;
    }
    number[i] = 1 + m;
    choosing[i] = 0;

    for (int j = 0; j < N; j++)
    {
        // Wait for threads to finish choosing.
        while (choosing[j] != 0) /* busy wait */ ;

        // Wait for those with lower tickets to finish. If we took
        // the same ticket number as another thread, the one with the
        // lowest ID gets to go first instead.
        int jn;
        while ((jn = number[j]) != 0 &&
               (jn < number[i] || (jn == number[i] && j < i)))
            /* busy wait */ ;
    }

    // Our ticket was called. Proceed to our region...
}

void LeaveCriticalSection(int i)
{
    number[i] = 0;
}

```

This algorithm is also unique when compared to previous efforts because threads are truly granted fair entrance into the region. Tickets are assigned on a first-come, first-served basis (FIFO), and this corresponds directly to the order in which threads enter the region.

Hardware Compare and Swap Instructions (Fast Forward to Present Day). Mutual exclusion has been the subject of quite a bit of research. It's easy to

take it all for granted given how ubiquitous and fundamental synchronization has become, but nevertheless you may be interested in some of the references to learn more than what's possible to describe in just a few pages (see Further Reading, Raynal).

Most of the techniques shown also share one thing in common. Aside from the bakery algorithm, each relies on the fact that reads and writes from and to natural word-sized locations in memory are atomic on all modern processors. But they specifically do not require atomic sequences of instructions in the hardware. These are truly "lock free" in the most literal sense of the phrase. However, most modern critical regions are not implemented using any of these techniques. Instead, they use intrinsic support supplied by the hardware.

One additional drawback of many of these software only algorithms is that one must know N in advance and that the space and time complexity of each algorithm depends on N. This can pose serious challenges in a system where any number of threads—a number that may only be known at runtime and may change over time—may try to enter the critical region. Windows and the CLR assign unique identifiers to all threads, but unfortunately these identifiers span the entire range of a 4-byte integer. Making N equal to 2^{32} would be rather absurd.

Modern hardware supports atomic **compare and swap (CAS)** instructions. These are supported in Win32 and the .NET Framework where they are called **interlocked** operations. (There are many related atomic instructions supported by the hardware. This includes an atomic bit-test-and-set instruction, for example, which can also be used to build critical regions. We'll explore these in more detail in Chapter 10, Memory Models and Lock Freedom.) Using a CAS instruction, software can load, compare, and conditionally store a value, all in one atomic, uninterruptible operation. This is supported in the hardware via a combination of CPU and memory subsystem support, differing in performance and complexity across different architectures.

Imagine we have a CAS API that takes three arguments: (1) a pointer to the address we are going to read and write, (2) the value we wish to place into this location, and (3) the value that must be in the location in

order for the operation to succeed. It returns true if the comparison succeeded—that is, if the value specified in (3) was found in location (1), and therefore the write of (2) succeeded—or false if the operation failed, meaning that the comparison revealed that the value in location (1) was not equal to (3). With such a CAS instruction in hand, we can use an algorithm similar to the first intuitive guess we gave at the beginning of this section:

```
int taken = 0;

void EnterCriticalSection()
{
    // Mark the region as taken.
    while (!CAS(&taken, 1, 0)) /* busy wait */ ;
}

void LeaveCriticalSection()
{
    taken = 0; // Mark the region as available.
}
```

A thread trying to enter the critical region continuously tries to write 1 into the taken variable, but only if it reads it as 0 first, atomically. Eventually the region will become free and the thread will succeed in writing the value. Only one thread can enter the region because the CAS operation guarantees that the load, compare, and store sequence is done completely atomically.

This implementation gives us a much simpler algorithm that happens to accommodate an unbounded number of threads, and does not require any form of alternation. It does not give any fairness guarantee or preference as to which thread is given the region next, although it could clearly be extended to do so. In fact, busy waiting indefinitely as shown here is usually a bad idea, and instead, true critical region primitives are often built on top of OS support for waiting, which does have some notion of fairness built in.

Most modern primitive synchronization primitives are built on top of CAS operations. Many other useful algorithms also can be built on top of CAS. For instance, returning to our earlier motivating data race, $(*a)++$, we

can use CAS to achieve a race-free and serializable program rather than using a first class critical region. For example:

```
void AtomicIncrement(int * p)
{
    int seen;
    do
    {
        seen = *p;
    }
    while (!CAS(p, seen + 1, seen));
}

// ... elsewhere ...

int a = 0;
AtomicIncrement(&a);
```

If another thread changes the value in location `p` in between the reading of it into the `seen` variable, the CAS operation will fail. The function responds to this failed CAS by just looping around and trying the increment again until the CAS succeeds. Just as with the lock above, there are no fairness guarantees. The thread trying to perform an increment can fail any number of times, but probabilistically it will eventually make forward progress.

The Harsh Reality of Reordering, Memory Models. The discussion leading up to this point has been fairly naïve. With all of the software-only examples of mutual exclusion algorithms above, there is a fundamental problem lurking within. Modern processors execute instructions out of order and modern compilers perform sophisticated optimizations that can introduce, delete, or reorder reads and writes. Reference has already been made to this point. But if you try to write and use a critical region as I've shown, it will likely not work as expected. The hardware-based version (with CAS instructions) will typically work on modern processors because CAS guarantees a certain level of read and write reordering safety.

Here are a few concrete examples where the other algorithms can go wrong.

- In the original strict alternation algorithm, we use a loop that continually rereads `turn`, waiting for it to become equal to the thread's

index `i`. Because `turn` is not written in the body of the loop, a compiler may conclude that `turn` is loop invariant and thus hoist the read into a temporary variable before the loop even begins. This will lead to an infinite loop for threads trying to enter a busy critical region. Moreover, a compiler may only do this under some conditions, like when non debug optimizations are enabled. This same problem is present in each of the algorithms shown.

- Dekker's algorithm fundamentally demands that a thread's write to its flags entry happens before the read of its partner's flags variable. If this were not the case, both could read each other's flags variable as false and proceed into the critical region, breaking the mutual exclusion guarantee. This reordering is legal and quite common on all modern processors, rendering this algorithm invalid. Similar requirements are present for many of the reads and writes within the body of the critical region acquisition sequence.
- Critical regions typically have the effect of communicating data written inside the critical region to other threads that will subsequently read the data from inside the critical region. For instance, our earlier example showed each thread executing `a++`. We assumed that surrounding this with a critical region meant that a thread, `t2`, running later in time than another thread, `t1`, would always read the value written by `t1`, resulting in the correct final value. But it's legal for code motion optimizations in the compiler to move reads and writes outside of the critical regions shown above. This breaks concurrency safety and exposes the data race once again. Similarly, modern processors can execute individual reads and writes out of order, and modern cache systems can give the appearance that reads and writes occurred out of order (based on what memory operations are satisfied by what level of the cache).

Each of these issues invalidates one or more of the requirements we sought to achieve at the outset. All modern processors, compilers, and runtimes specify which of these optimizations and reorderings are legal and, most importantly, which are not, through a **memory model**. These guarantees can, in principal, then be relied on to write a correct implementation

of a critical region, though it's highly unlikely anybody reading this book will have to take on such a thread. The guarantees vary from compiler to compiler and from one processor to the next (when the compiler's guarantees are weaker than the processor's guarantees), making it extraordinarily difficult to write correct code that runs everywhere.

Using one of the synchronization primitives from Win32 or the .NET Framework alleviates all need to understand memory models. Those primitives should be sufficient for 99.9 percent (or more) of the scenarios most programmers face. For the cases in which these primitives are not up to the thread—which is rare, but can be the case for efficiency reasons—or if you're simply fascinated by the topic, we will explore memory models and some lock free techniques in Chapter 10, Memory Models and Lock Freedom. If you thought that reasoning about program correctness and timings was tricky, just imagine if any of the reads and writes could happen in a randomized order and didn't correspond at all to the order in the program's source.

Coordination and Control Synchronization

If it's not obvious yet, interactions between components change substantially in a concurrent system. Once you have multiple things happening simultaneously, you will eventually need a way for those things to collaborate, either via centrally managed orchestration or autonomous and distributed interactions. In the simplest form, one thread might have to notify another when an important operation has just finished, such as a producer thread placing a new item into a shared buffer for which a consumer thread is waiting. More complicated examples are certainly commonplace, such as when a single thread must orchestrate the work of many subservient threads, feeding them data and instructions to make forward progress on a larger shared problem.

Unlike sequential programs, state transitions happen in parallel in concurrent programs and are thus more difficult to reason. It's not necessarily the fact that things are happening at once that makes concurrency difficult so much as getting the interactions between threads correct. Leslie Lamport said it very well:

We thought that concurrent systems needed new approaches because many things were happening at once. We have learned instead that . . . the

real leap is from functional to reactive systems. A functional system is one that can be thought of as mapping an input to an output. . . . A (reactive) system is one that interacts in more complex ways with its environment (see Further Reading, Lamport, 1993).

Earlier in this chapter, we saw how state can be shared in order to speed up communication between threads and the burden that implies. The patterns of communication present in real systems often build directly on top of such sharing. In the scenario with a producer thread and a consumer thread mentioned earlier, the consumer may have to wait for the producer to generate an item of interest. Once an item is available, it could be written to a shared memory location that the consumer directly accesses, using appropriate data synchronization to eliminate a class of concurrency hazards. But how does one go about orchestrating the more complex part: waiting, in the case that a consumer arrives before the producer has something of interest, and notification, in the case that a consumer has begun waiting by the time the producer creates that thing of interest? And how does one architect the system of interactions in the most efficient way? These are some topics we will touch on in this section.

Because thread coordination can take on many diverse forms and spans many specific implementation techniques, there are many details to address. As noted in the first chapter, there isn't any "one" correct way to write a concurrent program; instead, there are certain ways of structuring and writing programs that make one approach more appropriate than another. There are quite a few primitives in Win32 and the .NET Framework and design techniques from which to choose. For now we will focus on building a conceptual understanding of the approaches.

State Dependence Among Threads

As we described earlier, programs are comprised of big state machines that are traversed during execution. Threads themselves also are composed of smaller state machines that contribute to the overall state of the program itself. Each carries around some interesting data and performs some number of activities. An activity is just some abstract operation that possibly reads and writes the data and, in doing so, also possibly transitions between states, both local to the thread and global to the program. As we

already saw, some level of data synchronization often is needed to ensure invalid states are not reached during the execution of such activities.

It is also worth differentiating between internal and external states, for example, those that are just implementation details of the thread itself versus those that are meant to be observed by other threads running in a system, respectively.

Threads frequently have to interact with other threads running concurrently in the system to accomplish some work, forming a dependency. Once such a dependency exists, a dependent thread will typically have some knowledge of the (externally visible) states the depended-upon thread may transition between. It's even common for a thread to require that another thread is in a specific state before proceeding with an operation. A thread might only transition into such a state with the passing of time, as a result of external stimuli (like a GUI event or incoming network message), via some third thread running concurrently in the system producing some interesting state itself, or some combination of these. When one thread depends on another and is affected by its state changes (such as by reading memory that it has written), the thread is said to be **causally dependent** on the other.

Thinking about control synchronization in abstract terms is often helpful, even if the actual mechanism used is less formally defined. As an example, imagine that there is some set of states SP in which the predicate P will evaluate to true. A thread that requires P to be true before it proceeds is actually just waiting for any of the states in SP to arise. Evaluating the predicate P is really asking the question, "Is the program currently in any such state?" And if the answer is no, then the thread must do one of three things: (1) perform some set of reads and writes to transition the program from its current state to one of those in SP, (2) wait for another concurrent thread in the system to perform this activity' or (3) forget about the requirement and do something else instead.

The one example of waiting we've seen so far is that of a critical region. In the CAS based examples, a thread must wait for any state in which the taken variable is false to arise before proceeding to the critical region. Either it is already the case, or the thread trying to enter the region must wait for (2), another thread in the system to enable the state, via leaving the region.

Waiting for Something to Happen

We've encountered the topic of waiting a few times now. As just mentioned, a thread trying to enter a critical region that another thread is already actively running within must wait for it to leave. Many threads may simultaneously try to enter a busy critical region, but only one of them will be permitted to enter at a time. Similarly, control synchronization mechanisms require waiting, for example for an occurrence of an arbitrary event, some data of interest to become available, and so forth. Before moving on to the actual coordination techniques popular in the implementation of control synchronization, let's discuss how it works for a moment.

Busy Spin Waiting. Until now we've shown nothing but busy waiting (a.k.a. spin waiting). This is the simplest (and most inefficient) way to "wait" for some condition to become true, particularly in shared memory systems. With busy waiting, the thread simply sits in a loop reevaluating the predicate until it yields the desired answer, continuously rereading shared memory locations.

For instance, if P is some arbitrary Boolean predicate statement and S is some statement that must not execute until P is true, we might do this:

```
while (!P) /* busy wait */ ;  
S;
```

We say that statement S is guarded by the predicate P. This is an extremely common pattern in control synchronization. Elsewhere there will be a concurrent thread that makes P evaluate to true through a series of writes to shared memory.

Although this simple spin wait is sufficient to illustrate the behavior of our guarded region—allowing many code illustrations in this chapter that would have otherwise required an up-front overview of various other platform features—it has some serious problems.

Spinning consumes CPU cycles, meaning that the thread spinning will remain scheduled on the processor until its quantum expires or until some other thread preempts it. On a single processor machine, this is a complete waste because the thread that will make P true can't be run until the spinning thread is switched out. Even on a multiprocessor machine, spinning can lead to noticeable CPU spikes, in which it appears

as if some thread is doing real work and making forward progress, but the utilization is just caused by one thread waiting for another thread to run. And the thread remains runnable during the entire wait, meaning that other threads waiting to be scheduled (to perform real work) will have to wait in line behind the waiting thread, which is really not doing anything useful. Last, if evaluating P touches shared memory that is frequently accessed concurrently, continuously re-evaluating the predicate so often will have a negative effect on the performance of the memory system, both for the processor that is actually spinning and also for those doing useful work.

Not only is spin waiting inefficient, but the aggressive use of CPU cycles, memory accesses, and frequent bus communications all consume considerable amounts of power. On battery-powered devices, embedded electronics, and in other power constrained circumstances, a large amount of spinning can be downright annoying, reducing battery time to a fraction of its normal expected range, and it can waste money. Spinning can also increase heat in data centers, increasing air conditioning costs, making it attractive to keep CPU utilization far below 100 percent.

As a simple example of a problem with spinning, I'm sitting on an airplane as I write this paragraph. Moments ago, I was experimenting with various mutual exclusion algorithms that use busy waiting, of the kind we looked at above, when I noticed my battery had drained much more quickly than usual. Why was this so? I was continuously running test case after test case that made use of many threads using busy waits concurrently. At least I was able to preempt this problem. I just stopped running my test cases. But if the developers who created my word processor of choice had chosen to use a plethora of busy waits in the background spellchecking algorithm, it's probable that this particular word processor wouldn't be popular among those who write when traveling. Thankfully that doesn't appear to be the case.

Needless to say, we can do much better.

Real Waiting in the Operating System's Kernel. The Windows OS offers support for true waiting in the form of various kernel objects. There are two kinds of event objects, for example, that allow one thread to wait and have some other thread signal the event (waking the waiter[s]) at some point in

the future. There are other kinds of kernel objects, and they are used in the implementation of various other higher-level waiting primitives in Win32 and the .NET Framework. They are all described in Chapter 5, Windows Kernel Synchronization.

When a thread waits, it is put into a wait state (versus a runnable state), which triggers a context switch to remove it from the processor immediately, and ensures that the Windows thread scheduler will subsequently ignore it when considering which thread to run next. This avoids wasting CPU availability and power and permits other threads in the system to make forward progress. Imagine a fictional API `WaitSysCall` that allows threads to wait. Our busy wait loop from earlier might become something like this:

```
if (!P)
    WaitSysCall();
S;
```

Now instead of other threads simply making `P` true, the thread that makes `P` true must now take into consideration that other threads might be waiting. It then wakes them with a corresponding call to `WakeSysCall`.

```
Enable(P); // ... make P true ...
WakeSysCall();
```

You probably have picked up a negative outlook on busy waiting altogether. Busy waiting can be used (with care) to improve performance and scalability on multiprocessor machines, particularly for fine-grained concurrency. The reason is subtle, having to do with the cost of context switching, waiting, and waking. Getting it correct requires an intelligent combination of both spinning and true waiting. There are also some architecture specific considerations that you will need to make. (If it's not obvious by now, the spin wait as written above is apt to cause you many problems, so please don't try to use it.) We will explore this topic in Chapter 14, Performance and Scalability.

Continuation Passing as an Alternative to Waiting. Sometimes it's advantageous to avoid waiting altogether. This is for a number of reasons, including avoiding the costs associated with blocking a Windows thread.

But perhaps more fundamentally, waiting can present scheduling challenges. If many threads wait and are awoken nearly simultaneously, they will contend for resources. The details depend heavily on the way in which threads are mapped to threads in your system of choice.

As an alternative to waiting, it is often possible to use continuation passing style (CPS), a popular technique in functional programming environments (see Further Reading, Hoare, 1974). A continuation is an executable closure that represents “the rest” of the computation. Instead of waiting for an event to happen, it is sometimes possible to package up the response to that computation in the form of a closure and to pass it to some API that then assumes responsibility for scheduling the continuation again when the wait condition has been satisfied.

Because neither Windows nor the CLR offers first-class support for continuations, CPS can be difficult to achieve in practice. As we’ll see in Chapter 8, Asynchronous Programming Models, the .NET Framework’s asynchronous programming model offers a way to pass a delegate to be scheduled in response to an activity completing, as do the Windows and CLR thread pools and various other components. In each case, it’s the responsibility of the user of the API to deal with the fact that the remainder of the computation involves a possibly deep callstack at the time of the call. Transforming “the rest” of the computation is, therefore, difficult to do and is ordinarily only a reasonable strategy for applications level programming where components are not reused in various settings.

A Simple Wait Abstraction: Events

The most basic control synchronization primitive is the event, also sometimes referred to as a latch, which is a concrete reification of our fictional `WaitSysCall` and `WakeSysCall` functions shown above. Events are a flexible waiting and notification mechanism that threads can use to coordinate among one another in a less-structured and free-form manner when compared to critical regions and semaphores. Additionally, there can be many such events in a program to wait and signal different interesting circumstances, much like there can be multiple critical regions to protect different portions of shared state.

An event can be in one of two states at a given time: **signaled** or **nonsignaled**. If a thread waits on a nonsignaled event, it does not proceed until the event becomes signaled; otherwise, the thread proceeds right away. Various kinds of events are commonplace, including those that stay signaled permanently (until manually reset to nonsignaled), those that automatically reset back to the nonsignaled state after a single thread waits on it, and so on. In subsequent chapters, we will look at the actual event primitives available to you.

To continue with the previous example of guarding a region of code by some arbitrary predicate P, imagine we have a thread that checks P and, if it is not true, wishes to wait. We can use an event E that is signaled when P is enabled and nonsignaled when it is not. That event internally uses whatever waiting mechanism is most appropriate, most likely involving some amount of spinning plus true OS waiting. Threads enabling and disabling P must take care to ensure that E's state mirrors P correctly.

```
// Consuming thread:  
if (!P)  
    E.Wait();  
S;  
  
// Enabling thread:  
Enable(P); // ... make P true ...  
E.Set();
```

If it is possible for P to subsequently become false in this example and the event is not automatically reset, we must also allow a thread to reset the event.

```
E.Reset();  
Disable(P); // ... make P false ...
```

Each kind of event may reasonably implement different policies for waiting and signaling. One event may decide to wake all waiting threads, while another might decide to wake one and automatically put the event back into a nonsignaled state afterward. Yet another technique may wait for a certain number of calls to Set before waking up any waiters.

As we'll see, there are some tricky race conditions in all of these examples that we will have to address. For events that stay signaled or have some degree of synchronization built in, you can get away without extra data synchronization, but most control synchronization situations are not quite so simple.

One Step Further: Monitors and Condition Variables

Although events are a general purpose and flexible construct, the pattern of usage shown here is very common, for example to implement guarded regions. In other words, some event E being signaled represents some interesting program condition, namely some related predicate P being true, and thus the event state mirrors P's state accordingly. To accomplish this reliably, data and control synchronization often are needed together. For instance, the evaluation of the predicate P may depend on shared state, in which case data synchronization is required during its evaluation to ensure safety. Moreover, there are data races, mentioned earlier, that we need to handle. Imagine we support setting and resetting; we must avoid the problematic timing of:

```
t1: Enable(P) -> t2: E.Reset() -> t2: Disable(P) -> t1: E.Set()
```

In this example, t1 enables the predicate P, but before it has a chance to set the event, t2 comes along and disables P. The result is that we wake up waiting threads although P is no longer true. These threads must take care to re-evaluate P after being awakened to avoid proceeding blindly. But unless they use additional data synchronization, this is impossible.

A nice codification of this relationship between state transitions and data and control synchronization was invented in the 1970s (see Further Reading, Hansen; Hoare, 1974) and is called **monitors**. Each monitor implicitly has a critical region and may have one or more **condition variables** associated with it, each representing some condition (like P evaluating to true) for which threads may wish to wait. In this sense, a condition variable is just a fancy kind of event.

All waiting and signaling of a monitor's condition variables must occur within the critical region of the monitor itself, ensuring data race protection. When a thread decides to wait on a condition variable, it implicitly releases

ownership of the monitor (i.e., leaves the critical region), waits, and then reacquires it immediately after being woken up by another thread. This release-wait sequence is done such that other threads entering the monitor are not permitted to enter until the releaser has made it known that it is waiting (avoiding the aforementioned data races). There are also usually mechanisms offered to either wake just one waiting thread or all waiting threads when signaling a condition variable.

Keeping with our earlier example, we may wish to enable threads to wait for some arbitrary predicate P to become true. We could represent this with some monitor M (with methods Enter and Leave) and a condition variable CV (with methods Wait and Set) to represent the condition in which a state transition is made that enables P. (We could have any number of predicates and associated condition variables for M, but our example happens to use only one.) Our example above, which used events, now may look something like this:

```
// Consuming thread:  
M.Enter();  
while (!P)  
    CV.Wait();  
M.Leave();  
S; // (or inside the monitor, depending on its contents)  
  
// Enabling thread:  
M.Enter();  
Enable(P);  
CV.Set();  
M.Leave();  
  
// Disabling thread:  
M.Enter();  
Disable(P);  
M.Leave();
```

Notice in this example that the thread that disables P has no additional requirements because it does so within the critical region. The next thread that is granted access to the monitor will re-evaluate P and notice that it has become false, causing it to wait on CV. There is something subtle in this program. The consuming thread continually re-evaluates P in a while loop, waiting whenever it sees that it is false. This re-evaluation is necessary to

avoid the case where a thread enables P, setting CV, but where another thread “sneaks in” and disables P before the consuming thread has a chance to enter the monitor. There is generally no guarantee, just because the condition variable on which a thread was waiting has become signaled, that such a thread is the next one to enter the monitor’s critical region.

Structured Parallelism

Some parallel constructs hide concurrency coordination altogether, so that programs that use them do not need to concern themselves with the low-level events, condition variables, and associated coordination challenges. The most compelling example is data parallelism, where partitioning of the work is driven completely by data layout. The term structured parallelism is used to refer to such parallelism, which typically has well-defined begin and end points.

Some examples of structured parallel constructs follow.

- **Cobegin**, normally takes the form of a block in which each of the contained program statements may execute concurrently. An alternative is an API that accepts an array of function pointers or delegates. The cobegin statement spawns threads to run statements in parallel and returns only once all of these threads have finished, hiding all coordination behind a clean abstraction.
- **Forall**, a.k.a. parallel do loops, in which all iterations of a loop body can run concurrently with one another on separate threads. The statement following the loop itself runs only once all concurrent iterations have finished executing.
- **Futures**, in which some value is bound to a computation that may happen at an unspecified point in the future. The computation may run concurrently, and consumers of the future’s value can choose to wait for the value to be computed, without having to know that waiting and control synchronization is involved.

The languages on Windows and the .NET Framework currently do not offer direct support for these constructs, but we will build up a library of them in Chapters 12, Parallel Containers and 13, Data and Task Parallelism.

This library enables higher level concurrent programs to be built with more ease. Appendix B, Parallel Extensions to .NET, also takes a look at the future of concurrency APIs on .NET which contains similar constructs.

Message Passing

In shared memory systems—the dominant concurrent programming model on Microsoft’s development platform (including native Win32 and the CLR)—there is no apparent distinction in the programming interface between state that is used to communicate between threads and state that is thread local. The language and library constructs to work with these two very different categories of memory are identical. At the same time, reads from and writes to shared state usually mean very different things than those that work with thread-private state: they are usually meant to instruct concurrent threads about the state of the system so they can react to the state change. The fact that it is difficult to identify operations that work with this special case also makes it difficult to identify where synchronization is required and, hence, to reason about the subtle interactions among concurrent threads.

In message passing systems, all interthread state sharing is encapsulated within the messages sent between threads. This typically requires that state is copied when messages are sent and normally implies handing off ownership of state at the messaging boundary. Logically, at least, this is the same as performing atomic updates in a shared memory system, but is physically quite different. (In fact, using shared memory could be viewed as an optimization for message passing, when it can be proven safe to turn message sends into writes to shared memory. Recent research in operating system design in fact has explored using such techniques [see Further Reading, Aiken, Fahndrich, Hawblitzel, Hunt, Larus].) Due to the copying, message passing in most implementations is less efficient from a performance standpoint. But the overall thread of state management is usually simplified.

The first popular message passing system was proposed by C. A. R. Hoare as his Communicating Sequential Processes (CSP) research (see Further Reading, Hoare, 1978, 1985). In a CSP system, all concurrency is achieved by having independent processes running asynchronously. As they must

interact, they send messages to one another, to request or to provide information to one another. Various primitives are supplied to encourage certain communication constructs and patterns, such as interleaving results among many processes, waiting for one of many to produce data of interest, and so on. Using a system like CSP appreciably raises the level of abstraction from thinking about shared memory and informal state transitions to independent actors that communicate through well-defined interfaces.

The CSP idea has shown up in many subsequent systems. In the 1980s, actor languages evolved the ideas from CSP, mostly in the context of LISP and Scheme, for the purpose of supporting richer AI programming such as in the Act1 and Act2 systems (see Further Reading, Lieberman). It turns out that modeling agents in an AI system as independent processes that communicate through messages is not only a convenient way of implementing a system, but also leads to increased parallelism that is bounded only by the number of independent agents running at once and their communication dependencies. Actors in such a system also sometimes are called “active objects” because they are usually ordinary objects but use CSP-like techniques transparently for function calls. The futures abstraction mentioned earlier is also typically used pervasively. Over time, programming systems like Ada and Erlang (see Further Reading, Armstrong) have pushed the envelope of message passing, incrementally pushing more and more usage from academia into industry.

Many CSP-like concurrency facilities have been modeled mathematically. This has subsequently led to the development of the pi-calculus, among others, to formalize the notion of independently communicating agents. This has taken the form of a calculus, which has had recent uses outside of the domain of computer science (see Further Reading, Sangiorgi, Walker).

Windows and the .NET Framework offer only limited support for fine-grained message passing. CLR AppDomains can be used for fine-grained isolation, possibly using CLR Remoting to communicate between objects in separate domains. But the programming model is not nearly as nice as the aforementioned systems in which message passing is first class. Distributed programming systems such as Windows Communication Foundation (WCF) offer message passing support, but are more broadly used for coarse-grained parallel communication. The Coordination and Concurrency

Runtime (CCR), downloadable as part of Microsoft's Robotics SDK (available on MSDN), offers fine-grained message as a first-class construct in the programming model.

As noted in Chapter 1, Introduction, the ideal architecture for building concurrent systems demands a hybrid approach. At a coarse-grain, asynchronous agents are isolated and communicate in a mostly loosely coupled fashion; message passing is great for this. Then at a fine-grain, parallel computations share memory and use data and task parallel techniques.

Where Are We?

In this chapter, we've covered a fair bit of material. We first built up a good understanding of synchronization and time as they relate to concurrent programming and many related topics. Synchronization is important and relevant to all kinds of concurrent programming, no matter whether it is performance or responsiveness motivated, in the form of fine- or coarse-grained concurrency, shared-memory or message-passing based, written in native or managed code, and so on.

Although we haven't yet experimented with enough real mechanisms to build a concurrent program, we're well on our way. The following section, Mechanisms, spans seven chapters and focuses on the building blocks you'll use to build native and managed concurrent Windows programs. We'll start with the schedulable unit of concurrency on Windows: threads.

FURTHER READING

- M. Aiken, M. Fahndrich, C. Hawblitzel, G. Hunt, J. R. Larus. Deconstructing Process Isolation. *Microsoft Research Technical Report*, MSR-TR-2006-43 (2006).
- J. Armstrong. *Programming Erlang: Software for a Concurrent World* (The Pragmatic Programmers, 2007).
- C. Boyapati, B. Liskov, L. Shrira. Ownership Types for Object Encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)* (2003).
- P. Brinch Hansen. Structured Multiprogramming. *Communications of the ACM*, Vol. 15, No. 7 (1972).

- J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, S. Midkiff. Escape Analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (1999).
- E. W. Dijkstra. Co-operating Sequential Processes. In *Programming Languages* (Academic Press, 1965).
- E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, Vol. 8, No. 9 (1965).
- F. Drexhammar, C. Schulte. Implementation Strategies for Single Assignment Variables. *Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS)* (2004).
- R. H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 7, Issue 4 (1985).
- M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. In *ACM Transactions on Programming Languages and Systems*, 12 (3) (1990).
- R. Hieb, R. Kent Dybvig. Continuations and Concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1990).
- C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, Vol. 17, No. 10 (1974).
- C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, Vol. 21, No. 8 (1978).
- C. A. R. Hoare. *Communicating Sequential Processes* (Prentice Hall, 1985).
- C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., M. E. Zosel. *The High Performance FORTRAN Handbook* (MIT Press, 1994).
- L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, Vol. 17, No. 8 (1974).
- L. Lamport. Verification and Specification of Concurrent Programs. *A Decade of Concurrency: Reflections and Perspectives, Lecture Notes in Computer Science*, Number 803 (1993).
- H. Lieberman. Concurrent Object-oriented Programming in Act 1. *Object-oriented Concurrent Programming* (MIT Press, 1987).

- G. L. Peterson. Myths About the Mutual Exclusion Problem. *Inf. Proc. Lett.*, 12, 115–116 (1981).
- M. Raynal. *Algorithms for Mutual Exclusion* (MIT Press, 1986).
- D. Sangiorgi, D. Walker. *The Pi-Calculus: A Theory of Mobile Processes* (Cambridge University Press, 2003).
- N. Shavit, D. Touitou. Software Transactional Memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (1995).
- B. Stroustrup. *The C++ Programming Language*, Third Edition (Addison-Wesley, 1997).

PART II

Mechanisms

3

Threads

INDIVIDUAL PROCESSES ON Windows are sequential by default. Even on a multiprocessor machine, a program (by default) will only use one of them at a time. Running multiple processes at once creates concurrency at a very coarse level. Microsoft Word could be repaginating a document on one processor, while Internet Explorer downloads and renders a Web page on another, all while Windows Indexer is rebuilding search indexes on a third processor. This happens because each application is run inside its own distinct process with (one hopes) little interference between the two (again, one hopes), yielding better responsiveness and overall performance by virtue of running completely concurrently with one another.

The programs running inside of each process, however, are free to introduce additional concurrency. This is done by creating *threads* to run different parts of the program running inside a single program at once. Each Windows process is actually comprised of a single thread by default, but creating more than one in a program enables the OS to schedule many onto separate processors simultaneously. Coincidentally, each .NET program is actually multithreaded from the start because the CLR garbage collector uses a separate *finalizer* thread to reclaim resources. As a developer, you are free to create as many additional threads as you want.

Using multiple threads for a single program can be done to run entirely independent parts of a program at once. This is classic **agents style** concurrency and, historically, has been used frequently in server-side

programs. Or, you can use threads to break one big task into multiple smaller pieces that can execute concurrently. This is **parallelism** and is increasingly important as commodity hardware continues to increase the number of available processors. Refer back to Chapter 1, Introduction, for a detailed explanation of this taxonomy.

Threads are the fundamental units of schedulable concurrency on the Windows platform and are available to native and managed code alike. This chapter takes a look at the essentials of scheduling and managing concurrency on Windows using threads. The APIs used to access threading in native and managed code are slightly different, but the fundamental architecture and OS support are the same. But before we go into the details, let's precisely define what a thread is and of what it consists. After that, we'll move on to how programs use them.

Threading from 10,001 Feet

A thread is in some sense just a virtual processor. Each runs some program's code as though it were independent from all other virtual processors in the system. There can be fewer, equal, or more threads than real processors on a system at any given moment due (in part) to the multi-tasking nature of Windows, wherein a user can run many programs at once, and the OS ensures that all such threads get a fair chance at running on the available hardware.

Given that this could be as much a simple definition of an OS process as a thread, clearly there has to be some interesting difference. And there is (on Windows, at least). Processes are the fundamental unit of concurrency on many UNIX OSs because they are generally lighter-weight than Windows processes. A Windows process always consists of at least one thread that runs the program code itself. But one process also may execute multiple threads during the course of its lifetime, each of which shares access to a set of process-wide resources. In short, having many threads in a single process allows one process to do many things at once. The resources shared among threads include a single virtual memory address space, permitting threads to share data and communicate easily by reading from and writing to common addresses and objects in memory. Shared resources also include

things associated with the Windows process, such as the handle table and security token information.

Most people get their first taste of threading by accident. Developers use a framework such as ASP.NET that calls their code on multiple threads simultaneously or write some GUI event code in Windows Forms, MFC, or Windows Presentation Foundation, in which there is a strong notion of particular data structures belonging to particular threads. (We discuss this fact and its implications in Chapter 16, Graphical User Interfaces.) These developers often learn about concurrency “the hard way” by accidentally writing unreliable code that crashes or by creating an unresponsive GUI by doing I/O on the GUI thread. Faced with such a situation, people are quick to learn some basic rules of thumb, often without deeply understanding the reasons behind them. This can give people a bad first impression of threads. But while concurrency is certainly difficult, threads are the key to exploiting new hardware, and so it’s important to develop a deeper understanding.

What Is a Windows Thread?

We already discussed threads at a high level in previous chapters, but let’s begin painting a more detailed picture.

Conceptually speaking, a thread is an execution context that represents in-progress work being performed by a program. A thread isn’t a simple, physical thing. Windows must allocate and maintain a kernel object for each thread, along with a set of auxiliary data structures. But as a thread executes, some portion of its logical state is also comprised of hardware state, such as data in the processor’s registers. A thread’s state is, therefore, distributed among software and hardware, at least when it’s running. Given a thread that is running, a processor can continue running it, and given a thread that is not running, the OS has all the information it needs so that it can schedule the thread to run on the hardware again.

Each thread is mapped onto a processor by the Windows thread scheduler, enabling the in-progress work to actually execute. Each thread has an instruction pointer (IP) that refers to the current executing instruction. “Execution” consists of the processor fetching the next instruction, decoding it, and issuing it, one instruction after another, from the thread’s code,

incrementing the IP after ordinary instructions or adjusting it in other ways as branches and function calls occur. During the execution of some compiled code, program data will be routinely moved into and out of registers from the attached main memory. While these registers physically reside on the processor, some of this volatile state also abstractly belongs to the thread too. If the thread must be paused for any reason, this state will be captured and saved in memory so it can be later restored. Doing this enables the same IP fetch, decode, and issue process to proceed for the thread later as though it were never interrupted. The process of saving or restoring this state from and to the hardware is called a **context switch**.

During a context switch, the volatile processor state, which logically belongs to the thread, is saved in something called a **context**. The context switching behavior is performed entirely by the OS kernel, although the context data structure is available to user-mode in the form of a CONTEXT structure. Similarly, when the thread is rescheduled onto a processor, this state must be restored so the processor can begin fetching and executing the thread's instructions again. We'll look at this process in more detail later. Note that contexts arise in a few other places too. For example, when an exception occurs, the OS takes a snapshot of the current context so that exception handling code can inspect the IP and other state when determining how to react. Contexts are also useful when writing debugging and diagnostics tools.

As the processor invokes various function call instructions, a region of memory called the **stack** is used to pass arguments from the caller to the callee (i.e., the function being called), to allocate local variables, to save register values, and to capture return addresses and values. Code on a thread can allocate and store arbitrary data on the stack too. Each thread, therefore, has its own region of stack memory in the process's virtual address space. In truth, each thread actually has two stacks: a user-mode and a kernel-mode stack. Which gets used depends on whether the thread is actively running code in user- or kernel-mode, respectively. Each thread has a well-defined lifetime. When a new process is created, Windows also creates a thread that begins executing that process's entry-point code. A process doesn't execute anything, its threads do. After the magic of a process's first thread being created—handled by the OS's process creation routine—any

code inside that process can go ahead and create additional threads. Various system services create threads without you being involved, such as the CLR's garbage collector. When a new thread is created, the OS is told what code to begin executing and away it goes: it handles the bookkeeping, setting the processor's IP, and the code is then subsequently free to create additional threads, and so on.

Eventually a thread will exit. This can happen in a variety of ways—all of which we'll examine soon—including simply returning from the entry-point used to begin the thread's life an unhandled exception, or directly calling one of the platform's thread termination APIs.

The Windows thread scheduler takes care of tracking all of the threads in the system and working with the processor(s) to schedule execution of them. Once a thread has been created, it is placed into a queue of runnable threads and the scheduler will eventually let it run, though perhaps not right away, depending on system load. Windows uses preemptive scheduling for threads, which allows it to forcibly stop a thread from running on a certain processor in order to run some other code when appropriate. Pre-emption causes a context switch, as explained previously. This happens when a higher priority thread becomes runnable or after a certain period of time (called a **quantum** or a **timeslice**) has elapsed. In either case, the switch only occurs if there aren't enough processors to accommodate both threads in question running simultaneously; the scheduler will always prefer to fully utilize the processors available.

Threads can **block** for a number of reasons: explicit I/O, a hard page fault (i.e., caused by reading or writing virtual memory that has been paged out to disk by the OS), or by using one of the many synchronization primitives detailed in Chapters 5, Windows Kernel Synchronization and 6, Data and Control Synchronization. While a thread blocks, it consumes no processor time or power, allowing other runnable threads to make forward progress in its stead. The act of blocking, as you might imagine, modifies the thread data structure so that the OS thread scheduler knows it has become ineligible for execution and then triggers a context switch. When the condition that unblocks the thread arises, it becomes eligible for execution again, which places it back into the queue of runnable threads, and the scheduler will later schedule it to run using its ordinary thread scheduling

algorithms. Sometimes awakened threads are given priority to run again, something called a **priority boost**, particularly if the thread has awakened in response to a GUI event such as a button click. This topic will come up again later.

There are five basic mechanisms in Windows that routinely cause non-local transfer of control to occur. That is to say, a processor's IP jumps somewhere very different from what the program code would suggest should happen. The first is a context switch, which we've already seen. The second is exception handling. An exception causes the OS to run various exception filters and handlers in the context of the current executing thread, and, if a handler is found, the IP ends up inside of it.

The next mechanism that causes nonlocal transfer of control is the hardware interrupt. An interrupt occurs when a significant hardware event of interest occurs, like some device I/O completing, a timer expiring, etc., and provides an interrupt dispatch routine the chance to respond. In fact, we've already seen an example of this: preemption based context switches are initiated from a timer based interrupt. While an interrupt borrows the currently executing thread's kernel-mode stack, this is usually not noticeable: the code that runs typically does a small amount of work very quickly and won't run user-mode code at all.

(For what it's worth, in the initial SMP versions of Windows NT, all interrupts ran on processor number 0 instead of on the processor executing the affected thread. This was obviously a scalability bottleneck and required large amounts of interprocessor communication and was remedied for Windows 2000. But I've been surprised by how many people still believe this is how interrupt handling on Windows works, which is why I mention it here.)

Software based interrupts are commonly used in kernel and system code too, bringing us to the fourth and fifth methods: deferred procedure calls (DPCs) and asynchronous procedure calls (APCs). A DPC is just some callback that the OS kernel queues to run later on. DPCs run at a higher Interrupt Request Level (IRQL) than hardware interrupts, which simply means they do not hold up the execution of other higher priority hardware based interrupts should one happen in the middle of the DPC running. If anything meaty has to occur during a hardware interrupt, it usually gets

done by the interrupt handler queuing a DPC to execute the hard work, which is guaranteed to run before the thread returns back to user-mode. In fact, this is how preemption based context switches occur. An APC is similar, but can execute user-mode callbacks and only run when the thread has no other useful work to do, indicated by the thread entering something called an **alertable wait**. When, specifically, the thread will perform an alertable wait is unknowable, and it may never occur. Therefore, APCs are normally used for less critical and less time sensitive work, or for cases in which performing an alertable wait is a necessary part of the programming model that users program against. Since APCs also can be queued programmatically from user-mode, we'll return to this topic in Chapter 5, Windows Kernel Synchronization. Both DPCs and APCs can be scheduled across processors to run asynchronously and always run in the context of whatever the thread is doing at the time they execute.

Threads have a plethora of other interesting aspects that we'll examine throughout this chapter and the rest of the book, such as priorities, thread local storage, and a lot of API surface area. Each thread belongs to a single process that has other interesting and relevant data shared among all of its threads—such as the handle table and a virtual memory page table—but the above definition gives us a good roadmap for exploring at a deeper level.

Before all of that, let's review what makes a managed CLR thread different from a native thread. It's a question that comes up time and time again.

What Is a CLR Thread?

A CLR thread is the same thing as a Windows thread—usually. Why, then, is it popular to refer to CLR threads as “managed threads,” a very official term that makes them sound entirely different from Windows threads? The answer is somewhat complicated. At the simplest level, it effectively changes nothing for developers writing concurrent software that will run on the CLR. You can think of a thread running managed code as precisely the same thing as a thread running native code, as described above. They really aren't fundamentally different except for some esoteric and exotic situations that are more theoretical than practical.

First, the pragmatic difference: the CLR needs to track each thread that has ever run managed code in order for the CLR to do certain important jobs. The state associated with a Windows thread isn't sufficient. For example, the CLR needs to know about the object references that are live so that the garbage collector can determine which objects in the heap are still live. It does this in part by storing additional per-thread information such as how to find arguments and local variables on the stack. The CLR keeps other information on each managed thread, like event kernel objects that it uses for its own internal synchronization purposes, security, and execution context information, etc. All of these are simply implementation details.

Since the OS doesn't know anything about managed threads, the CLR has to convert OS threads to managed threads, which really just populates the thread's CLR-specific information. This happens in two places. When a new thread is created inside a managed program, it begins life as a managed thread (i.e., CLR-specific state is associated before it is even started). This is easy. If a thread already exists, however—that is it was created in native code and native-managed interoperability is being used—then the first time the thread runs managed code, the CLR will perform this conversion on-demand at the interoperability boundary.

Just to reiterate, all of this is transparent to you as a developer, so these points should make little difference. Knowing about them can come in useful, however, when understanding the CLR architecture and when debugging your programs.

Aside from that very down-to-earth explanation, the CLR has also decoupled itself from Windows threads from day one because there has always been the goal of allowing CLR hosts to override the default mapping of CLR threads directly to Windows threads. A CLR host, like SQL Server or ASP.NET, implements a set of interfaces, allowing it to override various policies, such as memory management, unhandled exception handling, reliability events of interest, and so on. (See Further Reading, Pratschner, for a more detailed overview of these capabilities.) One such overridable policy is the implementation of managed threads. When the CLR 2.0 was being developed, in fact, SQL Server 2005 experimented very seriously with mapping CLR threads to Windows fibers instead of threads, something they called **fiber-mode**. We'll explore in Chapter 9, Fibers, the

advantages fibers offer over threads, and how the CLR intended to support them. SQL Server has had a lot of experience in the past employing fiber based user-mode scheduling. We will also discuss a problem called **thread affinity**, which is related to all of this: a piece of work can take a dependency on the identity of the physical OS thread or can create a dependency between the thread and the work itself, which inhibits the platform's ability to decouple the CLR and Windows threads.

Just before shipping the CLR 2.0, the CLR and SQL Server teams decided to eliminate fiber-mode completely, so this whole explanation now has little practical significance other than as a possibly interesting historical account. But, of course, who knows what the future holds? User-mode scheduling offers some promising opportunities for building massively concurrent programs for massively parallel hardware, so the distinction between a CLR thread and a Windows thread may prove to be a useful one. That's really the only reason you might care about the distinction and why I labeled the concern "theoretical" at the outset.

Unless explicitly stated otherwise in the pages to follow, all of the discussions in this chapter pertain to behavior when run normally (i.e., no host) or inside a host that doesn't override the threading behavior. Trying to explain the myriad of possibilities simultaneously would be nearly impossible because the hosting APIs truly enable a large amount of the CLR's behavior to be extended and customized by a host.

Explicit Threading and Alternatives

We'll start our discussion about concurrency mechanisms at the bottom of the architectural stack with the Windows thread management facilities in Win32 and in the .NET Framework. This is called **explicit threading** in this book because you must be explicit about the creation and use of threads. This is a very low-level way to write concurrent software. Sometimes thinking at this low level is unavoidable, particularly for systems-level programming and, sometimes, also in application and library. Thinking about and managing threads is tricky and can quickly steal the focus from solving real algorithmic domain and business problems. You'll find that explicit threading quickly can become intrusive and pervasive in your program's architecture and implementation. Alternatives exist.

Thread pools abstract away the management of threads, amortizing the cost of creating and deleting them over the life of your process and optimizing the total number of threads to achieve superior all-around performance and scaling. Using a thread pool instead of explicit threading gets you away from thread management minutia and back to solving your business or domain problems. Most programmers can be very successful at concurrent programming without ever having to create a single thread by hand, thanks to carefully engineered Windows and CLR thread pool implementations.

Identifying patterns that emerge, abstracting them away, and hiding the use of threads and thread pools are also other useful techniques. It's common to layer systems so that most of the threading work is hidden inside of concrete components. A server program, for example, usually doesn't have any thread based code in callbacks; instead, there is a top-level processing loop that is responsible for moving work to run on threads. No matter what mechanisms you use, however, synchronization requirements are always pervasive unless alternative state management techniques (such as isolation) are employed.

Nevertheless, threads are a basic ingredient of life. Examining them in depth before looking at the abstractions that sit atop them will give you a better understanding of the core mechanisms in the OS, and from there, we can build up those (important and necessary) layers of abstraction without sacrificing knowledge of what underlies them. And perhaps you'll find yourself one day building such a layer of abstraction.

Last, a word of caution. Deciding precisely when it's a good idea to introduce additional threads is not as straightforward as you might imagine. Introducing too many can negatively impact your program's performance due to various fixed overheads and because the OS will spend increasingly more time trying to schedule them fairly as the ratio of threads to processors grows (we'll see details on this later). At the same time, introducing too few will lead to underutilized hardware and wasted opportunity. In some cases, the platform will help you create additional concurrency by using separate threads for some core system services (the CLR's ability to perform multi-threaded garbage collections is one example), but more often than not, it's left to you to decide and manage.

The Life and Death of Threads

As with most things, threads have a beginning and an end. Let's take a look at what causes the creation of a new thread, what causes the termination of an existing thread, and what precisely goes on during these two events. We'll also look at the `DllMain` method, which is a way for native code to receive notifications of thread creation and termination events.

Thread Creation

During the creation of a new process, Windows will automatically create a new thread to run the program's entry point code. That's typically your main function in your programming language of choice (i.e., `(w)main` in C++, `Main` in C#, and so forth). Without at least one thread, the process wouldn't be able to do anything because processes themselves don't execute code—threads do. Once the process has been bootstrapped, additional threads may be created by code run within the process itself by the mechanisms we're about to review.

Programmatically Creating Threads

When creating a new thread, you must specify a few pieces of information, including the function at which the thread should begin running—the thread start routine—and the Windows kernel takes care of everything thereafter. When the creation request returns successfully, the new thread will have been initialized, and, so long as it wasn't created as suspended (specified by an optional flag), registered into a queue of threads to be run and later scheduled onto a processor. When the thread actually gets to run on a processor is subject to the thread scheduler and, therefore, system load and available resources. In fact, the new thread may have already begun (or finished) running by the time the request for creation returns.

Once the new thread runs, its thread start routine can call any other code in the process, and so forth, accessing any shared memory in the process's address space, using other process-wide resources, and perhaps even creating additional threads of its own. The thread start routine can return normally or throw an unhandled exception, both of which terminate the thread, or alternatively the thread can be terminated via some

other more explicit mechanism. We'll take a look at each of these termination mechanisms momentarily. But first, let's see the APIs used to create threads.

Win32 and the .NET Framework offer different but very similar ways to create a new thread. If you're writing native C programs, there is also a separate set of C APIs you must use to ensure the C Runtime Library (CRT) is initialized properly. We'll start by looking at Win32. Both the .NET Framework and CRT thread creation routines effectively build directly on top of Win32.

In Win32. Kernel32 offers the `CreateThread` API to create a new thread.

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

`CreateThread` returns a `HANDLE` to the new thread kernel object, which can be passed to various other interesting Win32 APIs to later retrieve information about, interact with, or manipulate the newly created thread. (A `HANDLE`, by the way, is just an opaque pointer-sized value that indexes into a process-wide handle table. It's commonly used to refer to kernel objects. Managed code uses `IntPtr`s and `SafeHandles` to represent `HANDLE`s.) It must be closed when the creating thread no longer must interact with the new thread to avoid keeping the thread object's state alive indefinitely. The parameters to `CreateThread` are numerous:

- `LPSECURITY_ATTRIBUTES lpThreadAttributes`: a pointer to a `SECURITY_ATTRIBUTES` data structure. If `NULL`, the security attributes are inherited by the calling thread (which, if a thread along the way didn't specify overrides, in turn inherits them from the process). We will not discuss Windows object security in detail in this book; please refer to MSDN documentation and/or a book on Windows security for more details (see Further Reading, Brown).

- **SIZE_T dwStackSize:** the amount of user-mode stack, in bytes, to commit, in the virtual memory sense. If the **STACK_SIZE_PARAM_IS_A_RESERVATION** flag is present in the **dwCreationFlags** parameter, then this size represents the number of reserved bytes instead of committed bytes. 0 can be passed for **dwStackSize** to request that Windows use the process-wide default stack size. We discuss stack reservation, commit, and where this default comes from in the next chapter.
- **LPTHREAD_START_ROUTINE lpStartAddress:** a function pointer to the thread start routine. When Windows runs your thread, this is where it will begin execution. The type of function has the following signature:

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

The return value is captured and stored as the thread's exit code, which is then retrievable programmatically.

- **LPVOID lpParameter:** a pointer to memory you'd like to make accessible to the thread once it begins execution. This is opaque to Windows and is merely passed through as the value of your thread start routine's **lpParameter** argument. It's "opaque" because Windows will not attempt to dereference, validate it, or otherwise use it in any way. **NULL** is a valid argument value; without passing a pointer to some program data, the only valid way the thread will be able to find program data will be through accessing static or global variables.
- **DWORD dwCreationFlags:** a bit-flags value that enables you to indicate optional flags: that the stack size is for reservation rather than commit purposes (**STACK_SIZE_PARAM_IS_A_RESERVATION**), and/or that the thread should be left in a suspended state after **CreateThread** returns (**CREATE_SUSPENDED**). A thread that remains suspended must be resumed with a call to the Kernel32 **ResumeThread** API before it will be registered with the runnable thread queue and begin running. This can be useful if extra state must be prepared before the thread is able to begin executing. We look at thread suspension (**SuspendThread**) and resumption later.

- `LPDWORD lpThreadId`: An output pointer into which the `CreateThread` routine will store the newly created thread's process-wide unique identifier. As with the `HANDLE` returned, this can sometimes be used to subsequently interact with the thread. More often than not, it's just useful for diagnostics purposes. If you don't care about the thread's ID, as is fairly common, you can simply pass `NULL` (though on Windows 9X a valid non-`NULL` pointer must be supplied, otherwise `CreateThread` will attempt to dereference it and fail).

`CreateThread` can fail for a number of reasons, in which case the return value will be `NULL` and `GetLastError` may be used to retrieve details about the failure. Remember, each thread consumes a notable amount of system resources, including some amount of nonpageable memory, so if system resources are low, thread creation is very likely to fail: your code must be written to handle such cases gracefully, which may mean anything from choosing an alternative code-path or even terminating the program cleanly.

As a simple example of using `CreateThread`, consider Listing 3.1. In this code, the `main` routine is automatically called from the process's primary thread, which then invokes `CreateThread` to create a second program thread, supplying a function pointer to `MyThreadMain` as `lpStartAddress` and a pointer to the "Hello, World" string as `lpParameter`. Windows creates and enters the new thread into the scheduler's queue, at which point `CreateThread` returns and we make a call to the Win32 `WaitForSingleObject` API, passing the newly created thread's `HANDLE` as the argument. Though we don't look at the various Win32 wait functions Chapter 5, Windows Kernel Synchronization, this API call just causes the primary thread wait for the second thread to exit, allowing us to access and print the thread's exit code before exiting the program.

LISTING 3.1: Creating a new OS thread with Win32's CreateThread function

```
WIN32 - C++ CREATETHREAD.CPP
#include <stdio.h>
#include <windows.h>

DWORD WINAPI MyThreadStart(LPVOID);
```

```
int main(int argc, wchar_t * argv[])
{
    HANDLE hThread;
    DWORD dwThreadId;

    // Create the new thread.
    hThread = CreateThread(NULL,                  // lpThreadAttributes
                          0,                      // dwStackSize
                          &MyThreadStart, // lpStartAddress
                          "Hello, World", // lpParameter
                          0,                      // dwCreationFlags
                          &dwThreadId);   // lpThreadId

    if (!hThread)
    {
        fprintf(stderr, "Thread creation failed: %d\r\n",
                GetLastError());
        return -1;
    }

    printf("%d: Created thread %x (ID %d)\r\n",
           GetCurrentThreadId(), hThread, dwThreadId);

    // Wait for it to exit and then print the exit code.
    WaitForSingleObject(hThread, INFINITE);

    DWORD dwExitCode;
    GetExitCodeThread(hThread, &dwExitCode);
    printf("%d: Thread exited: %d\r\n",
           GetCurrentThreadId(), dwExitCode);
    CloseHandle(hThread);

    return 0;
}

DWORD WINAPI MyThreadStart(LPVOID lpParameter)
{
    printf("%d: Running: %s\r\n",
           GetCurrentThreadId(), reinterpret_cast<char *>(lpParameter));
    return 0;
}
```

Notice that we use a few other APIs that haven't been described yet. First, `GetCurrentThreadId` retrieves the ID of the currently executing thread. This is the same ID that was returned from `CreateThread`'s `lpThreadId` output parameter:

```
DWORD WINAPI GetCurrentThreadId();
```

And `GetExitCodeThread` retrieves the specified thread's exit code. We'll describe how exit codes are set when we discuss thread termination, but if you run this example, you'll see that when the thread terminates by its thread routine returning, the return value from the thread start is used as the exit code (which in this case means the value 0):

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode);
```

`GetExitCodeThread` sets the memory location behind the `lpExitCode` output pointer to contain the thread's exit code. Both the `ExitThread` and `TerminateThread` APIs, used to explicitly terminate threads, allow a return code to be specified at the time of termination. It is generally accepted practice to use non-0 return values to indicate that a thread exit was caused due to an abnormal or unexpected condition, while 0 is usually used to indicate that termination was caused by ordinary business. If you try to access a thread's exit code before it has finished executing, a value of `STILL_ACTIVE` (0x103) is returned: clearly you should avoid using this error code for meaningful values because it could be interpreted wrongly.

This example isn't very interesting, but it shows some simple coordination between threads. There is little concurrency here, as our primary thread just waits while the new thread runs. We'll see more interesting uses as we progress through the book.

Another API is worth mentioning now. As we've seen, `CreateThread` returns a `HANDLE` to the newly created thread. In some cases you'll want to retrieve the current thread's `HANDLE` instead. To do that, you can use the `GetCurrentThread` function.

```
HANDLE WINAPI GetCurrentThread();
```

The returned value can be passed to any `HANDLE` based functions. But note that the value returned is actually special—something called a **pseudo-handle**—which is just a constant value (-2) that no real `HANDLE` would ever contain. `GetCurrentProcess` works similarly (returns -1 instead). Not having to manufacture a real handle is more efficient, but more importantly, pseudo-handles do not need to be closed. That means you needn't call `CloseHandle` on the returned value. But because the pseudo-handle is always interpreted as “the current thread” by Windows,

you can't just share the pseudo-handle value with other threads (it would be subsequently interpreted by that thread as referring to itself). To convert it into a real handle that is shareable, you can call `DuplicateHandle`, which returns a new shareable `HANDLE` that must be closed when you are through with it. Here is a sample snippet of code that converts a pseudo-handle into a real handle, printing out the two values.

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

int main(int argc, wchar_t * argv[])
{
    HANDLE h1 = GetCurrentThread();
    printf("pseudo:\t%#x\r\n", h1);

    HANDLE h2;
    DuplicateHandle(
        GetCurrentProcess(), h1, GetCurrentProcess(), &h2,
        0, FALSE, DUPLICATE_SAME_ACCESS);

    printf("real:\t%#x\r\n", h2);

    CloseHandle(h2);
}
```

If all you've got is a thread's ID and you need to retrieve its `HANDLE`, you can use the `OpenThread` function. This also can be used if you need to provide a `HANDLE` that has been opened with only very specific access rights, that is, because you need to share it with another component.

```
HANDLE WINAPI OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadId
);
```

The `bInheritHandle` parameter specifies whether a `HANDLE` can be used by child processes (i.e., processes created by the one issuing the `OpenThread` call), and `dwThreadId` specifies the ID of the thread to which the `HANDLE` is to refer.

Finally, there is also a `CreateRemoteThread` function with nearly the same signature as `CreateThread`, with the difference that it accepts a process `HANDLE` as the first argument. As its name implies, this function

creates a new thread inside a process other than the caller’s. This is a rather obscure capability, but can come in useful for tools like debuggers.

In C Programs. When you’re programming with the C Runtime Library (CRT), you should use the `_beginthread` or `_beginthreadex` functions for thread creation in your C programs. These are defined in the header file `process.h`. These functions internally call `CreateThread`, but also perform some additional CRT initialization steps. If these steps are skipped, various CRT functions will begin failing in strange and unpredictable ways.

For example, the `strtok` function tokenizes a string. If you pass `NULL` as the string argument, it means “continue retrieving tokens from the previously tokenized string.” In the original CRT—which was written long before multithreading was commonplace on Windows—the ability to remember “the previous string” was implemented by storing the tokens in global variables. This was fine with single-threaded programs, but clearly isn’t for ones with multiple threads: imagine thread `t1` tokenizes a string, then another thread `t2` runs and tokenizes a separate string; when `t1` resumes and tries to obtain additional tokens, it will be inadvertently sharing the token information from `t2`. Just about anything can happen, such as global state corruption, which can cause crashes or worse. Other functions do similar things: for example, `errno` stores and retrieves the previous error (similar to Win32’s `GetLastError`) as global state.

With the introduction of the multithreaded CRT, `LIBCMT.LIB` (versus `LIBC.LIB`, usually accessed via the Visual C++ compiler switch `/MT`), all such functions now use thread local storage (TLS), which is just a collection of memory locations specific to each thread in the process. We’ll review TLS in more detail later. To ensure the TLS state that these routines rely on has been initialized properly, the thread calling `strtok` or any of the other TLS based functions must have been created with either `_beginthread` or `_beginthreadex`. If the thread wasn’t created in this way, these functions will try to access TLS slots that haven’t been properly initialized and will behave unpredictably.

The `_beginthread` and `_beginthreadex` functions are quite similar in form to the `CreateThread` function reviewed earlier. Because of the similarities, we’ll review them quickly.

```
uintptr_t _beginthread(
    void (*__cdecl * start_address)(void *),
    unsigned stack_size,
    void * arglist
);
uintptr_t _beginthreadex(
    void * security,
    unsigned stack_size,
    unsigned (__stdcall * start_address)(void *),
    void * arglist,
    unsigned initflag,
    unsigned * thrdaddr
);
```

Each takes a function pointer, `start_address`, to the routine at which to begin execution. The `_beginthread` function differs from `_beginthreadex` and `CreateThread` in that the function's calling convention must be `__cdecl` instead of `__stdcall`, as you would expect for a C based program versus a Win32 based one, and the return type is `void` instead of a `DWORD` (i.e., it doesn't return a thread exit code). Each takes a `stack_size` argument whose value is used the same as in `CreateThread` (`0` means the process-wide default) and an `arglist` pointer that is subsequently accessible via the thread start's first and only argument.

The `_beginthreadex` function takes two additional arguments. The value `CREATE_SUSPENDED` can be passed for the `initflag` parameter, which, just as with the `CreateThread` API, ensures that the thread is created in a suspended state and must be manually resumed with `ResumeThread` before it runs. There are no special CRT functions for thread suspend and resume. The `thrdaddr` argument, if non-NULL, receives the resulting thread identifier as an output argument.

In both cases, the function returns a handle to the thread (of type `uintptr_t`, which can safely be cast to `HANDLE`) or `0` if there was an error during creation. Be extremely careful when using `_beginthread`, as the thread's handle is automatically closed when the thread start routine exits. If the thread runs quickly, the `uintptr_t` returned could represent an invalid handle by the time `_beginthread` even returns. This is in contrast to `_beginthreadex` and `CreateThread`, which require that the code creating the thread closes the returned handle if it's not needed and makes `_beginthread` nearly useless unless the creating thread has no need to subsequently interact with the newly created thread.

We will discuss more about exiting threads in a CRT safe way later, when we talk about thread termination and the `_endthread` and `_endthreadex` functions.

In the .NET Framework. In managed code you can use the `System.Threading.Thread` class's constructors and `Start` methods to create a new managed thread. The primary difference between this mechanism and Win32's `CreateThread` is just that the CLR has a chance to set up various bookkeeping data structures, as described previously, and, of course, the use of a CLR object to represent the thread in your programs instead of an opaque `HANDLE`.

(There also is a corresponding class `System.Diagnostics.ProcessThread`, which also offers access to various thread information and attributes in managed code. This type exposes additional capabilities that the managed `Thread` object doesn't. However, you cannot retrieve an instance of `ProcessThread` from a `Thread` instance, and vice versa, so, as its name implies, this is much more useful as a diagnostics tool rather than something you will use in production code. Hence, most of this chapter ignores `ProcessThread` and instead focuses on the actual `Thread` class itself.)

First the thread object must be constructed using one of `Thread`'s various constructors.

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);

public class Thread
{
    public Thread(ThreadStart start);
    public Thread(ThreadStart start, int maxStackSize);
    public Thread(ParameterizedThreadStart start);
    public Thread(ParameterizedThreadStart start, int maxStackSize);
    ...
}
```

Assuming an unhosted CLR, each `Thread` object is just a thin object oriented veneer over an OS thread kernel object. Note that when you instantiate a new `Thread` object, the CLR hasn't actually created the underlying OS thread kernel object, user- or kernel-mode stack, and so on, just yet. This constructor just allocates some tiny internal data structures necessary to

store your constructor arguments so that they can be used should you decide to start the thread later. If you never get around to starting the thread, there will never be any OS resources backing it.

After creating the object, you must call the `Start` method on it to actually create the OS thread object and schedule it for execution. As you might imagine, the unhosted CLR uses the `CreateThread` API internally to do that.

```
public class Thread
{
    public void Start();
    public void Start(object parameter);
    ...
}
```

A thread created with the `ParameterizedThreadStart` based constructor allows a caller to pass an object reference argument to the `Start` method (as `parameter`), which is then accessible from the new thread's start routine as `obj`. This is similar to the `CreateThread` API, seen above, and provides a simple way of communicating state between the creator and createe. A similar effect can be achieved by passing a thread start delegate that refers to an instance method on some object, in which case that object's instance state will be accessible from the thread start via `this`. If a thread created with a `ParameterizedThreadStart` delegate is subsequently started with the parameterless `Start` overload, the value of the thread start's `obj` argument will be `null`.

There are a couple of constructor overloads that accept a `maxStackSize` parameter. This specifies the size of the thread's reserved and committed stack size (because in managed code both are the same). We return to more details about stacks in the next chapter, including why you might want to change the default.

It's also worth pointing out that many of `Thread`'s methods (in addition to most synchronization related methods), including `Start`, are protected by a Code Access Security `HostProtection` link demand for `Synchronization` and `ExternalThreading` permissions. This ensures that, while untrusted code can create a new CLR thread object (because its constructors are not protected), most code hosted inside a program like SQL Server cannot start or control a thread's execution. Deep examinations of security and hosting are both outside of the scope of this book. Please refer to Further Reading, Brown and Pratschner, for excellent books on the topics.

Listing 3.2 illustrates an example comparable to the Win32 code in Listing 3.1 earlier. Just as we had used the `WaitForSingleObject` Win32 API to wait for the thread to exit, we use Thread's `Join` method. We'll review `Join` in more detail later, though it doesn't get much more complicated than what is shown here. You'll notice that the CLR doesn't expose any sort of thread exit code capability.

LISTING 3.2: Creating a new OS thread with the .NET Framework's Thread class

```
using System;
using System.Threading;

class Program
{
    public static void Main()
    {
        Thread newThread = new Thread(
            new ParameterizedThreadStart(MyThreadStart));

        Console.WriteLine("{0}: Created thread (ID {1})",
            Thread.CurrentThread.ManagedThreadId,
            newThread.ManagedThreadId);

        newThread.Start("Hello world"); // Begin execution.

        newThread.Join(); // Wait for the thread to finish.

        Console.WriteLine("{0}: Thread exited",
            Thread.CurrentThread.ManagedThreadId);
    }

    private static void MyThreadStart(object obj)
    {
        Console.WriteLine("{0}: Running: {1}",
            Thread.CurrentThread.ManagedThreadId, obj);
    }
}
```

You can write this code more succinctly using C# 2.0's anonymous delegate syntax.

```
Thread newThread = new Thread(delegate(object obj)
{
    Console.WriteLine("{0}: Running {1}",
        Thread.CurrentThread.ManagedThreadId, obj);
});
newThread.Start("Hello world (with anon delegates)");
newThread.Join();
```

Using lambda syntax in C# 3.0 makes writing similar code even slightly more compact.

```
Thread newThread = new Thread(obj =>
    Console.WriteLine("{0}: Running {1}",
        Thread.CurrentThread.ManagedThreadId, obj)
);
newThread.Start("Hello, world (with lambdas)");
newThread.Join();
```

We make use of the `CurrentThread` static property on the `Thread` class, which retrieves a reference to the currently executing thread, much like `GetCurrentThread` in Win32. We then use the instance property `ManagedThreadId` to retrieve the unique identifier assigned by the CLR to this thread. This identifier is completely different than the one assigned by the OS. If you were to P/Invoke to `GetCurrentThreadId`, you'll likely see a different value.

```
public class Thread
{
    public static Thread CurrentThread { get; }
    public int ManagedThreadId { get; }
    ...
}
```

Again, this code snippet isn't very illuminating. We'll see more complex examples. But as you can see, the idea of a thread as seen by Win32 and managed code programmers is basically the same. That's good as it means most of what we've discussed and are about to discuss pertains to native and managed code alike.

Thread Termination

A thread goes through a complex lifetime, from runnable to running to possibly waiting, possibly being suspended, and so forth, but it will eventually terminate. Termination might occur as a result of any one of a number of particular events.

1. The thread start routine can return normally.
2. An unhandled exception can escape the thread start routine, "crashing" that thread.

3. A call can be made to one of the Win32 functions `ExitThread` or `TerminateThread`, either by the thread itself (synchronous) or by another thread (asynchronous). There is no direct equivalent to these functions in the .NET Framework, and P/Invoking to them will lead to much trouble.
4. A managed thread abort can be triggered by a call to the .NET Framework method `Thread.Abort`, either by the thread itself (synchronous) or by another thread (asynchronous). There is no equivalent in Win32. This approach in fact looks a lot like `ExitThread`, though you can argue that it is a “cleaner” way to shut down threads. We’ll see why shortly. That said, aborting threads is still (usually) a bad practice.

A managed thread may also be subject to a thread abort induced by the CLR infrastructure or a CLR host. Aborts also occur on all threads running code in an AppDomain when it is being unloaded. This is different from the previous item because it’s initiated by the infrastructure, which knows how to do this safely.

5. The process may exit.

Of course, the machine could get unplugged, in which case threads terminate, but since there’s not much our software can do in response to such an event, we’ll set this aside.

After a thread terminates, assuming the process remains alive, its data structures continue to live on until all of the `HANDLEs` referring to the thread object have been closed. The CLR thread object, for example, uses a finalizer to close this handle, which means that the OS data structures will continue to live until the GC collects the `Thread` object and then runs its finalizer, even though the thread is no longer actively running any code.

Several of the techniques mentioned are brute force methods for thread termination and can cause trouble (namely 3 and 4). Higher-level coordination must be used to cooperatively shut down threads or else program and user data can become corrupt.

Note that the termination of a thread may cause termination of its owning process. In native code, the process will exit automatically when the last thread in a process exits. In managed code, threads can be marked as a

background thread (with the `IsBackground` property), which ensures that a particular thread won't keep the process alive. A managed process will automatically exit once its last nonbackground thread exits. As with thread termination, there are other brute force (and problematic) ways to shut down a process, such as with a call to `TerminateProcess`.

Method 1: Returning from the Thread Start Routine

Any thread start routine that returns will cause the thread to exit. This is by far the cleanest way to trigger thread exit. The top of each thread's callstack is actually a Windows internal function that calls the thread start routine and, once it returns, calls the `ExitThread` API. This is true for both native and managed threads and is imposed by Windows. This is the cleanest shutdown method because the thread start routine is able to run to completion without being interrupted part way through some application specific code.

While not exposed through the managed thread object, each OS thread remembers an exit code, much like a process does. The `CreateThread` start routine function pointer type returns a `DWORD` value and the callback for `_beginthreadex` returns an `unsigned` value. Managed threading doesn't support exit codes and is evidenced by the fact that `ThreadStart` and `ParameterizedThreadStart` are typed as returning `void`. Programs can use exit codes to communicate the reason for thread termination. Windows stores the return value as part of the thread object so that it can be later retrieved with `GetExitCodeThread`, as we saw just a bit earlier. Most alternative forms of thread termination also supply a way to set this code.

Method 2: Unhandled Exceptions

If an exception reaches the top of a thread's stack without having been caught, the thread will be terminated. The default Windows and CLR behavior is to terminate the process when such an unhandled exception occurs (for most cases), though a custom exception filter can be installed to change this behavior. Of course, many exceptions are handled before getting this far, in which case there is no impact on the life of the thread. Additionally, some programs install custom top-level handlers that catch all exceptions, perform error logging, and attempt some level of data recovery before letting the process crash.

Process termination works by installing at the base of every Windows thread's stack an SEH exception filter. This filter decides what to do with unhandled exceptions. The details here differ slightly between native and managed code, because managed code wraps everything in its own exception filter and handler too.

The default filter in native code will display a dialog when the exception has been deemed to go unhandled during the first pass. It asks the user to choose whether to debug or terminate the process (the latter of which just calls `ExitProcess`). All of this occurs in the first pass of exception handling, so by default, no stacks have been unwound at this point. Anybody who has written code on Windows knows what this dialog looks like. Though it tends to change from release to release, it offers the same basic functionality: debug or terminate the process and, now in Windows Vista, check for solutions online.

The CLR installs its own top-level unhandled exception filter, which performs debugger notification, integrates with Dr. Watson to generate proper crash dumps, raises an event in the AppDomain so that custom managed code can execute shutdown logic, prints out more friendly failure information (including a stack trace) to the console, and unwinds the crashing thread's stack, letting managed finally blocks run. One interesting difference is that finally blocks are run when a managed thread crashes, while in native they are not (by default). This custom exception logic is run regardless of whether it was a managed or native thread in the process that caused the unhandled exception because the CLR overrides the process-wide unhandled exception behavior.

There are two special exceptions to the rule that any unhandled exception causes the process to exit: an unhandled `ThreadAbortException` or `AppDomainUnloadedException` will cause the thread on which it was thrown to exit, but will not actually trigger a process exit (unless it's the last nonbackground thread in the process). Instead, the exception will be swallowed and the process will continue to execute as normal. This is done because these exceptions are regularly used by the runtime and CLR hosts to carefully unload an AppDomain while still keeping the rest of the process alive.

Overriding the Default Unhandled Exception Behavior. There are a few ways in which you may override the default unhandled exception behavior. Doing so is seldom necessary. The first way allows you to turn off the default dialog in Win32 programs by passing the SEM_NOGPFAULTERRORBOX flag to the SetErrorMode function. This is usually a bad idea if you want to be able to debug your programs, but it can be useful for noninteractive programs:

```
UINT SetErrorMode(UINT uMode);
```

A change was made in the CLR 2.0 to make unhandled exceptions on the finalizer thread, thread pool threads, and user created threads exit the process. In the CLR 1.X, such exceptions were silently swallowed by the runtime. An unhandled exception is more often than not an indication that something wrong has happened and, therefore, the old policy tended to lead to many subtle and hard to diagnose errors. Swallowing the exception merely masked a problem that was sure to crop up later in the program's execution. At the same time, this change in policy can cause compatibility problems for those migrating from 1.X to 2.0 and above. A configuration setting enables you to recover the 1.X behavior.

```
<system>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</system>
```

Using this configuration setting is highly discouraged for anything other than as an (one hopes temporary) application compatibility crutch. It can create debugging nightmares. CLR hosts can also override (some of) this unhandled exception behavior, so what has been described in this section strictly applies only to unhosted managed programs. Please refer to Pratschner (see Further Reading) for details on how this is done.

Some of you might be wondering how the CLR is able to hook itself into the whole Windows unhandled exception process so easily. Any user-mode code can install a custom top-level SEH exception filter that will be called instead of the default OS filter when an unhandled exception occurs. SetUnhandledExceptionFilter installs such a filter.

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

`LPTOP_LEVEL_EXCEPTION_FILTER` is just a function pointer to an ordinary SEH exception filter.

```
LONG WINAPI UnhandledExceptionFilter(
    struct _EXCEPTION_POINTERS * ExceptionInfo
);
```

The `_EXCEPTION_POINTERS` data structure is passed by the OS—and is the same value you'd see if you were to call `GetExceptionInformation` by hand during exception handling—which provides you with an `EXCEPTION_RECORD` and `CONTEXT`. The record provides exception details and the `CONTEXT` is a collection of the processor's volatile state (i.e., registers) at the time the exception occurred. We review contexts later in this chapter. As with any filter, this routine can inspect the exception information and decide what to do. At the end, it returns `EXCEPTION_CONTINUE_SEARCH` or `EXCEPTION_EXECUTE_HANDLER` to instruct SEH whether to execute a handler or not.

(The details of the CLR and Windows SEH exception systems are fascinating, but are fairly orthogonal to the topic of concurrency. Therefore we won't review them here, and instead readers are encouraged to read Pietrek (see Further Reading) for a great overview.)

If you return `EXCEPTION_CONTINUE_SEARCH` from this top-level filter, the exception goes completely unhandled and the OS will perform the default unhandled exception behavior. That entails showing the dialog (assuming it has not been disabled via `SetErrorMode`) and calling `ExitProcess` without unwinding the crashing thread's stack. All of this happens during the first pass. If you return `EXCEPTION_EXECUTE_HANDLER`, however, a special OS-controlled handler is run. This SEH handler sits at the base of all threads and will call `ExitProcess` without displaying the standard error dialog. And because we have told SEH to execute a handler, the thread's stack is unwound normally, and, hence, the call to `ExitProcess` occurs during the second pass after finallys blocks have been run.

Method 3: ExitThread and TerminateThread (Native Code Only)

If you're writing native code, you can explicitly terminate a thread (although it is generally very dangerous to do so and should be done only after this is understood). This can be done for the current thread (synchronous) or another thread running in the system (asynchronous). There are two Win32 APIs to initiate explicit thread termination

```
VOID WINAPI ExitThread(DWORD dwExitCode);  
BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);
```

Calling `ExitThread` will immediately cause the thread to exit, without unwinding its stack, meaning that finally blocks and destructors will not execute. It changes the thread's exit code from `STILL_ACTIVE` to the value supplied as the `dwExitCode` argument. The thread's user- and kernel-mode stack memory is de-allocated, pending asynchronous I/O is canceled (see Chapter 15, Input and Output), thread detach notifications are delivered to all DLLs in the process that have defined a `DllMain` entry point, and the kernel thread object becomes signaled (see Chapter 5, Windows Kernel Synchronization). The thread may continue to use resources because the kernel object and its associated memory remains allocated until all outstanding `HANDLE`s to it have been closed.

If you created threads with the CRT's `_beginthread` or `_beginthreadex` function, then you must use the `_endthread` or `_endthreadex` function instead of `ExitThread`.

```
void _endthread();  
void _endthreadex(unsigned retval);
```

Internally, these both call `ExitThread`, but they additionally provide a chance for the CRT to de-allocate any per-thread resources that were allocated at runtime. Terminating threads created with the `_beginthread` routines using `ExitThread` or `TerminateThread` will cause these resources to be leaked. The leaks are so small that they could go unnoticed for some time, but will certainly cause progressively severe problems for long running programs. The only difference between `_endthread` and `_endthreadex` is that `_endthreadex` accepts a thread exit code as the `retval` argument, while `_endthread` simply uses `0` as the exit code.

The first method of terminating a thread described earlier—returning from the thread start routine—internally calls `ExitThread` (via `_endthreadex`) at the base of the stack, passing the routine’s return value as the `dwExitCode` argument. Exiting a thread can only occur synchronously on a thread; in other words, some other thread can’t exit a separate thread “from the outside.” This means that `ExitThread` is safer, though it can lead to issues like lock orphaning and memory leaks because the thread’s stack is not freed before exiting.

The `TerminateThread` function, on the other hand, is extremely dangerous and should almost never be used. The only possible situations in which you should consider using it are those where you are entirely in control of what code the target thread is executing. Terminating a thread this way does not free the user-mode stack and does not deliver `DllMain` notifications. Calling it synchronously on a thread is very similar to `ExitThread`, with these two differences aside. But calling it asynchronously can cause problems. The target thread could be holding on to locks that, after termination, will remain in the acquired state. For example, the thread might be in the process of allocating memory, which often requires a lock. Once terminated, no other thread would be able to subsequently allocate memory, leading to deadlocks. Similarly, the target could be modifying critical system state that could become corrupt when interrupted part way through. If you are considering using `TerminateThread`, you should follow it soon with a call to terminate the process as well.

In all cases, using higher-level synchronization mechanisms to shut down threads is always preferred. This typically requires some combination of state and cooperation among threads to periodically check for shutdown requests and voluntarily return back to the thread start routine when a request has been made. `ExitThread` and `TerminateThread` often seem like “short-cuts” to achieve this, while avoiding the need to perform this kind of higher-level orchestration; there’s certainly less tricky cooperation code to write because many important issues are hidden. Generally speaking, this should be considered a sloppy coding practice, viewed with great suspicion, and regarded as likely to lead to many bugs.

Managed code should never explicitly terminate managed threads using these mechanisms. Instead, synchronization should be used to orchestrate

exit or, in some specific scenarios, thread aborts can be used instead (see below). P/Invoking to `ExitThread` or `TerminateThread` will lead to unpredictable and unwanted behavior for much the same reason that calling `ExitThread` instead of `_endthreadex` can cause problems: that is, the CLR has state to clean up and bookkeeping to perform whenever a thread terminates.

Method 4: Thread Aborts (Managed Code Only)

Managed threads can be aborted. When a thread is aborted, the runtime tears it down by introducing an exception at the thread's current instruction pointer, versus stopping the thread in its tracks a la the Win32 `ExitThread` function. Using an exception such as this allows finally blocks to execute as the thread unwinds, ensuring that important resources are cleaned up appropriately. Moreover, the runtime is aware of certain regions of code that are performing uninterruptible operations, such as manipulating important system-wide state, and will delay introducing the aborting exception until a safe point has been reached.

Thread aborts can be introduced synchronously and asynchronously, just like `TerminateThread`. When an asynchronous abort is triggered, an instance of `System.Threading.ThreadAbortException` is constructed and thrown in the aborted thread, just as if the thread itself threw the exception. Synchronous aborts, on the other hand, are fairly straightforward: the thread itself just throws the exception. As described earlier, unhandled thread abort exceptions only terminate the thread on which the exception was raised, and do not cause the process to exit (unless that was the last nonbackground thread).

To initiate a thread abort, the `Thread` class offers an explicit `Abort` API.

```
public void Abort();
public void Abort(object stateInfo);
```

When aborting another thread asynchronously, the call to `Abort` blocks until the thread abort has been processed. Note that when the call unblocks, it does not mean that the thread has been aborted yet. In fact, the thread may suppress the abort, so there is no guarantee that the thread will exit. You should use other synchronization techniques (such as the `Join` API) if you must wait for the thread to complete. If the overload, which accepts the

`stateInfo` parameter, is used, the object is accessible via the `ThreadAbortException`'s `ExceptionState` property, allowing one to communicate the reason for the thread abort.

`ThreadAbortExceptions` thrown during a thread abort are special. They cannot be swallowed by catch blocks on the thread's callstack. The stack will be unwound as usual, but if a catch block tries to swallow the exception, the CLR reraises it once the catch block has finished running. An abort can be reset mid-flight with the `Thread.ResetAbort` API, which will allow exceptions to be caught and the thread to remain alive.

```
public static void ResetAbort();
```

The following code snippet illustrates this behavior.

```
try
{
    try
    {
        Thread.CurrentThread.Abort();
    }
    catch (ThreadAbortException)
    {
        // Try to swallow it.
    } // CLR automatically reraises the exception here.
}
catch (ThreadAbortException)
{
    Thread.ResetAbort();
    // Try to swallow it again.
} // The in-flight abort was reset, so it is not reraised again.
```

A single callstack may be executing code in multiple AppDomains at once. Should a `ThreadAbortException` cross an AppDomain boundary on a callstack, say from AppDomain B to A, it will be morphed into an `AppDomainUnloadedException`. Unlike thread abort exceptions, this exception type can be caught and swallowed by code running in A.

Delay-Abort Regions. As mentioned earlier, the runtime only initiates an asynchronous thread abort when the target thread is not actively running critical code: these are called **delay-abort regions**. Each of the following is considered to be a delay-abort region by the CLR: invocation of a catch or

finally block, code within a constrained execution region (CER), running native code on a managed thread, or invocation of a class or module constructor. When a thread is in such a region and is asynchronously aborted, the thread is simply marked with a flag (reflected in its state bitmask by `ThreadState.AbortRequested`), and the thread subsequently initiates the abort as soon as it exits the region, that is, when it reaches a safe point (taking into consideration that such regions may be nested). The determination of whether a thread is in a delay-abort region is made by the CLR suspending the target thread, inspecting its current instruction pointer, and so on.

Thread Abort Dangers. There are two situations in which thread aborts are always safe.

- The main purpose of thread aborts is to tear down threads during CLR AppDomain unloads. When an unload occurs—either because a host has initiated one or because the program has called the `AppDomain.Unload` function—any thread that has a callstack in an AppDomain is asynchronously aborted. As the abort exceptions reach the boundary of the AppDomain, the thread abort is reset and the exception turns into an `AppDomainUnloadedException`, which, as we've noted, can then be caught and handled. This is safe because nearly all .NET Framework code assumes that an asynchronous thread abort means the AppDomain is being unloaded and takes extra precautions to avoid leaking process-wide state.
- Synchronous thread aborts are safe, provided that callers expect an exception to be thrown from the method. Because the thread being aborted controls precisely when aborts happen, it's the responsibility of that code to ensure they happen when program state is consistent. A synchronous abort is effectively the same as throwing any kind of exception, with the notable difference that it cannot be caught and swallowed. It's possible that some code will check the type of the exception in-flight and avoid cleaning up state so that AppDomain unloads are not held up, but these cases should be rare.

All other uses of thread aborts are questionable at best. While a great deal of the .NET Framework goes to great lengths to ensure resources are not leaked and deadlocks do not occur (see Further Reading, Duffy, Atomicity and Asynchronous Exception Failures), the majority of the libraries are not written this way. Note that hosts can also initiate a so-called rude thread abort, which does not run finally blocks and will interrupt the execution of catch and finally clauses. This capability is used only by some hosts and not the unhosted CLR itself and, therefore, is inaccessible to managed code. A detailed discussion of this is outside the scope of this book.

While thread aborts are theoretically safer than other thread termination mechanisms, they can still occur at inopportune times, leading to instability and corruption if used without care. While the runtime knows about critical system state modifications, it knows nothing about application state and, therefore, aborts are not problem free. In fact, you should rarely (if ever) use one. But the runtime and its hosts are able to make use of them with great care, usually because possible state corruption can be contained appropriately.

As a simple illustration of what can go wrong when aborts occur at unexpected and inopportune places, let's look at an example that leads to a resource leak.

```
void UseSomeBigResource()
{
    IntPtr hBigResource = /* S0 */ Allocate();
    try
    {
        // Do something...
    }
    finally
    {
        Free(hBigResource);
    }
}
```

In this example, a thread abort could be triggered after the call to `Allocate` but before the assignment to the `hBigResource` local variable, at S0. An asynchronous thread abort here will lead to memory leakage (because the memory is not GC managed). Even if we were assigning the

result of `Allocate` to a member variable on a type that had a finalizer, to catch the case where the `try/finally` didn't execute the resource would leak because we never executed the assignment. If instead of allocating memory we were acquiring a mutually exclusive lock, for example, then an abort could lead to deadlock for threads that subsequently tried to acquire the orphaned lock. There are certainly ways to ensure reliable acquisition and release of resources (see Further Reading, Toub; Grunke Meyer), including using delay-abort regions with great care, but given that many of them are new to the CLR 2.0, most code that has been written remains vulnerable to such issues.

Method 5: Process Exit

The final method of terminating a thread is to exit the process without shutting down all of its threads. When it happens, it usually occurs in one of the following ways.

- Win32 offers `ExitProcess` and `TerminateProcess` APIs, which mirror the `ExitThread` and `TerminateThread` APIs reviewed earlier. When `ExitProcess` is called, `ExitThread` is called on all threads in the process, ensuring that DLL thread and process detach notifications are sent to DLLs loaded in the process. Threads are not unwound, so any destructors or finally blocks that are live on call stacks on these threads are not run. `TerminateProcess`, on the other hand, is effectively like calling `TerminateThread` on each thread and also skips the step of sending process detach notifications to loaded DLLs. Because these notifications are skipped, DLLs are not given a chance to free or restore machine-wide state.
- C programs can call either the `exit/_exit` or `abort` CRT library functions, which are similar to `ExitProcess` and `TerminateProcess`, respectively. Each contains additional logic, however. For example, `exit` invokes any routines registered with the CRT `atexit/_onexit` functions, and `abort` displays a dialog box indicating that the process has terminated abnormally.
- Managed code may call `Environment.Exit`, which triggers a clean shutdown of all threads in the process. The CLR will suspend all

threads, and then it will finalize any finalizable objects in the process. After this, it exits threads without running finally blocks. The CLR will actually create a so-called “shutdown watchdog thread” that monitors the shutdown process to ensure it doesn’t hang. As we’ll see in Chapter 6, Data and Control Synchronization, there are circumstances in which managed threads may hang during shutdown due to locks. If, after 2 seconds, the shutdown has not finished, the watchdog thread will take over and rudely shut down the process.

- Any managed code may also call `Environment.FailFast`. This is similar to calling `Exit`, except that it is meant for abnormal and unexpected situations where no managed code must run during the shutdown. This means that finalizers are not run, and `AppDomain` events are not called, and also an entry is made in the Windows Event Log to indicate failure.

The behavior explained above during shutdown in managed code always occurs. In fact, threads need to be terminated prematurely more frequently than you might think. That’s because a managed process exits when all nonbackground threads exit, and it is actually quite common to have many background threads (e.g., in the CLR’s thread pool).

Shutting down a process without cleanly exiting the application can lead to problems, particularly if you’re using `TerminateThread` or `FailFast`. These APIs are best used to respond to critical situations in which continuing execution poses more risk to the stability of the system and integrity of data than shutting down abruptly and possibly missing some important application-specific cleanup activities. For example, if a thread is in the middle of writing data to disk, it will be stopped midway, possibly corrupting data. Even if a thread has finished writing, data may not be flushed until a certain point in the future, and shutting down skips finally blocks, etc., which may result in buffers not being flushed. There are many things that can go wrong, and they depend on subtle timings and interactions, so a clean shutdown should always be preferred over all of the methods described in this section.

DllMain

We've referenced `DLL_THREAD_ATTACH` and `DLL_THREAD_DETACH` notifications at various points above. Now let's see how you register to receive such notifications. Each native DLL may specify a `DllMain` entry point function in which code to respond to various interesting process events may be placed. The signature of the `DllMain` function is:

```
BOOL WINAPI DllMain(
    HINSTANCE hInstDLL,
    DWORD fdwReason,
    LPVOID lpReserved
);
```

Defining a DLL entry point is optional. The OS will call the entry point for all DLLs that have defined entry points, as they are loaded into the process, when one of four events occurs. The event is indicated by the value of the `fdwReason` argument supplied by the OS:

- **DLL_PROCESS_ATTACH:** This is called when a DLL is first loaded into a process. For libraries statically linked into an EXE, this will occur at process load time, while for dynamically loaded DLLs, it will occur when `LoadLibrary` is invoked. This event may be used to perform initialization of data structures that the DLL will need during execution. If the `lpReserved` argument is `NULL`, it indicates the DLL has been loaded dynamically, while non-`NULL` indicates it has been loaded statically.
- **DLL_PROCESS_DETACH:** This is called when the DLL is unloaded from the process, either because the process is exiting or, for dynamically loaded libraries, when the `FreeLibrary` function has been called. The process detach notification handling code is ordinarily symmetric with respect to the process attach; in other words, it typically is meant to free any data structures or resources that were allocated during the initial DLL load. If `lpReserved` is `NULL`, it indicates the DLL is being dynamically unloaded with `FreeLibrary`, while non-`NULL` indicates the process is terminating.
- **DLL_THREAD_ATTACH:** Each time the process creates a new thread, this notification will be made. Any thread specific data structures may

then be allocated. Note that when the initial process attach notification is sent there is not an accompanying thread attach notification, neither will there be notifications for existing threads in the process when a DLL is dynamically loaded after threads were created.

- **DLL_THREAD_DETACH:** When a thread exits the system, the OS invokes the `DllMain` for all loaded DLLs and sends a detach notification from the thread that is exiting. This is the DLL's opportunity to free any data structures or resources allocated inside of the thread attach routine.

There is no equivalent to `DllMain` in managed code. Instead, there is an `AppDomain.ProcessExit` event that the CLR calls during process shutdown. If you are writing a C++/CLI assembly, or interoperating with an existing native DLL, however, you will be delivered `DllMain` notifications as normal.

The `DllMain` function is one of few places that program code is invoked while the OS holds the loader lock. The loader lock is a critical region used by the OS to protect access to loadtime state and automatically acquires it in several places: when a process is shutting down, when a DLL is being loaded, when a DLL is being unloaded, and inside various loader related APIs. It's a lock just like any other, and so it is subject to deadlock. This makes it particularly dangerous to write code in the `DllMain` routine. You must not trigger another DLL load or unload, and certainly should never synchronize with another thread that might hold a lock and then need to acquire the loader lock. It's easy to write deadlock prone code in your `DllMain` without even knowing it. Techniques like lock leveling (see Chapter 11, Concurrency Hazards, for details) can avoid deadlock, but generally speaking, it's better to avoid all synchronization in your `DllMain` altogether. See Further Reading, MSDN, Best Practices for Creating DLLs, for some additional best practices for DLL entry point code.

Prior to C++/CLI in Visual Studio 2005, it was impossible to create a C++ mixed mode native/managed DLL that contained a `DllMain` without it being deadlock prone. The reasons are numerous (see Further Reading, Brumme), but the basic problem is that it's impossible to run managed code without acquiring locks and possibly synchronizing with other threads (due to GC), which effectively guarantees that deadlocks are always

possible. If you're still writing code in 1.0 or 1.1, workarounds are possible (see Further Reading, Currie). As of Visual C++ 2005, however, managed code is not called automatically inside of `DllMain` and thus it's possible to write safe deadlock free entry points, provided you do not call into managed code explicitly. See Further Reading, MSDN, Visual C++: Initialization of Mixed Assemblies for details.

There is a hidden cost to defining `DllMain` routines. Every time a thread is created or destroyed, the OS must enumerate all loaded DLLs and invoke their `DllMain` functions with an attach or detach notification, respectively. Win32 offers an API to suppress notifications for a particular DLL, which can avoid this overhead when the calls are unnecessary.

```
BOOL WINAPI DisableThreadLibraryCalls(HMODULE hModule);
```

Using this API to suppress DLL notifications can provide sizeable performance improvements, particularly for programs that load many DLLs and/or create and destroy threads with regularity. But use it with caution. If a third party DLL has defined a `DllMain` function, it's probably for a reason; suppressing calls into it is apt to cause unpredictable behavior.

Thread Local Storage

Programs can store information inside thread local storage (TLS), which permits each thread to maintain some private data that isn't shared among other threads but that is globally accessible to any code running on that thread. This enables one part of the program to place data into a known location so another part can subsequently access and/or modify it. Static variables in C++ and C#, for example, refer to memory that is shared among all threads in the process. Accessing this shared state must be done with care, as we've established in previous chapters. It's often more attractive to isolate data so that synchronization isn't necessary or because the specific details of your problem allow or require information to be thread specific.

That's where TLS comes into the picture. With TLS, each thread in the system is allocated a separate region of memory to represent the same logical variable. Native and managed code both offer TLS support, with very similar programming interfaces, but the details of each are rather different. We'll review both, in that order.

Win32 TLS

There are two TLS modes for native code: dynamic and static. **Dynamic TLS** can be used in any situation, including static and dynamic link libraries, and executables. **Static TLS** is supported by the C++ compiler and may only be used for statically linked code but has the advantage of greater efficiency when accessing TLS information. Code can freely intermix the two in the same program and process without problems.

Dynamic TLS. In order to use native TLS to store and retrieve information, you must first allocate a TLS slot for each separate piece of data. Allocating a slot simply retrieves a new index and removes it from the list of available indices in the process. This slot index is a numeric **DWORD** value that is used to set or retrieve a **LPVOID** value stored in a per thread, per slot location managed by the OS. In fact, this value is just an index into an array of **LPVOID** entries that each thread has allocated at thread instantiation time.

Reserving a new index is done with the **TlsAlloc** API.

```
DWORD WINAPI TlsAlloc();
```

All TLS slots are 0 initialized when a thread is created, so all slots will initially contain the value **NULL**. The index itself should be treated as an opaque value, much like a **HANDLE**. Each thread in the process uses this same index value to access the same TLS slot, meaning that the value is typically shared in some static or global variable that all threads can access.

If **TlsAlloc** returns **TLS_OUT_OF_INDEXES**, the allocation of the TLS slot failed. The per thread array of TLS slots is limited in number (64 in Windows NT, 95; 80 in Windows 98; and 1,088 in Windows 2000 and beyond, according to MSDN and empirical results). If too many components in a process are fighting to create large numbers of slots, this error can result. In practice, this seldom arises, but the error condition needs to be handled.

Once a TLS slot has been allocated, the **TlsSetValue** and **TlsGetValue** functions can be used to set and retrieve data from the slots, respectively.

```
BOOL WINAPI TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);  
LPVOID WINAPI TlsGetValue(DWORD dwTlsIndex);
```

Note that the TLS slot **dwTlsIndex** isn't validated at all, other than ensuring it falls within the range of available slots mentioned above

(i.e., so that an out-of-bounds array access doesn't result). This means that, due to programming error, you can accidentally index into a garbage slot and the OS will permit you to do so, leading to unexpected results. In the case where you provide a dwTlsIndex value outside of the legal range (e.g., less than 0 or greater than 1,087 on Windows 2000), TlsSetValue returns FALSE and TlsGetValue returns NULL. GetLastError in both cases will return ERROR_INVALID_PARAMETER (87). Note that NULL is a legal value to store inside a slot, which can be easily confused with an error condition; TlsGetValue indicates the lack of error by setting the last error to ERROR_SUCCESS.

Last, you must free a TLS slot when it's no longer in use. If this step is forgotten, other components trying to allocate new slots will be unable to re-use the slot, which is effectively a resource leak and can result in an increase in `TLS_OUT_OF_INDEXES` errors. Freeing a slot is done with the `TlsFree` function.

```
BOOL WINAPI TlsFree(DWORD dwTlsIndex);
```

This function returns FALSE if the slot specified by dwTlsIndex is invalid, and TRUE otherwise. Note that freeing a TLS slot zeroes out the slot memory and simply makes the index available for subsequent calls to TlsAlloc. If the LPVOID value stored in the slot is a pointer to some block of memory, the memory must be explicitly freed before freeing the index. As soon as the TLS slot is free, the index is no longer safe to use—the slot can be handed out immediately to any other threads attempting to allocate slots concurrently, even before the call to TlsAlloc returns, in fact.

It's common to use `DllMain` to perform much of the aforementioned TLS management functions, at least when you're writing a DLL. For example, you can call `TlsAlloc` inside `DLL_PROCESS_ATTACH`, initialize the slot's contents for each thread inside `DLL_THREAD_ATTACH`, free the slot's contents during `DLL_THREAD_DETACH`, and call `TlsFree` inside of `DLL_PROCESS_DETACH`. For instance:

```

{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
            // Allocate a TLS slot.
            if ((g_dwMyTlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
            {
                ; // Handle the error ...
            }
            break;

        case DLL_PROCESS_DETACH:
            // Free the TLS slot.
            TlsFree(g_dwMyTlsIndex);
            break;

        case DLL_THREAD_ATTACH:
            // Allocate the thread-local data.
            TlsSetValue(g_dwMyTlsIndex, new int[1024]);
            break;

        case DLL_THREAD_DETACH:
            // Free the thread local data.
            int * data = reinterpret_cast<int *>(
                TlsGetValue(g_dwMyTlsIndex));
            delete [] data;
            break;
    }
}

```

Recall from earlier that there are some cases in which thread attach and detach notifications may be missed. If a DLL is loaded dynamically, for example, threads may exist prior to the load, in which case there will not be `DLL_THREAD_ATTACH` notifications for them. For that reason, you will usually need to write your code to check the TLS value to see if it has been initialized and, if not, do so lazily. And as noted earlier, sometimes `DLL_THREAD_DETACH` notifications will be skipped. There is little within reason you can do here, and so killing threads in a manner that skips detach notifications when TLS is involved often leads to leaks. This is yet another reason to avoid APIs like `TerminateThread`.

Static TLS. Instead of writing all of the boilerplate to `TlsAlloc`, `TlsFree`, and manage the per-thread data for each TLS slot, you can use the C++ `__declspec(thread)` modifier to turn a static or global variable into a TLS

variable. To do this, instead of writing the code above to `TlsAlloc` and `TlsFree` a slot in `DllMain`, you can simply write:

```
_declspec(thread) int * g_dwMyTlsIndex;
```

You will still need to initialize and free the array itself, however, on a per thread basis. You can do this inside your own `DllMain` thread attach and detach notification code.

When you use `_declspec(thread)`, the compiler will perform all of the necessary TLS management during its own custom `DllMain` initialization and produces more efficient code when reading from and writing to TLS. Static TLS is substantially faster than dynamic TLS because the compiler has enough information to emit code during compilation that accesses slot addresses with a handful of instructions versus having to make one or more function calls to obtain the address, as with dynamic TLS. The compiler knows the three pieces of information it needs to create code that calculates a TLS slot's address: the TEB address (which it finds in a register), the slot index (known statically), and the offset inside the TEB at which the TLS array begins (constant per architecture). From there, it's a simple matter of some pointer arithmetic to access the data inside a TLS slot.

There are limitations around when you can use static TLS, however. You can only use it from within a program or a DLL that will only be linked statically. In other words, it cannot be used reliably when loaded dynamically via `LoadLibrary`. If you try, you will encounter sporadic access violations when trying to access the TLS data.

Managed Code TLS

Similar to native code, there are two modes of TLS access for managed code. But unlike native code, neither has strict limitations about which kind can be used in any particular program. A single program can, in fact, use a combination of both without worry that they will interact poorly with one another.

Thread Statics. The `ThreadStaticAttribute` type is a custom attribute that can be applied to any static field. (While neither the compiler nor

runtime will prevent you from placing it on an instance field, doing so has no effect whatsoever.) This has the effect of giving each thread a separate copy of that particular static variable. For example, say we had a class `C` with a static field `s_array` and wanted each thread to have its own copy:

```
class C
{
    [ThreadStatic]
    static int[] s_array;
}
```

Now each thread that accesses `s_array` will have its own copy of the value. This is accomplished by the CLR managing an array of TLS slots hanging off the managed thread object. All references to this field are emitted by the JIT as method calls to a special helper function that knows how to access the thread local data. Managed TLS access is slower than static TLS in native code because there are extra hidden function calls and many more indirections.

All call sites that access the variable must check for lazy initialization. There is no direct equivalent to `DllMain`'s attach and detach notifications that can be used for this purpose. Even if a static field initializer is provided, it will only run the first time the variable is accessed (which only works for the first thread that happens to access it). Detach notifications are unnecessary because data store in TLS variables will be garbage collected once the thread dies. It's a good idea, however, to set TLS variables to `null` when they are no longer necessary, particularly if the thread is expected to remain alive for some time to come.

Dynamic TLS. Thread statics are (by far) the preferred means of TLS in managed code. However, there are some circumstances in which you may need more dynamic in the way that TLS is used. For example, with thread statics, the TLS information you need to store must be decided statically at compile-time, and you are required to arrange for a static field to represent the TLS data. Sometimes you may need per object TLS. Dynamic TLS allows you to create slots in this kind of way, very similar to how dynamic TLS in native code works.

To use dynamic TLS, you first allocate a new slot. Two kinds of slots are available, those accessed by name and unnamed slots accessed via a slot object. These are allocated with the `AllocateNamedDataSlot` and `AllocateDataSlot` static methods on the `Thread` class.

```
public static LocalDataStoreSlot AllocateNamedDataSlot(string name);
public static LocalDataStoreSlot AllocateDataSlot();
```

When specifying a named slot, the name supplied must be unique, or else an `ArgumentException` will be thrown. In both cases, a `LocalDataStoreSlot` object will be returned. In the case of `AllocateDataSlot`, you must save this object in order to access the slot. If you lose it, you can't access the slot ever again. For named slots, there is a method to look up the slot, though saving it can avoid unnecessary subsequent lookups.

```
public static LocalDataStoreSlot GetNamedDataSlot(string name);
```

`GetNamedDataSlot` will lazily allocate the slot if it hasn't been created already.

Once a slot has been created, you may set and get data using the `SetData` and `GetData` static methods, respectively. Each accepts a `LocalDataStoreSlot` as an argument, and enables you to store and retrieve references to any kind of object.

```
public static object GetData(LocalDataStoreSlot slot);
public static void SetData(LocalDataStoreSlot slot, object data);
```

Last, it is important to free named slots when you no longer need them with the `Thread` class's `FreeNamedDataSlot` static method.

```
public static void FreeNamedDataSlot(string name);
```

If you fail to free a named slot, it will stay around until the `AppDomain` or process exits, and data stored under the slot will remain referenced for each thread that has used it (until the thread itself goes away). The `LocalDataStoreSlot` type has a finalizer, which handles cleanup for unnamed slots once you drop all references to instances. However, the `Thread` object itself keeps a reference to all named slots that have been

created, so even if your program drops all references to it, the slot will not be reclaimed as you might imagine.

Where Are We?

This chapter has reviewed a lot of the basic functionality of Windows and CLR threads. Threads are the underpinning of all concurrency on the Windows OS, and so this foundational knowledge is necessary no matter what kind of concurrency you are using. We looked at the lifetime of threads, including how to start and stop them, in addition to some of the most common attributes of threads such as TLS. Subsequent chapters will build on this information.

The next chapter will do just that and will take the discussion of threads to the next level. It is called Advanced Threads for a reason. This chapter intentionally focused more on the basics while the next chapter intentionally focuses on more low-level and internal details.

FURTHER READING

- A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Second Edition (Addison-Wesley, 2006).
- B. Grunckemeyer. Constrained Execution Regions and Other Errata. Weblog article, <http://blogs.msdn.com/bclteam/archive/2005/06/14/429181.aspx> (2005).
- K. Brown *The .NET Developer's Guide to Windows Security* (Addison-Wesley, 2004).
- C. Brumme. Startup, Shutdown, and Related Matters. Weblog article, <http://blogs.msdn.com/cbrumme/archive/2003/08/20/51504.aspx> (2003).
- S. Currie. Mixed DLL Loading Problem. MSDN documentation, [http://msdn2.microsoft.com/enus/library/Aa290048\(VS.71\).aspx](http://msdn2.microsoft.com/enus/library/Aa290048(VS.71).aspx) (2003).
- J. Duffy. Atomicity and Asynchronous Exception Failures. Weblog article, <http://www.bluebytesoftware.com/blog/2005/03/19/AtomicityAndAsynchronousExceptionFailures.aspx> (2005).
- J. Duffy. The CLR Commits the Whole Stack. Weblog article, <http://www.bluebytesoftware.com/blog/2007/03/10/TheCLRCommitsTheWholeStack.aspx> (2007).

- MSDN. Visual C++: Initialization of Mixed Assemblies. MSDN documentation, [http://msdn2.microsoft.com/en-us/library/ms173266\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/ms173266(VS.80).aspx).
- MSDN. Best Practices for Creating DLLs. MSDN documentation, http://www.microsoft.com/whdc/driver/kernel/DLL_bestprac.mspx (2006).
- M. Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling. *Microsoft Systems Journal*, <http://www.microsoft.com/msj/0197/Exception/Exception.aspx> (1997).
- S. Pratschner. *Customizing the Microsoft .NET Framework Common Language Runtime* (MS Press, 2005).
- S. Toub. High Availability: Keep Your Code Running with the Reliability Features of the .NET Framework. *MSDN Magazine* (October 2005).

4

Advanced Threads

THE PREVIOUS CHAPTER reviewed the basics of Windows and CLR threads. Several other interesting, but less basic, aspects were mentioned only in passing or deferred altogether. This chapter presents some detailed parts of threads, including bits of interesting state comprising them (such as user-mode stacks), how the OS schedules threads, ways that you can control their execution directly, and more. All of this information will come in handy sometime and has been put in a separate chapter to minimize distracting from the fundamental topics needed for concurrent programming.

Thread State

In order to logically represent some in-progress execution, each thread has a large amount of other interesting state associated with it. The most notable piece of state is the stack memory used for function calling and the like, but additional state such as the **thread environment block (TEB)** is also an important part of a thread's physical makeup.

User-Mode Thread Stacks

Each OS thread has a user-mode stack used for execution. A stack is just a contiguous region of memory of fixed size in the enclosing process's virtual address space. Each thread tracks the "current location" in the stack, via a

pointer, which grows downward in the address space. The *beginning* of a stack, thus, has a higher address than its end: as more and more stack space is used, the stack pointer (stored in the ESP register on modern processors) is decremented. X86-inspired processors offer a handful of instructions that use the stack, such as PUSH and POP, to place data onto and to remove data from the stack, respectively, and CALL and RET, which implement function calling by pushing and popping function return addresses.

A thread's stack is used primarily by compilers to implement function calls and to store local variable and argument values that can't remain in registers (e.g., due to register pressure). Many locals are therefore stored on the stack, and some objects are allocated inline on the stack instead of, say, in the heap with a pointer on the stack. In C++ this decision is made by the developer, while in .NET value type locals are allocated on the stack. Both systems also offer ways to allocate raw memory directly on the stack instead of the heap: in VC++, there is an `_alloca` function and in C# you can use the `stackalloc` keyword to create value type arrays. Many system components, including the CLR and the Windows structured exception handling (SEH) subsystem, also store additional information on the stack.

As an example of how function calls use the stack, consider the following C# code. It shows a simple method `Main` (the program's entry point) that calls a method `f`, which calls `g`.

```
class TestProgram
{
    static int Main(string[] args) { return f(1, 5); }

    static int f(int x, int y) { return g(x + y); }

    static int g(int count)
    {
        int z = count + 6;
        System.Diagnostics.Debugger.Break();
        return z;
    }
}
```

We call the static method `Debugger.Break` inside of `g`. This just manufactures an exception and notifies the debugger, allowing us to stop at a particular point in the program so we can examine the stack. (The same can be accomplished in native code with a call to the Win32 `DebugBreak`

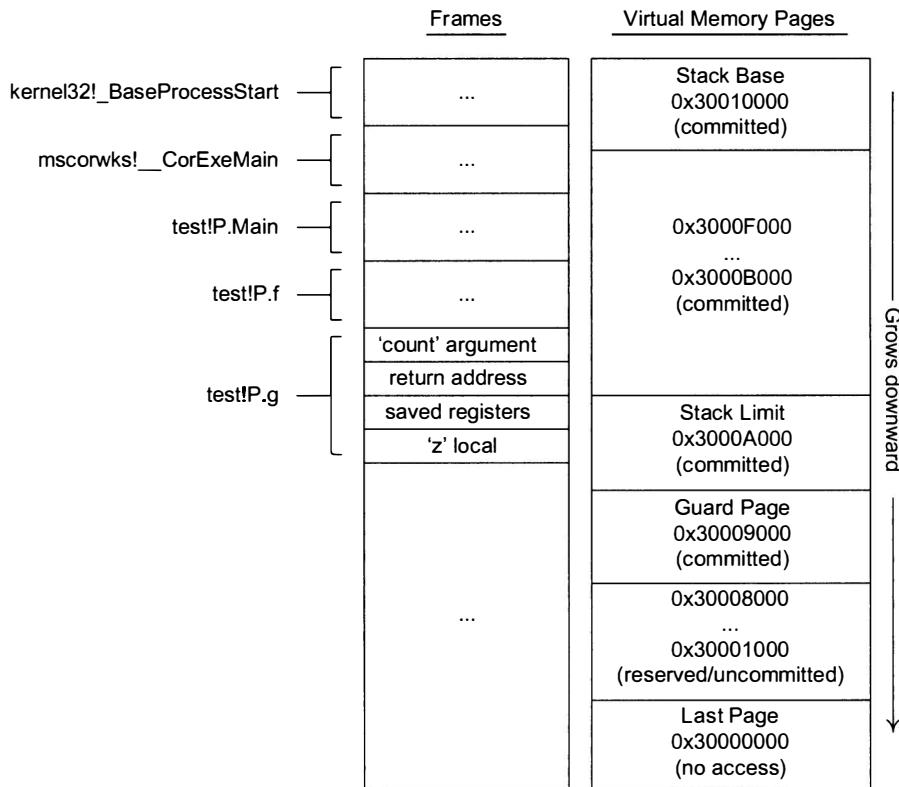


FIGURE 4.1: Graphic depiction of the stack for the above program

function.) If we sketched the stack at this point, it would look something like Figure 4.1. The `_BaseProcessStart` and `__CorExeMain` functions are called automatically by Windows, but eventually we end up in the C# `Main` method.

In our example, each function that has been called on the stack has its own activation frame, containing the arguments supplied by callers, the return address to jump back to after the function has completed, any register values that must be saved on entry and restored on exit, and local variables that the function requires. Because stack grows downward in the address space, the first function's activation frame starts at an address less than the function that it calls. So, for example, the frame for `g` might require 12 bytes on a 32-bit machine: 4 (`sizeof(int)` for the `count` argument) + 4 (`sizeof(void *)` for the return address) + 0 (assuming no saved registers) + 4 (`sizeof(int)` for the local variable `z`). Details about

the precise format of these frames are outside of the scope of this book and depend on the calling convention used by the compiler generating the frames (i.e., `cdecl`, `stdcall`, `fastcall`, or `thiscall`), which is a contract between the caller and callee functions about how registers and the stack are used during function calls.

Most of the details discussed in this section are not necessary to understand in depth during development of concurrent programs, but come in extremely handy when debugging them or simply when trying to understand how the system works. Also note that everything said here applies equally to fiber user-mode stacks (see Chapter 9, Fibers): in some cases, what is said only applies when the fiber is actively running on a thread, such as when getting stack information from the TEB, but in other cases, it doesn't matter. We'll begin with brief overview of stack sizes and how to control them, then specifically how the stack memory is laid out, what happens when stack space is exhausted, and, along the way, we'll also examine some useful stack-related debugger commands.

Stack Reservation and Commit Sizes

There are actually two parts to a thread's stack size: the reserve and the commit size. Windows memory management deals in terms of virtual memory pages, which, for small page configurations (the default), are 4KB apiece in size on X86 and X64, and 8KB on IA64. When memory is allocated, programs may reserve a certain amount up front and later commit those when the program actually needs to write to them. Reserving a page allocates internal virtual memory bookkeeping data structures, but the page will not yet actually consume any physical memory. When it is committed, space in the pagefile is used to back the memory required; eventually, when it is accessed, the pages are brought into physical RAM. While the CLR hides virtual memory almost entirely from developers, memory reservation and commit are exposed directly to Win32 programs via `VirtualAlloc` and `VirtualAllocEx`. These same reserve and commit concepts apply equally to both heap and stack memory.

The sizes of the user-mode stack are determined at thread creation time by one of two things. For the first thread created in a process—that is, the default thread that runs the EXE's entry point code—the size information is

always taken from a special stack size header embedded inside the portable executable (PE) image, which is the format for all Windows binaries. So any compiler or linker that emits a PE image knows how to set the stack sizes. For other threads created during the process's execution, a different stack size argument may be passed explicitly to the thread creation APIs. If an override size is not supplied, new threads use the sizes specified in the executable. The reverse is true also: changing the stack size header has no affect on threads that are created with an explicitly overridden set of values for the commit and reserve sizes.

The default reserve size for all of Microsoft's mainstream runtimes (e.g., the CLR), linkers (e.g., LINK.EXE), and compilers (e.g., VC++ compiler) is 1MB. The CLR always commits the whole stack memory for managed threads as soon as a managed thread is created, or lazily when a native thread becomes a managed thread. This is done to ensure that stack overflow can be dealt with predictably by the execution engine (as examined shortly). Most native Windows linkers and compilers values use just a single page for the default commit size. These defaults are just right for most applications.

It's possible to change the default sizes. There are two main reasons this can be useful. First, when many threads are created in a process, the default of 1MB stack per thread can add a considerable amount of virtual memory consumption to the program. Second, some programs must run code that uses deeply recursive function calls, or otherwise run into stack overflow problems. Typically this should be fixed in the source code, but if you are using a third party or legacy component, increasing the stack size can be a simple workaround.

If your code ends up hosted inside an existing EXE, you will inherit different settings. For instance, ASP.NET uses stack sizes of 256KB to minimize the process-wide stack usage; this was accomplished by modifying the stack settings in the aspnet_wp.exe worker process EXE. So if you write a Webpage, you'll be running within this constraint.

Changing the PE Stack Sizes. In some cases, you might want to change the stack settings yourself, either for the entire EXE or for individual threads that are created. If you need to modify the default stack size, then

you can do so when you build your EXE. Native linkers and compilers typically offer this, while managed code compilers do not. For example, the Microsoft LINK.EXE linker offers a /STACK switch, and the VC++ CL.EXE compiler offers a /F switch. You may also add a STACKSIZE statement to your module definition (.DEF) file.

For instance, here is the format for LINK.EXE and CL.EXE.

```
LINK.EXE ... /STACK:reserveBytes,[commitBytes]  
CL.EXE ... /F reserveBytes
```

You also can modify an existing binary with the EDITBIN.EXE command. This works for native and managed binaries and is the easiest way to change a managed EXE's default stack sizes because you can't do it at build time. This is also sometimes a useful way to work around a stack overflow problem after a program has been deployed—perhaps due to having to operate on a larger quantity of data than expected—without having to recompile and redeploy a program. You specify the reserve and, optionally, the commit bytes via the /STACK switch.

```
EDITBIN.EXE ... /STACK:reserveBytes,[commitBytes]
```

Specifying Stack Sizes at Creation Time. It's possible to specify stack sizes on a per thread basis.

In managed code, the `System.Threading.Thread` class's constructor provides two overloads that accept a `maxStackSize` parameter. As noted earlier, the full stack is committed at creation time for all managed threads, and so the `maxStackSize` parameter represents both the reserve and the commit size: they are effectively the same.

The Win32 `CreateThread` API's `dwStackSize` parameter can be used to override the default values stored in the executable. (For C programs, setting the `stack_size` parameter for `_beginthread` or `_beginthreadex` accomplishes the same thing.) The stack size argument in this case is a number of bytes and will be automatically rounded up to the nearest page allocation granularity (usually 4KB or 8KB). The value will be used as the commit size, and the reserve size is taken from the PE file; alternatively, if `STACK_SIZE_IS_A_RESERVATION` is passed in the `dwCreationFlags` argument (or `initflags` for `_beginthreadex`), the value is used for the reservation size.

instead and the commit size is taken from the PE. If the reservation size is smaller than the commit size, the reservation size is rounded up to the nearest 1MB aligned value that is larger than the commit size.

The following code illustrates overriding the default stack sizes in C# and VC++.

```
// C#:  
Thread t1 = new Thread(MyThreadStart, 1024 * 512);  
  
// VC++:  
HANDLE t2 = CreateThread(  
    NULL, 1024 * 512, &MyThreadStart, NULL, NULL, &dwThreadId);  
HANDLE t3 = CreateThread(  
    NULL, 1024 * 512, &MyThreadStart,  
    NULL, STACK_SIZE_PARAM_IS_A_RESERVATION, &dwThreadId);
```

Because of the defaults noted previously, the resulting stack sizes for these threads are as follows: t1 reserves 512KB (64 pages on IA64, 128 otherwise) and commits the entire stack (512KB); t2 reserves 1MB (128 pages on IA64, 256 otherwise, assuming the defaults for most Windows EXEs) and commits 512KB; and, t3 reserves 512KB and commits a single page.

Stack Memory Layout

Each Windows stack has a stack base and stack limit, which collectively represents the active range of memory for any given stack. Because the stack memory is only committed as needed, the active range is almost always a subset of the available, reserved range of memory. The base is the virtual memory address at which the stack begins, exclusive, and the limit is the address of the last committed usable page on the stack, inclusive. (Recall that the stack grows downward, so this convention may be counterintuitive at first.) As already hinted at, the stack limit does not represent the end of the stack's reserved memory: as more stack pages are needed by the program (i.e., as it calls functions, etc.), additional pages are committed on demand, and the stack limit is updated by the OS accordingly. This can continue without problem so long as the limit needn't exceed the bottom of the reserved range of stack memory.

Just beyond the stack limit (i.e., before it in the address space) lies the stack's guard page. Each virtual memory page in Windows can be marked

with attributes to indicate—in addition to whether it is committed or reserved—whether it is read-only, disallows all access, copied when a write is made to it, and so forth. The guard page is merely a committed virtual address page marked with a special PAGE_GUARD page protection attribute. When memory with this attribute is accessed, the attribute is cleared and the OS will raise a STATUS_GUARD_PAGE_VIOLATION exception. While you can use this attribute for other kinds of memory, the OS uses this as an indication that it needs to commit the next page of stack memory. It catches the exception, commits the next page of the stack, marks it as the new guard page, and then resumes at the faulting instruction. If that new guard page is ever accessed, the whole thing happens again: this is how the stack grows dynamically. This is also when the OS will raise an ERROR_STACK_OVERFLOW exception if it notices that there is no more room for a guard page or if there isn't sufficient pagefile space to back an additional guard page. We'll explore stack overflow soon.

Guaranteeing More Committed Guard Space. I've already mentioned that the OS will normally use a single page for the guard region of memory. As of Windows Server 2003 SP1 (server) or Windows Vista (client), however, a program can explicitly request that the OS use larger chunks of memory for the guard region, on a per thread basis. (Note that this is also available on Windows XP X64 edition, but not the 32-bit SKUs.) This is accomplished with the SetThreadStackGuarantee API.

```
BOOL WINAPI SetThreadStackGuarantee(PULONG StackSizeInBytes);
```

The `StackSizeInBytes` argument is a pointer to a `ULONG` containing the number of bytes you'd like to be used for the guard region. After the call returns successfully, the `ULONG` will have been set by the API to contain the old value. You can retrieve the current value without modification by populating the `ULONG` with the value `0` before making the call. If the requested size is smaller than the current guarantee size, the new value is ignored. This API affects only the thread on which it has been called, that is, there isn't a version that accepts a `HANDLE` to any arbitrary thread.

After calling this, the OS will always commit new guard regions on the current thread in increments of whatever region size you supplied. If you

request 32KB, for example, then you will always have 32KB of stack space dedicated to being the guard page. This leads to fewer guard page exceptions. This memory is generally unusable, however, so you can trigger stack overflows more easily this way. If your stack is 1MB, for instance, and you set a guarantee size of 512KB, then the amount of stack space your program can actually use will be reduced to half.

The reason you might want to use this is that it gives more memory that is guaranteed to be committed in which to run stack overflow handling logic. When a stack overflow happens, you typically will not have much stack space in which to do anything. The default of a single page is insufficient to do anything even moderately clever. Some systems need to do clever things, even if that's limited to just logging the failure somehow (e.g., to the Windows Event Log), and `SetThreadStackGuarantee` can help achieve these things. Refer to the section on stack overflow for some more details.

Spelunking in Stack Land. Let's take a look at an actual example. The thread base and limit are stored in the TEB, which can be dumped from a WinDbg session using the `!teb` command. WinDbg also offers the `!vadump` command, allowing you to dump information about virtual memory pages. ("Vadump," as you might have already guessed, is short for virtual address dump. This capability is available through the standalone tool, VADUMP.EXE, which you can download from Microsoft.com.) Using a combination of the two, we can dump some interesting information about a few stacks and take a look at what's going on.

To compare the differences between managed and native thread stacks (e.g., to illustrate that the CLR commits the entire stack up front), let's break into the main method for two nearly identical programs. Dumping the TEB for both reveals these sample values.

Native thread:

```
0:000> !teb  
TEB at 7efdd000  
...  
StackBase: 0000000000180000  
StackLimit: 000000000017e000  
...
```

Managed thread:

```
0:000> !teb  
TEB at 7efdd000  
...  
StackBase: 0000000000180000  
StackLimit: 0000000000179000  
...
```

You'll notice a subtle difference between the two. The managed stack's `StackLimit` is about 5 pages (i.e., 4KB pages, or 20KB) further along than the native stack. This is simply because the amount of code that has run leading up to the `main` method requires more stack to be committed in the case of managed code. The CLR has to invoke various startup routines, load an assembly, run the JIT compiler, and so forth, and so we'd expect more stack to have been used in the process. The CLR also uses `SetThreadStackGuarantee`, causing the OS to move the stack limit in greater increments. Although the CLR commits the whole stack up front with `VirtualAlloc`, the managed thread's `StackLimit` still grows in the usual manner. The only difference is that new guard regions have already been committed in the CLR case, so the only bookkeeping necessary is to move the guard attribute down the stack region.

The real differences arise when we dump the pages associated with each stack using `!vadump`. This command will dump out all of the allocated virtual memory regions in the process, so we'll have to do a little searching to find the pages of interest. Because we know in both cases the stack size is 1MB, we just subtract 1MB from the stack base—which, in this particular case, means `0x180000 - 0x100000` and results in the address `0x080000`. Since we care only about memory in this range, here's a list of all the regions from `0x080000` through `0x180000`, marked with numbers so we can reference them in a moment.

Native stack regions:	Managed stack regions:
(1)	BaseAddress: 0000000000090000 RegionSize: 000000000001000 State: 00002000 MEM_RESERVE Type: 00020000 MEM_PRIVATE
(2)	BaseAddress: 0000000000080000 RegionSize: 00000000000fd000 State: 00002000 MEM_RESERVE Type: 00020000 MEM_PRIVATE
(3)	BaseAddress: 0000000000181000 RegionSize: 000000000001000 State: 00002000 MEM_RESERVE Type: 00020000 MEM_PRIVATE

(4)

BaseAddress:	000000000017d000	BaseAddress:	0000000000182000
RegionSize:	0000000000001000	RegionSize:	0000000000007000
State:	00001000 MEM_COMMIT	State:	00001000 MEM_COMMIT
Protect:	00000104 ... PAGE_READWRITE + PAGE_GUARD	Protect:	00000104 ... PAGE_READWRITE + PAGE_GUARD
Type:	00020000 MEM_PRIVATE	Type:	00020000 MEM_PRIVATE

(5)

BaseAddress:	000000000017e000	BaseAddress:	0000000000179000
RegionSize:	0000000000002000	RegionSize:	0000000000007000
State:	00001000 MEM_COMMIT	State:	00001000 MEM_COMMIT
Protect:	00000004 PAGE_READWRITE Protect:	00000004 PAGE_READWRITE	
Type:	00020000 MEM_PRIVATE	Type:	00020000 MEM_PRIVATE

In native code, there are three distinct regions (2, 4, and 5), and in managed code there are five. Let's inspect each in detail. Because the stack grows downward in the address space, we'll discuss them in the reverse order:

5. The actively used portion of the stack. It is fully committed, backed by the pagefile, and several pages are probably (but not necessarily) resident in RAM. Notice that the `BaseAddress` is equal to the thread's current `StackLimit`, and that `BaseAddress + RegionSize` equals `StackBase`. This is a basic invariant. The thread is actively reading from and writing to its stack memory only within this region, and the `ESP` register is likely pointing inside of it unless stack growth is imminent.
4. The guard region of the stack. Notice that its protection attributes include `PAGE_GUARD`, and that it too is committed. When the stack grows into the guard region, the current pages inside the guard will become part of region 5, and the next pages further down in the stack will become the new guard region. A few things are worth noting. Notice that the guard page is a single page in the native case, but its `RegionSize` is `0x7000` (28KB) in managed. That's because the CLR always uses the `SetThreadStackGuarantee` for managed threads on OSs that support it. It does this in order to make responding to stack overflow and shutting down the CLR cleanly possible.
3. This is the last page of the used portion of the stack and will never truly be committed. It's often referred to as the "hard guard page"

and is treated specially. If you try to write to it, the OS will immediately terminate your process. In the wink of an eye it's gone, without callbacks or clean shutdown. As the actual guard region moves down the stack, the OS moves this page too.

2. The currently unused portion of the stack. Here you will find the biggest obvious difference between native and managed code: notice the native pages are marked `MEM_RESERVE` while the managed pages are marked `MEM_COMMIT`. Remember, that's because the CLR commits the whole thing up front using `VirtualAlloc`. And as mentioned before, because it uses `VirtualAlloc` directly, the guard page is left intact and must still move around normally.
1. This is the final destination of the hard guard page and is completely unusable. It cannot be committed and attempting to write to it always terminates the process. As the OS moves the guard region downward, the hard guard page remains behind the guard and will "slide into place" in this location once the whole stack has been committed by the program. This particular page is part of region #2 for native stacks, but it is listed separately for the managed stack because it's marked as `MEM_RESERVE` and not manually committed.

Stack Traces. A stack trace is just a textual representation of the current stack's state. Traces are most often used during debugging or error reporting to determine where a problem occurred. For example, the callstack for the program shown at the beginning of this section might have a trace something like this, listing the most recent function call to least recent.

```
test.exe!P.g(int count = 6) Line 13  C#
test.exe!P.f(int x = 1, int y = 5) Line 8 + 0x8 bytes C#
test.exe!P.Main(string[] args = {Dimensions:[0]}) Line 4 + 0xc bytes C#
mscoree.dll!__CorExeMain@0() + 0x34 bytes
kernel32.dll!_BaseProcessStart@4() + 0x23 bytes
```

Typical traces just expose the current function calling chain, including function names, and often useful debugging information such as line numbers. Sometimes, as is in the above example, information about argument values passed to active functions are captured also.

A stack trace will always contain function names for managed assemblies, since they are stored in the assembly's metadata, and whether source line numbers are available depends on whether a PDB was generated (via the C# compiler's /debug switch, for example) and found during trace generation. For unmanaged binaries, on the other hand, a PDB is required (via the VC++ compiler's /Zi switch, for example) in order for traces to contain both function names and line numbers. Specific details often depend heavily on the compiler and debugger in question.

The above stack traces show mscoree.dll's `__CorExeMain@0` and kernel32.dll's `_BaseProcessStart@4` functions. These only show up if you've turned on "Native Debugging" in Visual Studio in the Project Properties window (displayed in the Call Stack window or by running the `>K`, `~*K`, or related commands in the Immediate window), or if you're using a native debugger such as the Kernel Debugger or WinDbg. And even then you may not see what you expect. If you've not configured your system's debugging symbol (PDB) path correctly, the function names for mscoree.dll and kernel32.dll won't even show up. You'll only see names for the functions for which PDBs could be found.

CONFIGURING DEBUG SYMBOLS

To ensure stack trace information shows up for system DLLs, go to Visual Studio's Tools>Options menu, select Debugging>Symbols, and add the location <http://msdl.microsoft.com/download/symbols>. This downloads the symbols from Microsoft's public symbol server. You can also enter a file path in which to cache the symbols (e.g., `c:\symbols`), so that they needn't be downloaded each time you initiate a debugging session that requires them, which is sometimes a time consuming operation. You can also do this via a system-wide environment variable: `_NT_SYMBOL_PATH=SRV*c:\symbols*http://msdl.microsoft.com/download/symbols`.

Stack traces are used in a few other places. CLR exceptions capture the stack trace at the point of a throw to make it simpler to print and/or log the cause of the exception. This is exposed through any `Exception` object's `StackTrace` property, which is just a string.

The .NET Framework also allows you to programmatically capture and inspect a program's stack trace in a more structured format (i.e., not just a string) using the `System.Diagnostics.StackTrace` class. This class offers an array of `StackFrame` instances, each of which has strongly typed information about the trace: file name, file line and column numbers (if the PDB was found when the trace was generated), IL or native offset, and the `MethodBase` (reflection object) for the target method. Calling `ToString` on the `StackTrace` object offers a quick way to obtain a textual trace.

To capture a new trace, instantiate a new `StackTrace` object: the no-argument constructor captures the current thread's stack trace, the constructor accepting an `Exception` captures the stack trace present at the time the target exception was thrown, and the constructor with a `Thread` parameter asynchronously captures some other target thread's trace. Each of these offers an overload that accepts a Boolean parameter, `fNeedFileInfo`, which, if `true`, also generates file information from the PDB file, if available. It is `false` by default.

CAUTION

Capturing a stack trace from another thread while it is running requires that you suspend it first, otherwise you may end up with a corrupt stack trace. This can be done with the `Thread` class's `Suspend` method, as we'll see later; after you are done capturing the trace, you must remember to resume it with the `Resume` method. Thread suspension is generally speaking a dangerous activity, so please first refer to and read the later section if you intend to do this.

Stack Overflow

A stack overflow can happen in two situations:

1. A thread tries to commit more stack pages than it has reserved.
2. Committing a new guard page fails due to lack of physical memory and/or pagefile space.

The former often happens due to application bugs, such as infinite recursion. But it can occur due to deep callstacks, especially if the size of the

stack reservation is smaller than the default of 1MB, as is the case with ASP.NET and WSDL.EXE. Extensive use of stack allocations via C#'s `stackalloc` keyword, fixed arrays, large value types, or VC++'s `_alloca` function can make overflows more likely. A workaround for such situations is to increase the stack size of threads in the program, either by changing the source or by editing the PE file to have larger default stack sizes, as described earlier in this chapter. But in most cases, a better solution is to treat it as a bug and rely less aggressively on stack allocation.

Running out of pagefile space happens only under extremely stressful (and, one hopes, rare) conditions, that is, when there's no free disk space on the machine to back stack memory in the pagefile. Typically there is no way to deal with this programmatically, except to fail as gracefully as possible and perhaps notify the user so that he or she may respond by freeing up resources. It is particularly important, albeit difficult, to ensure user data doesn't become corrupt in such situations. This is often treated similar to out of memory in that it's notoriously difficult to harden libraries and programs to respond predictably in such situations.

Stack overflow is usually catastrophic for Windows programs. Some Win32 libraries and commercial components may respond very poorly to it. For example, a Win32 `CRITICAL_SECTION` that has been initialized so as to never block can end up stack overflowing in the process of trying to acquire the lock. Yet MSDN claims this cannot fail. A stack overflow here can lead to an orphaned critical section at the very least, and can cause subsequent deadlocks. Worse, the `CRITICAL_SECTION` may even become corrupt in some circumstances. This only happens in very low resource conditions, which are difficult to reproduce and test.

Because of the extreme difficulty associated with stack overflow hardening, very little of the library code Microsoft ships, including Win32 and the .NET Framework, can continue operating correctly after a stack overflow has occurred. The core of the Windows OS and the CLR itself are hardened, but usually the only intelligent and conservative response to stack overflow is to terminate the process abruptly.

And that's just what the CLR does (as of 2.0). It reacts to stack overflow by issuing a fail fast (see `Environment.FailFast`). This logs a Windows Event Log entry and immediately terminates the process without unwinding

threads, running finally blocks, or running finalizers. As with any normal unhandled exception, a debugger will be given a first and second chance to debug the process. Previously, in 1.0 and 1.1, a `StackOverflowException` was generated, and could be caught. The new behavior ensures that subtle problems caused by the inability of a component to react to stack overflow are not permitted to run rampant, which would otherwise possibly trigger silent data corruption. CLR hosts such as SQL Server can override this policy, but when they do so they assume all of the responsibility for containing the possible damage.

Unmanaged code can catch a stack overflow exception using an SEH try/catch clause.

```
_try
{
    ...
}
_catch (GetExceptionCode() == STATUS_STACK_OVERFLOW)
{
    ...
}
```

But the same caveats mentioned before still apply. It is extremely difficult to determine when it is or isn't safe to proceed running any code in the process at all. Because the decision is not enforced by a runtime, as is the case with managed code, native applications and libraries are all over the map when it comes to responding to stack overflow. Some Win32 APIs and COM components actually catch stack overflow and try to continue running, for instance.

An overflow due to the first cause above (running out of reserved space) actually happens before the last reserved page is committed. On X86 and X64 platforms, the two last pages, and on IA64, the last three pages, are never used for guard page usage. Instead, they are reserved for executing necessary stack overflow exception handling should the guard ever reach them. For most applications, this still isn't sufficient, however, which is why the CLR uses `SetThreadStackGuarantee` as noted earlier.

The CLR goes a step further and doesn't have to worry about the second cause of stack overflow mentioned earlier. Because the CLR pre-commits all managed thread stacks, stack overflow due to inability to back stacks in the pagefile is simply not possible. These situations are effectively turned into

`OutOfMemoryExceptions` during thread creation. This technique is not without flaws: namely, it puts quite a bit of pressure on the pagefile. For instance, if you create 1,000 threads in a process, you will need 1GB of pagefile space just for their stacks alone. This doesn't eat up physical memory until the pages are written to and faulted into RAM, but managed programs end up using more disk space than their native counterparts.

If a program decides to continue running after a stack overflow has occurred, it is imperative that the guard page is reset. When a stack overflow has occurred, it means there is no longer a page in the stack region of memory with the `PAGE_GUARD` attribute on it. Resetting the guard region can be done manually via the virtual memory Win32 functions (i.e., `VirtualAlloc`) or the CRT's `_resetstkoflw` function. If the stack overflow logic attempts to commit beyond the last page—or if a bug prevents the guard page from being restored and subsequent code overflows the stack again—an access violation exception will occur. This is done to prevent an error in stack overflow from overwriting arbitrary memory below the stack, which could result in security problems. Due to exhaustion of all stack space, this access violation will probably not be handled gracefully. Windows needs user-mode stack space to dispatch exceptions, so if the stack has grown to the point where an access violation happens, it may not be able to do so. Windows detects this and responds by abruptly terminating the process. No error dialog will be shown, no warning is issued, and the process just disappears.

Stack Probes and Reliability. The CLR's policy of failing a process in response to stack overflow without running finally blocks or finalizers could lead to problems for some code. If managed code was amidst a multistep update to some machine-wide persistent state (such as the registry) when an overflow tore down the process, it could lead to corruption. In some cases, corruption is limited to a single process. In others, it may affect the entire system, but will be cleared up with a reboot. In yet other cases, the situation could be more severe. In any case, the user of an end application is likely to be left dissatisfied with the experience, and so we'd like to ensure our software minimizes the probability and rate of such occurrences. Instead of

executing arbitrary code after a stack overflow has happened, the CLR permits code to probe for sufficient stack before beginning some operation. A probe attempts to commit a predetermined amount of stack from the current ESP, and, if it fails, the stack overflow occurs immediately. Since this happens entirely before starting the critical operation, you have some assurance that, so long as the critical code runs in under the probe size worth of stack, a stack overflow will not be triggered. The code can still accidentally use more than was probed for, in which case all bets are off. Also note that another thread in the system could trigger a stack overflow, leading to the process exiting, so this approach is still not foolproof.

This probing capability is exposed in a number of ways. In its rawest form, you can make a call to the `RuntimeHelpers.ProbeForSufficientStack` API, located in the `System.Runtime.CompilerServices` namespace. It checks for a hard coded amount of stack space: 12 pages of stack (96KB on IA64, 48KB otherwise). For example:

```
void CriticalFunction()
{
    RuntimeHelpers.ProbeForSufficientStack();
    // We are guaranteed 12 pages of stack to use on this thread here.
}
```

A call to this API is implicit with any constrained execution region (CER) in the CLR, which is denoted by a try-catch-finally block preceded by a call to `RuntimeHelpers.PrepareConstrainedRegions`. The `RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup` API enables you to execute some arbitrary body code and, even if doing so causes a stack overflow, ensures that if the stack is unwound the cleanup code is called, for example in hosted situations like running inside of SQL Server. The body code and cleanup code are both represented with delegates passed to the method. Note that this does not hold in the unhosted case, because the CLR doesn't unwind the stack normally—it just issues a fail fast.

Finally, if you need more than 12 pages or would like to probe for a more precise amount, you can simulate this using C#'s stack allocation feature:

```
unsafe static void ProbeForStackSpace(int bytes)
{
    byte * bb = stackalloc byte[bytes];
}
```

The `ProbeForStackSpace` method takes an integer `bytes` representing the number of bytes to probe for and attempts to stack allocate that much data. If it fails to do so, a stack overflow will be triggered. We'll see later how to rewrite this function to return a `bool` instead of triggering overflow when there is insufficient space.

Internal Data Structures (KTHREAD, ETHREAD, TEB)

A thread's internal state is comprised mainly of three data structures, aside from its user- and kernel-mode stack: the kernel thread (KTHREAD), executive thread (ETHREAD), and thread environment block (TEB). You seldom run into these in everyday programming, but knowing about them can come in handy during debugging and even when writing certain classes of programs. In fact, the KTHREAD and ETHREAD are in the system address space, not user-mode, and so the only structure you can access from user-mode is the TEB. Many Win32 APIs are meant to manipulate fields of these structures without you needing to know that they even exist. In this section, we'll briefly review these data structures at a high level, and see some of the debugging commands that allow you to access them.

The KTHREAD and ETHREAD structures contain a lot of information that is specific to thread management and execution on Windows, for example, thread priority, state, kernel-mode stack addresses, its wait list, owned mutexes, TLS array, and so on. You can dump the contents of these data structures from WinDbg using the `dt nt!_kthread` and `dt nt!_ethread` commands. We won't delve too much into the details of each, since there's quite a bit, and most of it is irrelevant to user-mode (and, in most cases, even kernel-mode!) programming. Please refer to Further Reading, Russinovich and Solomon's *Microsoft Windows Internals* book for more details on these data structures.

Because the TEB is available to user-mode code, we'll review it in a bit more detail. Related, there is a data structure called the thread information block (TIB) which offers additional information about a thread, but which is, like KTHREAD and ETHREAD only accessible to kernel-mode code. The TEB contains things like a pointer to the exception chain, the stack addresses, a pointer to the process environment block (PEB), last error information (from Win32 API calls), and the number of CRITICAL_SECTIONS owned by the thread, among other things.

You can print out TEB information with the !teb command from WinDbg.

```
TEB at 7fffd000
ExceptionList:      000ee3a4
StackBase:          00130000
StackLimit:         000eb000
SubSystemTib:       00000000
FiberData:          00001e00
ArbitraryUserPointer: 00000000
Self:               7ffdf000
EnvironmentPointer: 00000000
ClientId:           0000268c . 00002690
RpcHandle:          00000000
Tls Storage:        00000000
PEB Address:        7ffdb000
LastErrorValue:     0
LastStatusValue:    c0000034
Count Owned Locks: 0
HardErrorMode:      0
```

By default !teb will print the active thread's TEB. You can specify the address of another thread's TEB as an argument to !teb. Addresses are printed alongside the threads when you run the WinDbg ~ command to show all threads in the process. There is also a !peb command which prints related information that is stored at the process level instead of per thread.

Programmatically Accessing the TEB

Sometimes it can be useful to access the TEB information from code. To do so, Ntdll.dll exports an undocumented function from WinNT.h.

```
PTEB NtCurrentTeb();
```

The PTEB structure gives you direct access to the current thread's TEB. This function returns you a PTEB, which is defined as `_TEB *`. `_TEB` is an internal data structure defined in `winternl.h`, and consists of a bunch of byte arrays. Directly accessing the raw `_TEB` structure is not recommended. Instead, you can cast the PTEB to a PNT_TIB, which itself is defined in `WinNT.h` as `_NT_TIB *`. This data structure is not actually documented—meaning you can actually rely on it not breaking between versions of Windows—but it also provides access to the TEB's information in a strongly typed way.

Unfortunately, while you are given many of the more interesting fields, you can't access every single bit of information in the TEB via _NT_TIB.

```
typedef struct _NT_TIB
{
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
    union
    {
        PVOID FiberData;
        DWORD Version;
    };
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
}
NT_TIB, *PNT_TIB;
```

As an example of using NtCurrentTeb, the following code simply prints out the current thread's stack base and limit.

```
PNT_TIB pTib = reinterpret_cast<PNT_TIB>(NtCurrentTeb());
printf("Base = %p, Limit = %p\r\n",
       pTib->StackBase, pTib->StackLimit);
```

Believe it or not, this capability can come in useful. For example, this kind of code can be used to determine whether a pointer refers to memory in the heap or the current thread's stack, simply by comparing it with the StackBase and StackLimit from the TEB. For additional ideas on what this capability can be used for, refer to Matt Pietrek's excellent Microsoft Systems Journal Articles in Further Reading (Pietrek, 1996; 1998).

Accessing the TEB via the FS Register. There's a shortcut to access the TEB. You can always find a pointer to the current one in the register FS:[18h] on X86 machines.

```
PNT_TIB pTib;
__asm
{
    mov eax,fs:[18h]
    mov pTib,eax
}
printf("Base = %p, Limit = %p\r\n",
       pTib->StackBase, pTib->StackLimit);
```

Many compilers emit code to access things in the TEB such as the SEH exception chain directly via the FS register versus making one or more function calls and pointer dereferences.

There's another shortcut you can take. Because the FS segmented register has its base set to the TEB itself, you can access fields by specifying offsets. The previous snippet works because, if you look at the _NT_TIB data structure above, the Self pointer is 24 (i.e., 0x18) bytes from the start, assuming a 32-bit architecture with 4 byte pointers. We can use the same technique to access any of the fields. If we want to directly access the stack base and limit, for instance, we can use FS:[04h] for the base and FS:[08h] for the limit.

```
void * pStackBase;
void * pStackLimit;
__asm
{
    mov eax,fs:[04h]
    mov pStackBase,eax
    mov eax,fs:[08h]
    mov pStackLimit,eax
}
printf("Base = %p, Limit = %p\r\n",
       pStackBase, pStackLimit);
```

Unfortunately, the `_asm` keyword is not supported on all architectures and isn't available in managed code, so the above code is only guaranteed to work on X86 VC++. Furthermore, the hard-coded offsets 04h and 08h are clearly wrong on 64-bit architectures: you need more than 4 bytes to represent a 64-bit pointer. `NtCurrentTeb` provides access to the TEB without requiring programs to hard-code all of this architecture specific information.

Example Usage: Checking Available Stack Space. In some rare cases, it might be useful to query for the remaining stack space on your thread and change behavior based on it. As one example, it could enable you to fail gracefully rather than causing a stack overflow. A UI that needs to render some very deep XML tree and does so using stack recursion could limit its recursion or show an error message based on this information, as yet another example. If the UI program finds that it has insufficient stack space,

it may decide that it needs to spawn a new thread with a larger stack to perform the rendering. Or it may log an error message when testing so that the developers can fine tune the stack size or depend less heavily on stack allocations or so the program can show a dialog box and fail.

The TEB's `StackBase` and `StackLimit` fields can be used to determine the active stack range. The `StackLimit` is only updated as you touch pages on the stack and, thus, it's not a reliable way to find out how much uncommitted stack is left. There's an undocumented field, `DeallocationStack`, at `0x0E0C` bytes from the beginning of the TEB that will give you this information, but that's undocumented, subject to change in the future, and is too brittle to be reliable.

The `RuntimeHelpers.ProbeForSufficientStack` function reviewed earlier may appear promising, but it won't work for this purpose. It probes for a fixed number of bytes (48KB on X86/X64), and, if it finds there isn't enough, it induces the normal CLR stack overflow behavior. That will tear your process down, which is not what we want. The same is true of the function shown earlier that uses `stackalloc`.

The good news is that the `VirtualQuery` Win32 function will provide this information. It returns a structure, one field of which is the `AllocationBase` for the original allocation request. When Windows reserves a thread's stack, it does so as one contiguous piece of memory. The memory manager remembers the base address supplied at creation time, and this is the "end" of the stack; that is, it's the same as the `DeallocationStack` from the TEB. If we're in managed code, all we need to do is use P/Invoke to access this information.

Let's create a new version of the `CheckForSufficientStack` function using this API. Unlike the one earlier, which triggers a stack overflow if there isn't enough stack space, our new function takes a number of bytes as an argument and returns a bool to indicate whether there is enough stack to satisfy the request, enabling the caller to react accordingly.

```
public unsafe static bool CheckForSufficientStack(long bytes)
{
    MEMORY_BASIC_INFORMATION stackInfo = new MEMORY_BASIC_INFORMATION();

    // We subtract one page for our request. VirtualQuery rounds up
    // to the next page. But the stack grows down. If we're on the
    // first page (last page in the VirtualAlloc), we'll be moved to
```

```

// the next page, which is off the stack! Note this doesn't work
// right for IA64 due to bigger pages.
IntPtr currentAddr = new IntPtr((uint)&stackInfo - 4096);

// Query for the current stack allocation information.
VirtualQuery(currentAddr, ref stackInfo,
              sizeof(MEMORY_BASIC_INFORMATION));

// If the current address minus the base (remember: the stack
// grows downward in the address space) is greater than the
// number of bytes requested plus the reserved space at the end,
// the request has succeeded.
return ((uint)currentAddr.ToInt64() - stackInfo.AllocationBase) >
    (bytes + STACK_RESERVED_SPACE);
}

// We are conservative here. We assume that the platform needs a
// whole 16 pages to respond to stack overflow (using an X86/X64
// page-size, not IA64). That's 64KB, which means that for very
// small stacks (e.g. 128KB) we'll fail a lot of stack checks
// incorrectly.
private const long STACK_RESERVED_SPACE = 4096 * 16;

[DllImport("kernel32.dll")]
private static extern int VirtualQuery (
    IntPtr lpAddress,
    ref MEMORY_BASIC_INFORMATION lpBuffer,
    int dwLength);

private struct MEMORY_BASIC_INFORMATION
{
    internal uint BaseAddress;
    internal uint AllocationBase;
    internal uint AllocationProtect;
    internal uint RegionSize;
    internal uint State;
    internal uint Protect;
    internal uint Type;
}

```

Notice that we have to consider some amount of reserved space at the end of the stack because, as we reviewed earlier, at least a few pages are reserved for stack overflow handling. The code above assumes 16 4KB pages are required; this is more than is typically needed, so it may lead to false positives (but we hope no false negatives). Also note the program above is very X86/X64 specific and won't work reliably on IA-64: it hard codes a 4KB page size. It's a trivial exercise to extend this to use information

from `GetSystemInfo` to use the right page size dynamically. If this function returns true, you can be guaranteed that an overflow will not occur, except for scenarios in which the guard page size has been modified with a previous call to `SetThreadStackGuarantee`.

Contexts

When a context switch removes a thread from a processor, the OS will capture its volatile register state, among other things, so that it can be subsequently restored when it is appropriate for the thread to run again. The resulting state is stored inside of a `CONTEXT` data structure. This data structure, in addition to the `GetThreadContext` and `SetThreadContext` methods, are all accessible from user-mode code, enabling you to capture a thread's current context for inspection and even allow you to restore a separate `CONTEXT` to an existing thread, respectively. These are very powerful capabilities.

```
BOOL WINAPI GetThreadContext(HANDLE hThread, LPCONTEXT lpContext);
BOOL WINAPI SetThreadContext(HANDLE hThread, const LPCONTEXT lpContext);
```

Both accept a `HANDLE` to the target thread, and a pointer to a `CONTEXT`. `GetThreadContext` will populate the target structure, while `SetThreadContext` will copy state from the provided structure to the target thread. Both functions return `FALSE` to indicate failure. It is illegal to call either of these on a thread that is actively running. The function will not necessarily fail if you do so, but the resulting `CONTEXT` state will likely be corrupt. Instead, you must use thread suspension (see `SuspendThread` and `ResumeThread` below) to guarantee the thread is not running during context capture or restore.

The `CONTEXT` structure itself varies from processor to processor because each of its fields corresponds to a separate register on the CPU. To do anything meaningful with the context, you will usually have to write `#ifdef'd` code that accesses different registers based on whether the CPU architecture is X86, X64, IA64, etc. There are some register names in common among architectures—such as `EIP`, `EAX`, `EBX`, `ESP`, etc.—so sometimes architecture specific code isn't strictly necessary.

Note that `CONTEXT` has a field, `ContextFlags`, that controls the behavior of `GetThreadContext` and `SetThreadContext`. When set, it restricts the registers captured or restored to a subset of the registers available on the

processor. `CONTEXT_ALL` specifies that the full context should be captured, and other possible values include things such as `CONTEXT_CONTROL`, `CONTEXT_DEBUG`, `CONTEXT_FLOATING_POINT`, among others, each of which represents some collection of the register state. The possible values vary by processor architecture and are usually masked together, so refer to `WinNT.h` for the possible settings.

Contexts also are used during exception handling and are accessible from SEH exception handlers to aid in the determination of an exception's cause. The `GetExceptionInformation` routine returns a pointer to an `EXCEPTION_POINTERS` data structure, which is just two pointers: one refers to an `EXCEPTION_RECORD` containing details about the exception code and faulting address, and the other refers to a `CONTEXT` containing the register state at the time of the exception itself. These details often come in handy when determining how to respond to an exception, particularly for systems code, restartable exceptions, and also for debuggers.

Inside Thread Creation and Termination

Now we will take a look at how thread creation and termination work internally.

Thread Creation Details

When Windows creates a new thread, regardless of whether initiated by Win32 or the .NET Framework APIs, the following steps are performed (in roughly this order).

1. Important thread specific data structures, such as the `KTHREAD`, `ETHREAD`, and `TEB`, are allocated. We reviewed these structures above. Additionally, structures required for asynchronous procedure calls (APCs), local procedure calls (LPCs), memory management, I/O, mutex ownership, and thread creation information are allocated and initialized. A unique thread ID is generated.
2. The thread's context, which is comprised of CPU specific register information, is allocated. This results in a `CONTEXT` that is subsequently used to capture and restore processor state during

context switches. This data structure is accessible from the `GetUserContext` Win32 API.

3. The user-mode stack in the process's address space is created. The amount of stack memory that is reserved and committed for this thread can be controlled with parameters to thread creation and/or configuration, as described earlier. The kernel-mode stack is then created and initialized.
4. The Windows subsystem process, `CSRSS.exe`, is notified of the new thread, which gives it a chance to record information necessary to initialize the thread's state and execute it.
5. The first thread in a process must complete the process initialization before executing the thread start routine, which includes loading required DLLs, notifying any debuggers attached to the process's debugging port, initializing system services, initializing TLS and related data structures, and sending a `DLL_PROCESS_ATTACH` notification to all of the DLLs loaded into the process via their `DllMain` functions.
6. Deliver `DLL_THREAD_ATTACH` notifications to all DLLs in the process.
7. If `CREATE_SUSPENDED` was not set when the thread was created, the thread is resumed, meaning that the thread immediately becomes runnable. This permits the Windows thread scheduler to assign it to a processor for execution. After this occurs, the thread will begin execution in the thread's thread state routine.
8. The creation function returns. In the case of Win32's `CreateThread`, the return value is the new thread `HANDLE`, and the output thread ID parameter is set to the unique identifier assigned to the thread earlier.

Thread Termination Details

As we've seen, the thread termination process differs slightly depending on whether a thread is exited cleanly or terminated abruptly with `TerminateThread`. In any case, just as there are common steps taken during thread creation, there are some steps that are common during thread termination. Notable exceptions are mentioned in line.

1. Send `DLL_THREAD_DETACH` notifications to each DLL loaded in the process. `TerminateThread` API skips this step.
2. The thread kernel object is set to a signaled state. Signaling the thread object means you can use the thread's `HANDLE` as you would any other Win32 synchronization event or primitive. We'll see in Chapter 5, Windows Kernel Synchronization, how you can use this signal to wait for another thread to exit.
3. Free the user-mode stack. As with DLL notifications, `TerminateThread` does not perform this particular step. Instead, the user-mode stack for abruptly terminated threads will be freed when the process itself finally exits.
4. Any internal kernel-mode data structures, including the stack, context, TEB, TLS memory, and other data structures that are specific to a thread and which were mentioned earlier during creation are freed.

Thread Scheduling

We'll explore the way Windows schedules threads onto hardware processors in this section. We also will take a look at some APIs that can be used to influence the kernel thread scheduler's decisions, such as restricting on which processors a certain thread is allowed to run, among other things. For a very detailed overview of the internals of the Windows scheduler, please refer to Russinovich and Solomon's excellent *Microsoft Windows Internals* book (see Further Reading).

As of Windows 95 and Windows NT, the Windows OS uses **preemptive scheduling** for all threads on the system, also known as time-slicing. The term preemptive scheduling means that Windows may interrupt a thread in order to let another thread run on its current processor, in contrast to the alternative of cooperative scheduling, in which a thread itself must explicitly relinquish its execution privileges before another thread can run on its current processor. (Windows offers limited support for cooperative scheduling, as we explore further in Chapter 9, Fibers.) Preemption is used to ensure that threads are given a fair and roughly equal amount of execution time, given the available hardware. When a thread runs, it is preempted if

it exceeds its quantum—which is just a specific period of time that varies from one OS SKU to the next. If there are other threads waiting to execute when the quantum expires, the OS may use a context switch to allow the other thread to run on the processor instead.

The Windows thread scheduler is also priority based. All processes in a system are given a priority class and individual threads within those processes may be assigned even finer-grained priorities. The scheduler will always prefer to run the thread with the highest priority in the system and will preempt lower priority threads that are already running should a higher priority thread become runnable. There are some exceptions in which the OS will let another lower priority thread run before a higher priority one, normally to combat the possibility of starvation; this can happen if there are always higher priority threads ready to run, because they would otherwise always get preference over the lower priority threads.

The scheduler is strictly thread based and not process based at all. This means, for example, that if there are two processes running, one of which has nine always running threads and the other one, all at equal priority, then the first process will receive 90 percent of the processor time while the other gets the remaining 10 percent. (Each thread gets 10 percent.) People often expect that each process will receive a fair amount of processor time—in this case, that would mean that both processes will receive 50 percent apiece—but Windows does not work this way.

Thread States

A thread goes through a transition between several logical states throughout its execution.

- Initialized (0): currently being allocated and initialized by the OS.
- Ready (1): ready to run (a.k.a. runnable) and is in the thread scheduler's dispatcher database. After a thread has been initialized, it transitions into this state, so long as the `CREATE_SUSPENDED` flag was not passed.
- Running (2): actively running on a processor.
- Standby (3): has been selected to run on a processor, but has not physically begun executing yet. It is no longer under consideration

in the dispatcher queue, and may or may not make it to Running depending on whether the thread is context switched out beforehand. There is a state that was added to Windows Server 2003, Deferred Ready (7), which effectively indicates the same condition.

- Terminated (4): has finished running code, and will be destroyed once all outstanding `HANDLE`s to its object are closed.
- Waiting (5): not under consideration for execution by the thread scheduler. A transition to this state is made anytime a thread voluntarily sleeps, waits on a kernel synchronization object, or performs an I/O activity. Thread suspension also places the suspended thread into the Waiting state until it has been resumed, thus threads created with the `CREATE_SUSPENDED` flag transition directly from Initialized to Waiting after creation.
- Transition (6): this state reflects the fact that a thread could otherwise be runnable, but is temporarily ineligible because some important pageable kernel memory needed for to run has been paged to the disk, for example, kernel-mode stack. The thread will transition back to Ready once the data is faulted back into physical memory.

While there are no simple Win32 APIs accessible to query a thread's state, you can access it through performance counters. You can access the performance counter APIs or simply view them in the Windows Performance Monitor (`perfmon.exe`) application. The counter "Thread\Thread State" reports back the current state number (see above) for a particular thread. Related, there is also a "Thread\Thread Wait Reason" counter, which indicates the reason a thread is in the Waiting state. The possible values here follow.

- Executive (0): waiting for a kernel executive object to become signaled, such as a mutex, semaphore, event, etc.
- Free Page (1): waiting for a free virtual memory page.
- Page-in (2): waiting for a virtual memory page to be backed by physical RAM, that is, to be paged into memory.
- Page-out (12): waiting for a virtual memory page to be paged out to disk.

- System allocation (3): the OS is in the process of allocating some system resource the thread needs in order to proceed with execution. This usually means space is needed from the OS paged or nonpaged pool.
- Execution delay (4): thread execution has been delayed by the OS.
- Suspended (5): has been suspended explicitly, either by passing the `CREATE_SUSPENDED` flag during creation or with the `SuspendThread` API.
- Sleep (6): a request has been made to explicitly place the thread into a wait state, usually by one of the thread sleep APIs.
- Event pair high (7) and low (8), and LPC receive (9) and send (10): used internally only. A LPC is used internally by Windows for interprocess communication, for example, with protected subsystem processes like `CSRSS.exe`. These indicate a send or receive is in progress. Event pairs are used during this communication.

Both the thread state and wait reason are available from the managed `ProcessThread` class in `System.Diagnostics`. It offers a `ThreadState` and `ThreadWaitReason` property, which internally query the performance counters and produce a nice enum value to work with instead of requiring memorizing these values.

Also note that each managed thread has a separate kind of state. The above state is managed by the OS and can only be retrieved in user-mode through performance counters. But the CLR also tracks its own state during important transitions, for its own internal bookkeeping, which is accessible from the normal `System.Threading.Thread` object. It has a `ThreadState` property that returns an enum value of type `ThreadState`. The set of states reported by this are slightly different than the aforementioned. In addition, some of these states reflect a mutually exclusive thread state while others are merely thread attributes. A thread's state will always report one from the former and 0 or more of the latter.

We'll review the former first. The names are the enum values themselves:

- `Unstarted` (8): the thread object has been created, but has not been started yet (e.g., with a call to the `Start` method).

- **Running** (0): either ready to run or is actually running on a processor. This does not necessarily mean the thread is physically running. This point can be confusing at first, particularly when coming straight from an explanation of the OS states used. The CLR doesn't know (as the OS does) when a thread is running on a processor or not.
- **WaitSleepJoin** (32): indicates the thread is currently waiting for a kernel object, another thread, or has explicitly slept for a certain period of time. This does not include threads that are blocked on I/O.
- **Suspended** (64): temporarily suspended, due to a call to `Thread.Suspend`.
- **Stopped** (16): has completed execution and is no longer actively running code.
- **Aborted** (256): has been aborted (see the thread aborts section earlier for details), but has not yet completely shut down.

Note that the `Thread.IsAlive` property returns a `bool` indicating whether the thread is still alive, that is, that its `ThreadState` does not contain the stopped state.

And here are the various flags attributes.

- **Background** (4): indicates that the thread is a background, versus foreground, thread. We reviewed background threads earlier in passing. In summary, this means the thread will not keep the process alive. Once all nonbackground threads exit, the process will exit.
- **StopRequested** (1): in the process of being terminated.
- **SuspendRequested** (2): in the process of being suspended.
- **AbortRequested** (128): a thread abort has been requested, but has not yet been processed yet. This is normally because the target thread is still in a delay-abort region. As soon as it leaves such a region it will process the abort request.

Because the CLR manages all of the states, some may become out of sync with what is actually happening. For example, if a native component

suspends a managed thread, that thread will be in a suspended mode, but its state will not report back `Suspended` if queried. Similarly, if a `P/Invoke` into a native API ends up blocking the calling thread on a native synchronization object, the CLR will not know to update the managed thread's state to `Wait-SleepJoin` and therefore it will incorrectly report back `Running` as its state.

Priorities

Because thread priorities are so fundamental to how the Windows thread scheduler works, it's important to understand them. It's particularly important to understand them, because only then will you appreciate why you should avoid using them under most circumstances. Priorities are not as simple as you might at first imagine because the priority, from the scheduler's standpoint, is comprised of two components: the process's **priority class** and the individual thread's **relative priority**. These things taken together form a numeric **priority level**, which falls in the range of 1 to 31, inclusive.

Higher levels indicate higher priorities. Process priority classes are furthermore organized into so-called dynamic (1–15) and real-time (16–31) ranges. There is only a single class within the real-time range, but there are several within the dynamic range. Each class has a default level within the range which threads will, by default, get assigned; however, relative priorities can be set on individual threads to add or subtract an offset from this default.

In Win32, a process's priority class can be set via `SetPriorityClass` or retrieved via `GetPriorityClass`. Each of these functions takes a `HANDLE` to the target process.

```
BOOL WINAPI SetPriorityClass(HANDLE hProcess, DWORD dwPriorityClass);
DWORD WINAPI GetPriorityClass(HANDLE hProcess);
```

In the .NET Framework, you can change a process's priority class with the `System.Diagnostics.Process` class; this type offers a `PriorityClass` property, which accepts a value of the enum type `ProcessPriorityClass`.

```
public class Process
{
    public ProcessPriorityClass PriorityClass { get; set; }
    ...
}
```

Table 4.1 lists all of the priority classes along with their constants and levels:

TABLE 4.1: Windows priority classes and Win32 and .NET enum values

Title	Win32 Constant Value	.NET Enum Value	Level Range	Default
Real-time	REAL_TIME_PRIORITY_CLASS	RealTime	16–31	24
High	HIGH_PRIORITY_CLASS	High	11–15	13
Above Normal	ABOVE_NORMAL_PRIORITY_CLASS	AboveNormal	8–12	10
Normal	NORMAL_PRIORITY_CLASS	Normal	6–10	8
Below Normal	BELOW_NORMAL_PRIORITY_CLASS	BelowNormal	4–8	6
Idle	IDLE	Idle	1–6	4

Each thread may furthermore be assigned a relative priority. In Win32, a thread's priority may be set with `SetThreadPriority` and similarly can be retrieved with `GetThreadPriority`.

```
BOOL WINAPI SetThreadPriority(HANDLE hThread, int nPriority);
int WINAPI GetThreadPriority(HANDLE hThread);
```

And in the .NET Framework, the managed thread class, `System.Threading.Thread`, offers a `Priority` property that accepts values of the enum type `ThreadPriority`.

```
public class Thread
{
    public ThreadPriority Priority { get; set; }
    ...
}
```

(Note that the `System.Diagnostics.Process` class also offers a `PriorityLevel` property, which also allows you to adjust a thread's relative

priority. Using it, however, is discouraged. Setting a managed thread's priority via the `Thread` class enables the CLR to do additional bookkeeping which is used, for example, to reset priorities if a thread is accidentally returned back to the thread pool with a higher priority than normal.)

There are seven possible relative priority offsets you may assign to a thread, two of which are not supported in managed code (unless you use `ProcessThread`, which supports all seven). Most of these offsets either add or subtract a constant, though two of them effectively set the thread's priority level to an absolute value depending on the process priority class. They are shown in Table 4.2.

TABLE 4.2: Windows relative priorities and Win32 and .NET enum values

Title	Win32 Constant Value	.NET Enum Value	Level Modifier
Time Critical	THREAD_PRIORITY_TIME_CRITICAL	n/a (not supported)	Absolute value: 31 for real-time range, 15 for dynamic range
Highest	THREAD_PRIORITY_HIGHEST	Highest	+2
Above Normal	THREAD_PRIORITY_ABOVE_NORMAL	AboveNormal	+1
Normal	THREAD_PRIORITY_NORMAL	Normal	+0 (default)
Below Normal	THREAD_PRIORITY_BELOW_NORMAL	BelowNormal	-1
Lowest	THREAD_PRIORITY_LOWEST	Lowest	-2
Idle	THREAD_PRIORITY_IDLE	n/a (not supported)	Absolute value: 15 for real-time range, 1 for dynamic range

To take an example, imagine we have a process with the default priority class of Normal (8). When we create a thread, it will also by default be given the Normal relative priority (+0). Therefore, the thread's level is 8. If we were to instead assign the thread a different relative priority, say, Highest (+2), then this thread would have a level of 10 (8 + 2). If, on the other hand, we gave a thread Highest relative priority (+2) inside of a process that has a priority class of High (13), then the thread's resulting priority level would be 15 (13 + 2), the highest possible priority level in the dynamic range.

Notice that the default real-time priority level (24) plus `THREAD_PRIORITY_HIGHEST` or minus `THREAD_PRIORITY_LOWEST` still leaves many levels inaccessible. That is, $24 + 2$ is 26, yet the maximum in the real-time range and class is 31, and similarly $24 - 2$ is 22, yet the minimum is 16. This is why `SetThreadPriority` takes an `int` as its argument. To access the other values in the range, you can pass values here by hand: -7, -6, -5, -4, -3, 3, 4, 5, and 6.

On Windows Vista and Server 2008, a new feature called I/O Prioritization has been added. This regulates the scheduling of I/Os because contention for the disk can artificially boost the priority of lower priority processes and threads by allowing them to interfere with higher priority ones. Five priorities are used: Critical, High, Medium, Low, and Very Low. Assignment of priority to an I/O request is handled primarily by the OS and drivers, although you have some control over it by assigning thread priorities. By default, all I/O under a priority of Medium, but you may pass the value `PROCESS_MODE_BACKGROUND_BEGIN` to `SetPriorityClass` to lower the I/O Priority to Very Low, and `PROCESS_MODE_BACKGROUND_END` to revert it. Similarly, you can pass `THREAD_MODE_BACKGROUND_BEGIN` to the `SetThreadPriority` function to lower I/O Priority for that particular thread, and `THREAD_MODE_BACKGROUND_END` to revert this change. This is used by programs such as the Windows Search Indexer to prevent it from interfering with other interactive applications.

Now that we've seen how priority level is calculated and how to adjust priority classes and thread relative priorities, some words of warning are appropriate. Any priorities over the Above Normal class should be avoided almost entirely. Using them will interfere with other system services that usually run at high priorities within the dynamic range, possibly causing hangs and system instability. Using real-time priorities is discouraged even

more strongly. Many device drivers, interrupts, and kernel services, like the memory manager, run in this range. And, as you might imagine, given the naming, any delays can cause serious trouble, possibly even data corruption if system services cannot respond to requests within a certain window of time. Most programs and threads should use the default priority level (Normal/Normal) and leave it to the thread scheduler to ensure they are given a fair chance to execute.

Quantums

A **quantum** is the amount of time a thread is permitted to run before possibly being preempted so that the scheduler can run another runnable thread on the processor. The specific interval used for thread quantum varies between machines, server, and client OSs and can be modified through configuration. Quantum are based on the system clock interval that, on most modern systems, ranges from 10 milliseconds to 15 milliseconds per interval. The default quantum time on Windows client OSs (e.g., Windows 2000, XP, and Vista) is 2 clock intervals. The default time on server OSs (e.g., Windows Server 2000, Server 2003, and Server 2008) is 12 clock intervals. Client quantum are shorter than server quantum to increase responsiveness and provide fairer scheduling of threads on the system. Contrast this with a server program in which throughput and performance are usually of more importance, where shorter quantum usually mean more context switching and worse performance.

You can explicitly select the default client or server settings on any SKU by going to the Advanced settings tab in your Computer's System Properties configuration. Select Performance Settings and choose Advanced. You will see a dialog that says "Adjust for best performance of" with two options: either "Programs" or "Applications" (depending on the specific OS), which selects the client settings, or "Background services," which selects the server settings. There is also a system registry key, `\HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation`, which enables you to tune the quantum settings even more. A detailed discussion of this capability is not included in this book; please refer to Further Reading, Windows XP Embedded Team, for details.

Quantum accounting is done inside of an interrupt routine in the OS. When this interrupt fires, the actively running thread's quantum counter

is decremented; if the quantum expired, a context switch is triggered, which may result in a new thread preempting the current one. If the quantum has not been exhausted, the thread remains running. Note that when a thread voluntarily blocks, its quantum remains intact. So if a thread has nearly exhausted its quantum and blocks, for instance, then when its wait is satisfied it may not run for a full quantum.

Modifications to the thread scheduler's quantum accounting algorithm were made in Windows Vista and Server 2008. Two problems existed on previous versions of Windows that could lead to unfairness and unpredictability in the way that thread execution times were measured. The first is that interrupts that executed in the context of a thread would count towards that thread's quantum. Say that a thread's quantum was 15 milliseconds and 5 milliseconds of that time were spent executing interrupts; in this case, the thread would only be running its code for 10 milliseconds. Vista no longer accounts for interrupt time when deciding whether to switch out a thread. The second problem was that the scheduler didn't account for threads being scheduled in the middle of a quantum interval. The OS uses a timer interrupt routine to account for execution time. If this timer was set to execute every 15 milliseconds and some thread was scheduled in the middle of such an interval, say after 5 milliseconds, then when the timer fired next the OS would charge the thread for the full 15 milliseconds, when in fact it only ran for 10 milliseconds. Vista prefers to undercharge threads instead. This same thread would run for nearly a full timer interval longer than it should—since the granularity of the timer routine remains the same—but ensures threads are not unfairly starved.

Priority and Quantum Adjustments

A thread's priority or quantum will receive special treatment by the Windows thread scheduler under some circumstances. This includes temporary boosts due to various events of interest—such as a GUI thread receiving a new message, starvation detected by the scheduler, etc.—or due to the new multimedia class scheduler that Windows provides as of Vista.

Temporary Boosting

There are several circumstances during which a thread will receive a temporary boost to its priority, its quantum, or both. When a boost occurs, the

thread's relative priority is incremented by a certain number depending on the circumstance. Windows only boosts thread priorities for threads in the dynamic range and will never boost a thread's priority into the real-time priority range (i.e., above absolute priority 16). Once a thread's priority has been boosted, its priority level will subsequently "decay" by -1 for each quantum that passes while it is running, until it returns back to the original priority level. If a thread is preempted mid-quantum, it will still continue to enjoy the benefits of the boost when it is scheduled to run next.

The circumstances are as follows.

- Windows has a service called the **balance set manager**. It runs asynchronously on a system thread looking for starved threads; these are threads that have been waiting to run in the ready state for 4 seconds or longer. If it finds one, it will give the thread a temporary priority boost. It always boosts a starved thread's priority to level 15, regardless of its current value. This is done to combat starvation, for instance, when many higher priority threads are constantly running such that lower priority threads never get a chance to execute.
- When a thread wakes up because the event or semaphore it was waiting on has become signaled, the thread enjoys a temporary priority boost of +1. This is applied to the thread's base priority, so if the thread is already enjoying a priority boost, the effect will not be cumulative. This is done to improve throughput and, in part, in an attempt to avoid **lock convoys**. We'll see in Chapter 6, Data and Control Synchronization, that additional improvements have been made to Windows locks to avoid convoys, rendering the priority boosting technique here effectively redundant.
- When a GUI thread wakes up due to a new message being enqueued into its window's message queue, it receives a temporary priority boost of +2. This is done to improve the responsiveness of interactive applications, in which a new message typically triggers a user visible side effect and thus should be done as quickly as possible to avoid perceptive delays in the user interface.
- When a thread wakes up due to the completion of an I/O, it receives a temporary priority boost of +1. This is done to improve both throughput and responsiveness. Often the completion of I/O on a

server is “chunked,” meaning the server will issue additional I/O when another completes; the boost allows the thread to initiate the additional I/O sooner. But on client-side programs, there may be some user visible action taken at the completion of an I/O, and the boost also ensures that this effect happens sooner.

- Whenever a thread in the foreground process completes a wait activity—defined by the process window that has the current focus in Explorer—it receives an additional priority boost of +1 or +2, depending on system configuration. Unlike other boosts, this boost is additive and will be applied to the thread’s current priority, no matter if it has already been boosted or not. So if the thread woke up due to an event, semaphore, I/O, or GUI message, it receives that boost plus the special foreground priority boost.
- On client OS SKUs (i.e., any installation configured with the “Programs” setting mentioned above in the context of Performance Settings), all threads in the foreground process receive a quantum boost so long as the process remains in the foreground. This boost multiplies the quantum for all threads by three. So for example, instead of having a quantum of 2 clock ticks on client machines, these threads have quantums of 6 clock ticks. This reduces context switches and allows the program to maintain responsiveness.

You can turn off dynamic priority boosting with the `SetThreadPriorityBoost` API, and you can query whether boosting has been turned off with `GetThreadPriorityBoost`.

```
BOOL WINAPI SetThreadPriorityBoost(
    HANDLE hThread,
    BOOL DisablePriorityBoost
);
BOOL WINAPI GetThreadPriorityBoost(
    HANDLE hThread,
    PBOOL pDisablePriorityBoost
);
```

The return values indicate whether the function has succeeded (TRUE) or failed (FALSE). `GetThreadPriorityBoost` returns the current value in the `pDisablePriorityBoost` argument. A value of TRUE means dynamic boosting is enabled, while FALSE means it has been disabled. It is not

possible to turn off quantum boosting, nor is it possible to turn off the priority boosts that are applied by the Windows balance set manager or to foreground threads when waits are satisfied. It only applies to event, semaphore, I/O, and GUI thread boosts.

Multimedia Scheduler

As of Windows Vista, a new multimedia thread scheduler has been added to the system, called the multimedia class scheduler service (MMCSS). This is not really a thread scheduler per se, it's simply a service running in svchost.exe at a very high priority that monitors the activity of multimedia programs that have been registered with the system. It cooperates with them to boost priorities to ensure smoother multimedia playback. The service boosts threads inside of a multimedia program into the real-time range while it is actively playing media, but throttles this boosting periodically to avoid starving other processes on the system.

Windows Media Player 11 automatically registers itself, but any third party programs can also register programs with MMCSS. Programs do so by adding an entry to the `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks` registry key. A complete description of each of the settings is outside of the scope of this book. Please refer to MSDN and Further Reading, Russinovich, 2007, for additional details.

Sleeping and Yielding

It is sometimes necessary for a program to remove the current thread from the purview of the Windows thread scheduler for a certain period of time. There are three APIs that can be used to do this in Win32: `Sleep`, `SleepEx`, and `SwitchToThread`.

```
VOID WINAPI Sleep(DWORD dwMilliseconds);
DWORD WINAPI SleepEx(DWORD dwMilliseconds, BOOL bAlertable);
BOOL WINAPI SwitchToThread();
```

There is one such API in managed code, the static method `Thread.Sleep`, which offers two overloads to accommodate specifying the duration as either an `int` or a `TimeSpan`.

```
public static void Sleep(int32 millisecondsTimeout);
public static void Sleep(TimeSpan timeout);
```

Sleeping via the Win32 `Sleep` or `SleepEx` API or the .NET `Thread.Sleep` method will conditionally remove the calling thread from the current processor and possibly remove it from the scheduler's runnable queue. If the value of the duration argument is `0`, then Windows will only remove the current thread from the processor if there is another thread ready to run with an equal or higher priority. If there are runnable threads at a lower priority, the calling thread will continue running instead of yielding to the other threads.

Passing a value greater than `0` for the argument unconditionally results in a context switch: the calling thread removed it from the scheduler's runnable queue for approximately the duration specified. I say "approximately" because the resolution of the system clock determines how close to the milliseconds timeout the thread will sleep. As an example, if the system clock is only 10 milliseconds, as is fairly common on many machines, then specifying anything less than 10 is effectively rounded up to 10 milliseconds. It is possible to adjust the timer granularity with the `timeBeginPeriod` and `timeEndPeriod` APIs, but doing so can adversely affect the performance and power usage of your system.

Passing `TRUE` as `bAlertable` to the `SleepEx` routine specifies whether you wish to allow asynchronous procedure calls (APCs) to dispatch, if any are in the thread's APC dispatch queue waiting to run. APCs are discussed in Chapter 5, Windows Kernel Synchronization, so we will defer additional discussion of this API until then. The meaning of alertability here is identical to the meaning of alertability when waiting on kernel objects.

The Win32 `SwitchToThread` API is usually what you want to use in cases where you'd normally call `Sleep` with a value of `0` for its timeout argument. It will always yield the current processor for a single timeslice to another thread, if one is ready to run, regardless of priority. If there are no other runnable threads, then the calling thread stays running on the processor. We'll see cases in Chapter 14, Performance and Scalability, where using `Sleep` instead of `SwitchToThread` can lead to starvation and severe performance issues when writing low-level synchronization code that employs spin waiting.

Suspension

Windows offers the capability to suspend a thread's execution for an arbitrary length of time. When a thread has been suspended, the OS places

it into a suspended state and it is not eligible for execution until it has been resumed. When a thread becomes suspended, it conceptually works as though that thread's timeslice expires, resulting in the thread to be context switched off of the current processor. And when the thread is resumed, it's very much as though the thread has awakened from an OS wait, that is, it is placed into the runnable queue and will be subsequently scheduled to run on a processor.

Both Win32 and the .NET Framework have APIs to do this. Also, recall from earlier that the `CreateThread` API supports the `CREATE_SUSPENDED` flag, which ensures a thread starts life off in the suspended state and must be resumed explicitly before it runs. The Win32 APIs to suspend and resume are `SuspendThread` and `ResumeThread`:

```
DWORD WINAPI SuspendThread(HANDLE hThread);  
DWORD WINAPI ResumeThread(HANDLE hThread);
```

Each function takes a thread `HANDLE` and returns a `DWORD` that represents the suspension count prior to the call. Threads use a counter to handle cases where more than one call to suspend the same thread has been made. When the counter is above 0, the thread is suspended, and when it reaches 0, the thread is resumed again. A return value of -1 indicates error, and the details of the failure can be retrieved with `GetLastError`.

Managed code offers equivalents to these APIs as instance methods on the `Thread` class.

```
public void Suspend();  
public void Resume();
```

These don't return a recursion counter like the native APIs, although they use the Windows APIs internally and therefore also properly support recursive calls.

Suspension can be very dangerous to use in your programs. Unless the thread issuing the suspension knows precisely what the target thread is doing, the target thread may be in the middle of executing arbitrary critical regions of code. If thread A suspends B while B holds lock M and then A subsequently tries to acquire lock M, it will not be permitted to do so. And thread A may subsequently end up blocking indefinitely unless it knows to resume B and wait for it to release M before reattempting the

suspension. This is usually impossible except for very constrained circumstances. This danger is why the suspension APIs in managed code have been marked as “obsolete” in the .NET Framework 2.0, so that you will receive compiler warning messages when you use them. Also, if a thread is suspended and never resumed, that thread and its resources will stay around until the process exits.

One of the biggest misuses of thread suspension is to use it for synchronization. This is never appropriate. We’ll review appropriate synchronization mechanisms that must be used instead in the next two chapters.

There are of course cases in which suspension is useful. We saw earlier that to capture a stack trace programmatically in managed code, the target thread must be suspended for a period of time. The CLR’s GC also uses thread suspension when it needs to walk stacks to find live references on the stack. Thread suspension is frequently used in debuggers and profilers. For example, WinDbg and Visual Studio offer a “freeze threads” feature that uses thread suspension liberally. All of these share something in common. They do not invoke arbitrary program code while a thread is suspended; instead, usually a thread will be suspended for a very brief period of time, information is gathered, and then the thread is resumed. In other words, the scope of the suspension is fixed, well known, and short in duration.

Affinity: Preference for Running on a Particular CPU

The Windows thread scheduler uses many factors when determining how to schedule threads on a multiprocessor system. Each process or individual threads may be optionally confined to a subset of the CPU’s using “hard” CPU affinity. This guarantees that the scheduler will only run a given thread on a certain subset of the machine’s processors.

Each thread also has something called an ideal processor. When a processor is free and multiple runnable threads are available, the scheduler will prefer to pick one with an ideal processor of the one under consideration. But if this condition cannot be met, the OS will schedule a thread that has a different ideal processor. Similarly, Windows tracks the last processor on which a thread ran previously. Given a set of threads with a different ideal processor than the one being considered, Windows will prefer to pick

one that most recently ran on the processor. Considering the ideal and last processor improves memory locality and helps to evenly distribute the workload across the machine.

Let's now review how your programs can control hard affinity and ideal processor settings, including how to use them in your programs.

CPU Affinity

Normally a process's threads are eligible for execution on any of the available processors. Windows is free to select the processor on which a thread will run at any given time based on its own internal scheduling algorithms, preferring to fully utilize all processors over keeping a thread running on the same processor over a period of time. We've noted already that the scheduler tracks an ideal processor and the last processor on which the thread ran, and prefers to run it on one of those each time the thread must run. But if the ideal processor is busy, Windows will throw out this preference and search for a new, available processor. This kind of thread migration can incur runtime costs, primarily due to cache effects: the new thread that displaces it will likely have to incur a large number of cache misses to bring its data and instructions into the processor cache and similarly for the thread migrating elsewhere.

Processes and threads can be explicitly assigned a CPU affinity, which guarantees Windows will only schedule threads on a certain subset of the processors. This avoids migration entirely. For some specialized cases, affinity can be useful, but it often prevents the thread scheduler from performing its job. There are other strange issues that using affinity can bring about. If it happens that many threads are affinitized to the same processor (perhaps inside multiple processes), for example, the entire system performance can degrade because a number of threads are clumped together on a subset of the processors while the others remain idle. Therefore, everything mentioned in this section should be used with great care.

Some software vendors (that will remain unnamed) have shipped software with the process affinitized to CPU 0 or have asked that customers running on multi-CPU boxes use affinity to work around concurrency bugs in their software. This was more popular when Windows first began

running on SMPs and has mostly gone by the wayside as parallel architectures have become more and more common. Nevertheless, I hope your reaction to this practice is the same as mine (not positive). Using CPU affinity to achieve functional correctness is most likely an indication of more serious problems with your software.

Affinity assigned to a process is inherited by all of that process's threads, while affinity assigned directly to a thread is specific to that thread. (Process affinities are also inherited by other processes created by that process.) A thread's affinity can be more restrictive than its process's, but not less. For example, if the process is affinitized to processors 0, 1, and 3, then a single thread in the process cannot be affinitized to just processor 2 because processor 2 doesn't appear in its corresponding process's affinity. But any combination of processors 0, 1, and/or 3 is certainly acceptable.

Affinities take the form of bit-masks in which each bit corresponds to one processor (the least significant bit corresponding to processor 0): a 0 value for any given bit indicates that the process or thread cannot run on the given processor, while a 1 bit means that it can. The affinity mask is a pointer size value, meaning 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. There is also a so-called system affinity mask that is a mask containing 1 bits for all of the processors available to the system: this mask is system-wide, and much like the way in which thread masks must be subsets of the process mask, process affinities (and by inference thread affinities) may only assume values that are subsets of the system mask.

(Here's a bit of trivia: one of the surprisingly few reasons that Windows cannot currently support more than 32 CPUs on 32-bit machines and 64 CPUs on 64-bit machines is due to the size of affinity mask. Yes it's surprising, and yes it's true.)

Let's take an example: say you're running on a 32-bit 8-CPU machine and all processors are available to the system. The system mask will be the hexadecimal value `0x000000ff`, or, in 32 bits, `0000 0000 0000 0000 0000 0000 1111 1111`. Notice that lesser significant bits map to lower processor numbers; in this case, the bits read from right-to-left. (To save space we will omit writing out the 0s when all of the more significant bits are 0s.) If we wanted to confine all threads in a process to run on, say, the

4 even-numbered CPUs (i.e., 0, 2, 4, 6), we could set the process mask to `0x55`, or `0101 0101`. Notice the positions of the bits turned on correspond directly to the processors mentioned. All threads in the process would subsequently run only on those 4 specific processors. We could go further and set two individual threads' masks so that they won't share processors, say, to 2 CPUs apiece: `0x50` and `0x05`, respectively, or `0101 0000` and `0000 0101`. One of these threads will only use CPUs 0 and 2, while the other will be restricted to CPUs 4 and 6.

Assigning Affinity. There are four ways in which you can assign affinity.

First, you can store a process affinity mask inside an executable's PE file image header. None of the Windows SDK compilers or tools makes this very easy. Instead, you will need to edit the PE file with an editor. The IMAGECFG.EXE tool will do the trick. It used to be included in the Windows SDK, but now it's a little bit more difficult to find. With this tool, however, we could assign the process affinity `0x55` mentioned earlier to some fictional executable FOO.EXE via the command '`IMAGECFG.EXE FOO.EXE -a 0x55`'. You can also force the EXE to run only on a single CPU with the switch '`IMAGECFG.EXE FOO.EXE -u`', which is really just a shortcut for the option '`... -a 0x1`'.

Second, Win32 provides the APIs `GetProcessAffinityMask` and `SetProcessAffinityMask` functions to programmatically retrieve and set the affinity mask for the current process. The `GetProcessAffinityMask` also gives you access to the system affinity mask by setting the value behind the `lpSystemAffinityMask` pointer.

```
BOOL WINAPI GetProcessAffinityMask(
    HANDLE hProcess,
    PDWORD_PTR lpProcessAffinityMask,
    PDWORD_PTR lpSystemAffinityMask
);
BOOL WINAPI SetProcessAffinityMask(
    HANDLE hProcess,
    DWORD_PTR dwProcessAffinityMask
);
```

Here is an example of using these APIs to restrict the current process to CPUs 0, 2, 4, and 6.

```

HANDLE hProcess = GetCurrentProcess();
SetProcessAffinityMask(hProcess, static_cast<DWORD_PTR>(0x55));

DWORD_PTR pdwProcessMask, pdwSystemMask;
GetProcessAffinityMask(hProcess, &pdwProcessMask, &pdwSystemMask);

printf("processmask=%x, sysmask=%x\r\n", pdwProcessMask, pdwSystemMask);

```

Assuming we run this program on an 8-CPU machine, the output will be "processmask=0x55, sysmask=0xff". Trying to set a mask that isn't a strict subset of the system mask will fail, causing the `SetProcessAffinityMask` API to return FALSE.

The third way to assign affinity is to set a specific thread's CPU affinity with `SetThreadAffinityMask` instead of setting it process-wide:

```

DWORD_PTR WINAPI SetThreadAffinityMask(
    HANDLE hThread,
    DWORD_PTR dwProcessAffinityMask
);

```

Unlike process affinity, there isn't an easy API with which to retrieve the current affinity mask for a thread. This can be obtained from `SetThreadAffinityMask`: the return value is the old value for the mask. There is no way to retrieve the current mask without also modifying it. Attempting to specify an affinity mask that isn't a strict subset of the process affinity mask (and by inference the system mask) will fail, conveyed with a return value of 0.

Continuing to build on our earlier example, say we had two thread handles, `h1` and `h2`, referring to the two threads we want to affinitize to CPUs 0 and 2, and 4 and 6, respectively:

```

DWORD_PTR h1PrevAffinity = SetThreadAffinityMask(
    h1, static_cast<DWORD_PTR>(0x50));

DWORD_PTR h2PrevAffinity = SetThreadAffinityMask(
    h2, static_cast<DWORD_PTR>(0x05));

printf("h1prev=%x, h2prev=%x\r\n", h1PrevAffinity, h2PrevAffinity);

```

If we ran this on the same 8-CPU machine after affinitizing the whole process, the value printed to standard output would be "h1prev=0x55, h2prev=0x55".

The fourth and final way to assign affinity is to use a tool that programmatically sets the affinity. As you saw above, the `SetProcessAffinityMask` function takes any process `HANDLE` as its first argument. That handle needn't refer to the current process. Tools can use this to enable a process's affinity to be set after it has been started. Two Windows built-in tools allow you to do this and are worth mentioning:

- The `START` command allows you to pass the affinity mask as a command line argument, with the `/AFFINITY` switch. For example, to affinize a program `PROGRAM.EXE` to CPUs 0, 2, 4, and 6 we could run '`START /AFFINITY 0x55 PROGRAM.EXE`'. This utility makes it very easy to test or rerun your program with various kinds of affinity settings, which can help tremendously with debugging multithreaded related issues.
- As of Windows Server 2003, the Windows Task Manager permits you to set affinity for an existing process: go to the Processes tab, right click on the process you'd like to affinize (or unaffinize), and select the Set Affinity option. A list of check boxes, one for each processor, will be displayed. You can select or deselect as many as you'd like, which has the effect of changing the target process's current CPU affinity as it is running.

You can also set the process's CPU affinity with the `System.Diagnostics.Process` class's `ProcessorAffinity` property in the .NET Framework. Managed threads do not expose thread CPU affinity directly, but you could P/Invoke to the aforementioned Win32 APIs. (This is discouraged, however, due to possible unexpected interactions with services like the GC.) The `System.Diagnostics.ProcessThread`'s `ProcessorAffinity` allows you to set affinity in .NET, which just does the P/Invoke to `SetThreadAffinityMask` for you. The `ProcessThread` class does not, however, make it easy to retrieve a `HANDLE` to the current thread; if you need to affinize the calling thread, you'd need to P/Invoke on your own or manufacture a pseudo-`HANDLE` by hand. Be careful if you decide to do such things. You wouldn't want to forget to remove affinity before returning a thread back to the CLR thread pool, and you most certainly wouldn't want to leave affinity on the

finalizer thread, for example; the results could be very unpleasant in both cases and could affect the stability of the system.

Round Robin Affinitization. Sometimes a program will need to create the same number of threads as there are CPUs on the machine and then assign each to a separate CPU. This comes up in certain classes of data parallel algorithms of the kind we'll see in later chapters, in addition to more general systems that control the scheduling of threads. An initial approach might look something like this.

```
// Get the # of threads.
SYSTEM_INFO sysInfo;
GetSystemInfo(&sysInfo);

// Now spawn our threads and affinitize them.
HANDLE * pThreads = new HANDLE[sysInfo.dwNumberOfProcessors];
for (int i = 0; i < sysInfo.dwNumberOfProcessors; i++)
{
    pThreads[i] = CreateThread(...);
    SetThreadAffinityMask(pThreads[i], (1<<i));
}
...
```

There are a few problems with this code that might not be evident right away. First, it should now be evident that while `sysInfo.dwNumberOfProcessors` returns the count of processors on the machine this may not necessarily mean that the current process can run on all of them. The process may have had its CPU affinity set. So we will need to create only as many threads as we have 1 bits in the process's affinity mask.

Assuming we need to create an array of the correct size, we'd have to make two passes over the mask. One to count the 1 bits so we can size the array correctly, and then another to actually affinitize the threads we create. Note that we have to use the same mask for both passes since somebody could change the process-wide mask asynchronously as we are calculating them.

```
VOID GetAvailableProcessorsFromMask(
    DWORD_PTR * cdwProcs, DWORD_PTR ** ppdwPmasks)
{
    DWORD_PTR pdwProcMask, pdwSysMask;
    GetProcessAffinityMask(
        GetCurrentProcess(), &pdwProcMask, &pdwSysMask);
```

```
// First, count the processors.  
DWORD_PTR dwCount = 0;  
DWORD_PTR mask = pdwProcMask;  
while (mask > 0)  
{  
    if ((mask & 1) dwCount++;  
    mask >>= 1;  
}  
  
// Next, generate the masks.  
DWORD_PTR * dwMasks = new DWORD_PTR[dwCount];  
DWORD_PTR i = 0, j = 1;  
while (i < dwCount)  
{  
    while ((pdwProcMask & j) == 0)  
        j <= 1;  
    dwMasks[i] = j;  
    i++;  
    j <= 1;  
}  
  
*cdwProcs = dwCount;  
*ppdwP Masks = dwMasks;  
}  
  
...  
  
// Now spawn our threads and affinitize them.  
DWORD_PTR count;  
DWORD_PTR * masks;  
GetAvailableProcessorsFromMask(&count, &masks);  
HANDLE * pThreads = new HANDLE[count];  
for (int i = 0; i < count; i++)  
{  
    pThreads[i] = CreateThread(...);  
    SetThreadAffinityMask(pThreads[i], masks[i]);  
}  
delete [] masks;  
  
...
```

This information may be out of date as soon as it has been calculated, so it's still not foolproof. But it is better than not accounting for affinity at all. The naïve approach we began with may be appropriate for some systems, but if you expect processor affinity to be set with any regularity (particularly if your own code does it), then you should take it into consideration.

There's still another rather obscure issue remaining with this code. On a 64-bit system, the count of CPUs may be anywhere from 1 to 64. But if you are running a 32-bit process within WOW64, for example, then affinity masks will only be 32-bits wide. This could cause subtle program bugs if you ever make an assumption about the number of bits available in a mask directly correlating to the number of processors the OS claims are available. APIs that interact with processor affinities simulate greater than 32 processors in a WOW64 program by silently changing the bitmasks. Upon retrieval, the high and low 32 bits are combined using a bitwise OR, hence a mask of `0x1` could indicate either processor 1 or 32. A program in WOW64 that sets the thread affinity will restrict it to running on the first 32 processors.

Microprocessor Architecture Considerations. There are two particular microprocessor architectures in which affinity can be of particular interest. Affinity can be used to ensure threads run only on one of the logical processors when running on an Intel HyperThreading (HT) processor. Because each logical processor on a single HT chip shares a set of execution units, having many compute-intensive and low-memory-latency threads share a single HT chip can be inefficient. Not only does throughput drop, but scheduling the work can increase memory latency induced waits. (For instance, this might happen if a thread is able to normally keep all of its data in cache, but by scheduling multiple threads on the same HT chip, the total working memory needed by both cannot fit.) If we had two HT chips with two cores and two logical processors each (that's an 8-way), and four threads to run, we might choose to affinitize those threads to run only on processors 0, 2, 4, and 6 because the adjacent pairs (i.e., 0 and 1, 2 and 3, etc.) constitute the HT logical processors.

The second microprocessor architecture where affinity can be useful is Non-Uniform Memory Access (NUMA) machines. In a NUMA machine, there are separate nodes, where a node is some number of CPUs and a separate memory system. Memory transfer between nodes is very expensive—even more than an ordinary cache miss that has to hit main memory—and so it's generally best if a thread is run on a processor in the same node as the

memory it will frequently access. Windows is NUMA aware and will ensure memory allocated by a thread happens in the node on which the thread is actively running. But a thread may migrate, in which case some portion of its memory accesses will be cross node. Using affinity to tie a thread to a certain NUMA node can help to eliminate costly asymmetric memory accesses due to thread migration.

Ideal Processor

When a thread is created on multiprocessor systems, the OS auto-assigns it an ideal processor. The determination of ideal processor is fairly arbitrary: the OS uses a per process round robin algorithm to dole out ideal processors as they are needed. Each process is given a seed, and then anytime a thread is created within that process, the seed is incremented. Process seeds are also given out in a round robin fashion. The choice of ideal processor is also hyperthreading aware and attempts to utilize all physical processors before resorting to individual logical processors. This algorithm is meant to somewhat evenly distribute ideal processors among the threads created in the system and is apt to change at any time.

An ideal processor is the thread's preferred processor, and it remains constant throughout the life of that thread unless changed manually. The OS thread scheduler uses it during the algorithm which determines which thread to run next on a processor during context switches. Having an ideal processor increases the probability that a thread will run more frequently on one particular processor, which consequently means that the thread has a better chance of finding data it used previously in the processor's cache.

There is a Win32 API to retrieve or set the current thread's ideal processor. This can be used for situations in which hard affinity is too strong, but when some higher-level component knows that having a thread run regularly on a particular processor will lead to better performance.

```
DWORD WINAPI SetThreadIdealProcessor(
    HANDLE hThread,
    DWORD dwIdealProcessor
);
```

This API accepts a HANDLE to the thread whose ideal processor is to be accessed and a DWORD representing the new ideal processor for that thread. (Note that this value is not a bitmask as is used by some other Win32 APIs to represent processors; it's an actual integer value representing the processor number.) The function returns the old ideal processor number. If you want to obtain the current value for a thread's ideal processor without changing it, you may specify MAXIMUM_PROCESSORS for dwIdealProcessor, which causes it to return the current setting. This function can fail, in which case the return value is -1; this can happen, for example, if you specify an invalid processor.

Where Are We?

This concludes our two chapter overview of Windows and CLR threads. In this chapter, we looked very deeply at what thread stacks are comprised, their specific layout, and some interesting policy around how their memory is managed by the OS and CLR, such as stack growth and stack overflow. We also looked at TEBs and thread contexts. Various aspects of thread scheduling were also explored, including how the OS makes its scheduling decisions and how you can influence them with priorities, ideal processor settings, and affinity.

We will now turn our attention to some other kernel services that support concurrent programming: a set of rich kernel objects that can be used to synchronize among threads.

FURTHER READING

Windows XP Embedded Team. Master Your Quantum. Weblog article, <http://blogs.msdn.com/embedded/archive/2006/03/04/543141.aspx> (2006).

M. Pietrek. Under the Hood. *Microsoft Systems Journal*, <http://www.microsoft.com/msj/archive/S2CE.aspx> (1996).

M. Pietrek. Under the Hood. *Microsoft Systems Journal*, <http://www.microsoft.com/msj/0298/hood0298.aspx> (1998).

M. Russinovich, D. A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000*, Fourth Edition (MS Press, 2004).

M. Russinovich. Inside the Windows Vista Kernel: Part 1. *TechNet Magazine*,
<http://www.microsoft.com/technet/technetmag/issues/2007/02/VistaKernel>
(2007).

5

Windows Kernel Synchronization

In CHAPTER 2, Synchronization and Time, we discussed some of the basics of synchronization. This included the circumstances in which it's necessary to synchronize and some of the associated pitfalls. In this chapter, we'll look closely at the most fundamental support for synchronization offered by the Windows OS: **kernel objects**. These objects serve as the basic building blocks for all concurrent programs and primitive data structures. In fact, whether or not you use these objects directly in your code, you will almost always rely on them at some layer of software. Just about all synchronization primitives available in Win32 and the .NET Framework, including Win32 critical sections and CLR monitors (see Chapter 6, Data and Control Synchronization), for example, use them in one way or another. For this reason, we'll examine the details of them before looking at higher level data and control synchronization mechanisms in the next chapter.

Windows offers several different kinds of kernel objects. Some kinds offer more sophisticated functionality in addition to being useful for synchronization purposes—such as the thread kernel object representing an OS thread as reviewed in the past two chapters, file notification objects, and more—but we'll focus on synchronization behavior in this chapter.

Five object types are synchronization specific and, thus, of specific interest to us: **mutexes**, **semaphores**, **auto-reset events** (a.k.a. **synchronization events**), **manual-reset events** (a.k.a. **notification events**), and **waitable timers**. Each kernel object kind generally has its own Win32 API(s) and .NET Framework classes for object creation, management, and deletion. The kernel itself manages the memory and resources associated with each object, and user-mode code only manipulates such objects via these controlled APIs. Once an object has been created, it is subsequently referred to by user-mode code with its **HANDLE** in Win32 programming (which is a pointer sized opaque value). Handles to objects are reference counted, so multiple outstanding references will keep an object from being de-allocated. When objects are no longer in use, handles to them must be closed with the Win32 **CloseHandle** API.

The .NET Framework offers support for four of the five classes via instances of subclasses of the `System.Threading.WaitHandle` abstract base class. (The fifth class, **waitable timers**, is supported and exposed indirectly through the thread pool.) Kernel object classes in .NET offer a `Close` or `Dispose` method to close the underlying handle, and each such object is protected by a finalizer to ensure that kernel objects that haven't been explicitly closed don't result in permanent process-wide resource leakage.

The content of this chapter assumes that readers have a general familiarity with basic Windows topics like handles, handle lifetime, and the process handle table, named objects, object security, and so on. Several resources (see Further Reading, Petzold; Richter; Russinovich, Solomon) listed in the references at the end of this chapter cover these topics extensively. And although a lot of this chapter may seem Win32 specific—which could seem unimportant if you are writing all your code on the CLR—you'll find all of the information in this chapter useful and applicable to all Windows programming, regardless of the language or APIs used.

The Basics: Signaling and Waiting

The basic way synchronization happens via kernel objects is by **signaling** and **waiting**.

Each kernel object instance can be in one of two states at a given time: **signaled** or **nonsignaled**. The exact rules governing how an object

transitions between these two states are defined by the specific type of kernel object in question and vary a great deal. This difference is what makes each object special, allowing different sorts of objects to be used for different purposes.

But what does signaled versus nonsignaled mean to you as a Windows programmer? Chapter 2, Synchronization and Time, mentioned that spin waiting is usually an inefficient way to wait for events of interest to occur and that the OS intrinsically supports true waiting. We also saw in the chapters on threads that a thread can *block* for a variety of reasons: I/O, sleeping, and suspension, to name a few. Another useful way a thread can block is by waiting for a Windows executive kernel object to become signaled.

Once a thread has a reference to a kernel object, it can easily wait on with a Win32 or .NET wait API: it: if the object isn't signaled already, this results in a context switch. The thread is removed from the current processor, and is marked so that the OS thread scheduler knows it is currently ineligible for execution. As soon as the object later becomes signaled, the waiting thread is marked as runnable, which causes the kernel to place it back into the thread scheduler's queue of runnable threads. Eventually the thread will be chosen to run again on a processor based on the scheduler's standard scheduling algorithms.

Many threads can wait simultaneously for the same kernel object to become signaled. For certain kernel objects, only a fixed number of waiting threads will be awakened when it becomes signaled. In some cases, like mutexes and auto-reset events, that number will be one. Semaphores, on the other hand, have a count and will wake up a number of threads up to the current count value. If the count is three and five threads are waiting, only three will be awakened and the other two will remain blocked. Yet in other cases, such as manual-reset events, all waiting threads are awakened at once. When a fixed number of threads must be awakened, the OS uses a semi-fair algorithm to choose between them: as threads wait they are placed into a FIFO queue that the awakening logic consults when determining which thread to wake up. Threads that have been waiting for the longest are thus preferred over threads that have been waiting for less time. Although the OS does use a strict FIFO data structure to manage wait lists, we will see later that this ordering is regularly perturbed by other system code and is not reliable.

When a thread wants to wait for an object to become signaled, there are a number of Win32 APIs that can be used: `WaitForSingleObject`, `WaitForSingleObjectEx`, `WaitForMultipleObjects`, or `WaitForMultipleObjectsEx`. There are other alternative variants of these APIs, prefixed with `Msg`, that are used in GUI and COM programs so a thread can continue to process messages while it waits. COM also exposes a special `CoWaitForMultipleHandles` API that is frequently used by COM programs because it encapsulates some tricky message handling code to dispatch COM RPC calls. In managed code, you'll use the instance method `WaitHandle.WaitOne` on the managed object representing the kernel object, or the static methods `WaitAll` or `WaitAny`. These internally take care of COM and GUI message pumping, as needed. We'll discuss the exact differences and why you'd select one over the other in upcoming sections.

We'll review many of the kernel objects in detail throughout this chapter, but first, Table 5.1 depicts a summary of how the different types transition between states.

As Table 5.1 depicts, the transitions between the signaled and nonsignaled state vary between different object kinds. Some objects are modified as a result of a thread waiting on the signaled object. Mutexes, for example, become "owned" by the calling thread and transition immediately back to the nonsignaled state (atomically); a semaphore's count is decremented by one, possibly transitioning back to nonsignaled if this count reaches 0; and auto-reset events unconditionally transition back to the nonsignaled state, always. These effects actually enable powerful synchronization capabilities. Additional effects also are possible: waking from a wait on an event or semaphore object temporarily boosts the waking thread's priority to increase the probability that the waking thread will run again sooner rather than later, for instance, often leading to quicker rescheduling.

Why Use Kernel Objects?

As we'll review in the next chapter on data and control synchronization, there are many libraries available on the platform meant for synchronizing between threads. We're jumping ahead of ourselves a little, but you've heard of critical sections, condition variables, monitors, reader/writer locks, and the like. Using kernel objects directly is usually more expensive

TABLE 5.1: Kernel object types and state transitions

Object Type	Nonsignaled	Signaled
Console Input	The console input buffer is empty	There is unprocessed data in the console input buffer
Event (Auto-Reset)	Automatically when a thread waits on a signaled event	Set manually with the SetEvent API
Event (Manual Reset)	Reset manually with the ResetEvent API	Set manually with the SetEvent API
File, Directory, Named Pipe, or Communication Device	No outstanding asynchronous I/O packets have completed	Outstanding asynchronous I/O packets have completed and must be processed
File Change Notification	The file notification condition has not yet been met (see FindFirstChangeNotification)	A file change of interest has been detected
Job	The job and its related processes are running	A job's processes have completed
Keyed Event	No event has been registered for the key being waited on	An event has been registered for the key being waited on
Memory Resource Notification	No low memory resource condition exists (see CreateMemoryResourceNotification)	A low memory resource condition exists
Mutex (a.k.a. Mutant)	A thread successfully waits on a mutex	A thread calls ReleaseMutex (once per corresponding wait call)
Process	The process is running	The process has exited
Semaphore	The semaphore count has reached 0	The semaphore count has gone above 0
Thread	The thread is running	The thread has terminated
Waitable Timer (Auto-Reset)	Timer hasn't expired, or automatically reset to nonsignaled when a thread waits on a signaled timer	Timer has expired
Waitable Timer (Manual-Reset)	Timer hasn't expired, or when a call to SetWaitableTimer is made to manually reset it	Timer has expired

than these other primitives for several reasons, including the costly kernel transitions incurred for each API call made on one. Because kernel objects are allocated inside kernel memory, only code running in kernel-mode can access them. The alternative user-mode abstractions typically use kernel objects to implement waiting and signaling, but they are written to avoid kernel transitions wherever possible.

So if kernel objects are generally more expensive to use, why would you ever want to use one? Aside from being the core primitives out of which everything is built and facilitating interoperability with legacy code, there are a few useful features that kernel objects provide that normally can't be accessed if you only use the user-mode synchronization mechanisms.

- Kernel objects can be used for interprocess synchronization. They can be named and later looked up and, hence, can be a great way to protect machine-wide shared state. In the case of the CLR, they also can be used for inter-AppDomain synchronization, which other synchronization mechanisms usually don't support. This feature is a double-edged sword, however: with longer state lifetime comes great reliability responsibility, particularly in the area of recovering corrupt state after a process fails.
- Kernel objects can be secured via assigning access control lists (ACLs) and by requesting certain access rights when instantiating a new or finding an existing kernel object. For programs that use standard Windows security mechanisms, this can be an attractive feature, and it is typically not supported by other user-mode abstractions.
- You have more control over and can perform more sophisticated waits when using kernel objects, such as waiting for all or one out of a collection of objects to become signaled. This can be a very powerful capability, and there is generally no substitute on the platform that provides all of the same features. Similarly, you can decide whether to issue an alertable wait (to dispatch APCs) or to pump for GUI or COM RPC messages—two features generally not supported by many other synchronization mechanisms.
- Kernel objects can be used to interoperate between native and managed code.

Simply put, kernel objects are more powerful and comprise the base of the Windows platform's architectural support for concurrency. Many situations call for using one directly, although there are plenty of (possibly cheaper) alternatives to consider. And even in cases that do not call for their use, your API of choice will undoubtedly end up using them indirectly, whether you are required to know this or not. A solid understanding of them is always useful.

Waiting in Native Code

Let's now turn to the general-purpose wait APIs, starting with the native APIs. After that, we'll see how waiting differs in the CLR. Last, we'll look at all the specific kernel objects, what makes them unique, and how they are used.

WaitForSingleObject(Ex) and WaitForMultipleObjects(Ex)

The simplest way to wait on a kernel object in Win32 is to use one of the four standard waiting APIs mentioned earlier. The first two APIs allow you to wait on a single object, while the latter two enable waiting for multiple (either any or all) to become signaled:

```
DWORD WINAPI WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);
DWORD WINAPI WaitForSingleObjectEx(
    HANDLE hHandle,
    DWORD dwMilliseconds,
    BOOL bAlertable
);
DWORD WINAPI WaitForMultipleObjects(
    DWORD nCount,
    const HANDLE * lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds
);
DWORD WINAPI WaitForMultipleObjectsEx(
    DWORD nCount,
    const HANDLE * lpHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds,
    BOOL bAlertable
);
```

The single object wait APIs, `WaitForSingleObject` and `WaitForSingleObjectEx`, take a single `HANDLE` to an instance of any of the aforementioned waitable kernel objects and a timeout, `dwTimeout`, specified in milliseconds. The value `INFINITE`, which is just a constant defined as `-1` by `Windows.h`, can be passed to indicate that no timeout is desired. A value of `0` requests that the function check the object's state and return immediately, guaranteeing that if the object is nonsignaled, no blocking will occur. In other words, the function will not directly cause a context switch.

When the call to either function returns, the return value must be checked: a value of `WAIT_OBJECT_0 (0L)` means that the wait was successful and that the object had become signaled. If the specific type of kernel object's state can be changed by waiting, such as with a mutex, semaphore, or auto-reset event, these changes will have occurred by the time the function returns. A return value of `WAIT_TIMEOUT (258L)` means that the timeout expired before the object became signaled. The return value `WAIT_FAILED (0xffffffff)` represents an error, such as an invalid `HANDLE`, inability to allocate system resources to perform the wait, and so forth. `GetLastError` can then be called to retrieve additional details. A fourth possible return value, `WAIT_ABANDONED (128L)` will be described later when we discuss mutexes in depth; it only applies to waiting on mutex objects and indicates that the mutex was not properly released by some previously executed piece of code. Despite appearing to be an error, the wait is successful (i.e., the mutex is owned).

The multiple object variety of the wait APIs, `WaitForMultipleObjects` and `WaitForMultipleObjectsEx` effectively do the same thing as the single-object functions, with the only difference being that they can be used to wait for more than one kernel object at the same time. The `HANDLE`s to wait on are passed in the `lphandles` array, and the `nCount` argument represents the number of objects in the array.

The maximum number of handles you can wait on at once is 64, as specified by the `MAXIMUM_WAIT_OBJECTS` constant in `WinNT.h`. If you supply an argument of greater size, everything from the sixty-fourth element onward will be ignored. This limitation can sometimes be tricky to work around if the number of events you wait on varies dynamically. If this is a problem for you, please refer to Chapter 7, Thread Pools, where we look into a

feature supported by both the native and managed thread pools to register an arbitrary number of waits.

The `bWaitAll` argument specifies whether wait-all (TRUE) or wait-any (FALSE) behavior is desired. If you'd like to wait until all of the handles have become signaled, then you'll want to use a *wait-all* style wait (TRUE). If you instead want the wait to return as soon as any single one of the handles becomes signaled, then you want the default of *wait-any* (FALSE).

For wait-all style waits, the return values are similar to the single object APIs: `WAIT_OBJECT_0` indicates that all handles are signaled, `WAIT_TIMEOUT` indicates that the timeout expired, and `WAIT_FAILED` indicates a problem occurred. The only difference in return values for wait-all is the way in which abandoned mutexes are communicated, because we need to know not just that a mutex was abandoned, but which specific object it was. Similarly, for wait-any waits, we need to know the index of the `HANDLE` in the array for the object that became signaled and caused the function to return. Both cases are treated similarly.

For these cases, the element's array index is encoded in the return value itself. In the case of a wait-any, the return value will be `WAIT_OBJECT_0 + i`, where `i` is the signaled element's index in the `HANDLE` array and is within the range of `WAIT_OBJECT_0` to `WAIT_OBJECT_0 + nCount - 1`, inclusive. Remember that `WAIT_OBJECT_0` is just the value `0`, so you can directly use the return value to index into the array without any translation (though it's theoretically better to subtract `WAIT_OBJECT_0` in case the value changes in the future). If at least one of the handles was a mutex and it was found to be abandoned, the return value will instead be `WAIT_ABANDONED_0 + i`, where `i` is the abandoned mutex's index in the `HANDLE` array. To calculate the mutex's array index, simply subtract `WAIT_ABANDONED_0`, which is the same value as `WAIT_ABANDONED`. If there are multiple abandoned mutexes in the wait list, only the first (index-wise) will be communicated. An abandoned mutex does not imply failure: the wait will have been fully satisfied when you see a `WAIT_ABANDONED_0` value, that is, for a wait-all every other object is also signaled.

Wait-all is implemented efficiently in the Windows kernel, ensuring that a thread remains blocked even when only some of the many objects the thread is waiting for becomes signaled. A naïve implementation of wait-all would

be to loop over the objects and wait on each individually. But this has drawbacks. The performance drawback is obvious: there likely will be a context switch for every single object, as it becomes signaled. The functionality drawback is more subtle: if any of the objects' states are changed by waiting on them—as with mutexes, semaphores, and auto-reset events—the Windows implementation ensures these changes only occur once *all* the objects have become signaled, not one by one. This ensures that if a thread fails after some objects are signaled, but not others, there will be no state corruption.

Due to this, the FIFO ordering noted earlier is not strictly preserved for threads doing a wait-all. If thread t1 does a wait-all on objects A and B, and then A gets signaled, t1 must still wait for B to become signaled before waking up. In the meantime, some other thread t2 is still free to wait on A. Instead of holding up t2's wait indefinitely while t1 waits for B to also become signaled, Windows will let t2's wait on A succeed ahead of t1's. If that resets A's signal, t1 will then have to wait for A to become signaled again. This behavior also avoids deadlock: say t1 waited on objects A and B, in that order, and t2 waited on the same objects in the reverse order, B and then A, the naïve one-at-a-time approach would lead to deadlock.

This C++ code sample shows a wait-any style wait with boilerplate code that handles the various return values including translating them into an array index.

```
const int cHandles = ...;
HANDLE waitHandles[cHandles];
// ... populate our array with HANDLES ...

// Do the wait (possibly blocking the thread):
DWORD dwWaitRet = WaitForMultipleObjects(
    cHandles, &waitHandles[0], FALSE, INFINITE);
if (dwWaitRet >= WAIT_OBJECT_0 &&
    dwWaitRet < WAIT_OBJECT_0 + cHandles)
{
    HANDLE hSignaled = waitHandles[dwWaitRet - WAIT_OBJECT_0];
    // hSignaled is a handle to the object that became signaled...
}
else if (dwWaitRet >= WAIT_ABANDONED_0 &&
         dwWaitRet < WAIT_ABANDONED_0 + cHandles)
{
    HANDLE hAbandoned = waitHandles[dwWaitRet - WAIT_ABANDONED_0];
    // hAbandoned is a handle to the mutex that was abandoned...
}
```

```
else if (dwWaitRet == WAIT_TIMEOUT)
{
    // Handle timeout...
}
else if (dwWaitRet == WAIT_FAILED)
{
    DWORD dwError = GetLastError();
    // Handle error condition...
}
```

Alertable Waits. The `WaitForSingleObjectEx` and `WaitForMultipleObjectsEx` APIs have an extra parameter that we haven't described yet: `BOOL bAlertable`. For the non-Ex methods, this is effectively always `FALSE`. But if you pass `TRUE` explicitly and the thread blocks, it can be interrupted and wakened before the wait is satisfied by a Windows user-mode asynchronous procedure call (APC). APCs are discussed later, but in summary. An APC unblocks the thread so it can perform some interesting (but often unrelated) work instead of remaining in the wait state. They are used by some Win32 infrastructure—like marshaling the bytes read from a file into a buffer after an asynchronous `ReadFileEx` operation—without you necessarily being aware of it. If an APC interrupts the wait, the call will return even though objects haven't necessarily been signaled. In such cases, the return value will be `WAIT_IO_COMPLETION`.

In most cases, the caller should respond to a return value of `WAIT_IO_COMPLETION` by reissuing the wait. Restarting the wait is a little tricky because of timeouts: if a `dwTimeout` value other than `INFINITE` was specified, we will need to manually decrement the number of milliseconds that elapsed since the start of our previous wait. Otherwise, we'll possibly wait multiple times with the same original timeout, which would clearly be wrong (e.g., if `dwTimeout` was 1000, we could wait for 999 milliseconds, wake up due to an APC, wait again for 999 milliseconds, wake up due to an APC, and so forth). This demands some kind of time accounting, as the following code example illustrates:

```
#include <stdio.h>
#define _WIN32_WINNT 0x0400
#include <windows.h>

DWORD DoSingleWait(HANDLE h, DWORD dwMilliseconds, BOOL bAlertable)
{
    // Track the start and elapsed time.
```

```

DWORD dwStart = GetTickCount();
DWORD dwElapsed = 0;

// We need to loop due to APCs.
DWORD dwRet = 0;
while ((dwRet = WaitForSingleObjectEx(
    h, dwMilliseconds - dwElapsed, bAlertable)) ==
    WAIT_IO_COMPLETION)
{
    if (dwMilliseconds != INFINITE)
    {
        dwElapsed = GetTickCount() - dwStart; // Add wait time.

        if (dwElapsed >= dwMilliseconds)
        {
            // We've exceeded the wait time -- timeout.
            dwRet = WAIT_TIMEOUT;
            break;
        }
    }

    // ... got an APC, reissue the wait again ...
}

return dwRet;
}

```

This demonstrates a general purpose `DoSingleWait` routine that correctly adjusts the running timeout in the face of APCs and then, assuming the timeout hasn't been exceeded yet, reissues the wait on the same object. It could be easily extended to call `WaitForMultipleObjectsEx` instead, if we needed to wait on multiple handles. (In fact, we'll see such an extension when we look at the `Msg`-variant of the wait APIs in a few sections.) To simplify things, this example does not use a high-resolution timer, which means, depending on your OS configuration, the resolution may be limited to the normal system clock timer, usually between 10 and 15 milliseconds. This is typically fine, but if you are worried about such things, you might want to look at using `QueryPerformanceFrequency` and `QueryPerformanceCounter` instead of `GetTickCount`, at some expense.

Notice that restarting waits such as the `DoSingleWait` function leads to multiple calls to `WaitForSingleObjectEx` on the same object `HANDLE`. This has one subtle implication that was hinted at earlier. Although kernel

objects track and signal waiting threads in FIFO order, the current thread is removed completely from the wait queue when an APC wakes it. Therefore, each time the wait API is subsequently called, the thread must go back to the end of the object's wait queue. The kernel object infrastructure doesn't know anything about the restarted wait, and so any threads now ahead of it in line will be preferred when selecting a thread to be awakened. This is desirable, particularly if the APC takes some time to execute, there are multiple threads waiting for an object, and it is signaled before the APC finishes. The alternative would lead to threads waiting unnecessarily. APCs therefore disrupt the strict FIFO ordering of the OS kernel objects in ways that are hard to predict and explain. For cases with extremely busy kernel objects and heavy APC usage, you might notice some degree of starvation as a result. In practice, this extreme is rare.

Message Waits: GUI and COM Message Pumping

Threads that own message queues in Windows have to pump messages. A thread acquires such responsibility whenever a thread creates a GUI window, that is, by calling USER32's `CreateWindow` or `CreateWindowEx` function that will be sent messages that need processing. Other system services will create windows on behalf of the caller, most notably COM's `CoInitialize` or `CoInitializeEx` functions. And what exactly does it mean to "pump messages" anyway?

A thread's message queue is strikingly similar to its APC queue in the sense that each message enqueued represents some amount of work that needs to occur on that thread. Various components in the Windows infrastructure place messages into the window's message queue, and it's the responsibility of the thread that owns that particular window to ensure those messages get processed. Instead of entering an alertable wait state to dispatch messages, the thread must pump messages, that is, run its message loop in order to drain its message queue.

Most window messaging is hidden underneath GUI frameworks and COM proxy infrastructure that applications use indirectly. But a lot of system code needs to deal directly with such things. And failure to pump messages can occasionally lead to real trouble, ranging from unresponsive GUI programs to deadlocked COM components.

Threads place messages into a thread's queue through a variety of mechanisms, either synchronously or asynchronously. A simple way of adding new messages is via USER32's `PostMessage`, `PostThreadMessage`, `SendMessage`, `SendMessageCallback`, and related APIs. Posting a message enqueues a message into a particular window's message queue and then returns immediately, whereas sending a message enqueues the message and then waits for the window's thread to process the message (or, alternatively, ensures a callback is invoked when the thread processes the message).

```
BOOL PostMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
BOOL PostThreadMessage(
    DWORD iThread,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
LRESULT SendMessage(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam
);
BOOL SendMessageCallback(
    HWND hWnd,
    UINT Msg,
    WPARAM wParam,
    LPARAM lParam,
    SENDASYNCPROC lpCallback,
    ULONG_PTR dwData
);
```

These are really just special forms of interthread communication and synchronization that a fair bit of Windows and COM code happens to use. Interestingly, most of the Windows GUI subsystem is built on top of the message queue. Whenever a window is resized, clicked, or closed, this is communicated via a new message in the window's queue. The thread that owns the target window will eventually retrieve the message out of its

queue and perform the GUI task being requested. For GUI messages, then, a thread that owns a GUI message queue but isn't pumping messages, can lead to an unresponsive, hung UI, for example, where user clicks simply get placed into the message queue without a timely response from the program.

COM uses message queues in strange ways to support its **apartment threading model**. Apartments are just COM isolation and synchronization boundaries, and components within one apartment may send messages to components in another apartment in order to invoke functions and pass data. This is done through message passing and is built on the same message queue infrastructure used by GUIs. This works because each apartment has a message queue (created automatically by COM as a hidden USER32 "RPC" window during `CoInitialize`). When a thread outside the particular apartment needs to access a COM object created inside the apartment, it can't do so directly. Instead, most often the call occurs via a proxy COM interface pointer, produced by a call to the `CoMarshalInterface` API, which indirectly results in a message being queued into the destination apartment's message queue.

Why does all of this matter? Well, cross-apartment proxy calls need to "get into" the target component's apartment. You may wonder how this happens. Cross-apartment calls place a message into the target apartment's message queue, and then the caller waits for the target apartment to pump messages and dispatch the call. The target apartment's pumping has the effect of invoking the cross-apartment method call and marshaling the return value back to the calling apartment, typically via another cross apartment message send.

The specific mechanisms involved are rather complicated because to prevent deadlocks the calling apartment might have to pump messages of its own as the RPC call occurs. Imagine if the call originated in some source apartment and the marshaled function call executing inside the destination apartment turned around and tried to access a component in the source apartment; if the source apartment's thread was blocked waiting for the original RPC call to return, the result would be deadlock, for instance. Failure to pump in this case is worse than an unresponsive GUI application—it can lead to deadlocks that bring the program to a halt. All

of this can become even more complicated, involving circular calls between larger sets of apartments. A thorough treatment of COM itself is well outside of the scope of this book, and the curious reader is referred to Don Box's *Essential COM* (see Further Reading) for all the detail you could possibly desire. Also refer to *Effective COM* (see Further Reading) for some STA-specific rules and guidelines when writing COM code.

MsgWaitForMultipleObjects(Ex). Let's get back to the topic at hand: how do window messages get dispatched? Unlike APCs, which you'll recall are dispatched automatically by the Windows kernel whenever a thread performs an alertable wait, message queue messages must be processed by hand. Most GUI applications have a top-level **modal loop** whose job is to process messages as they arrive, by using the standard message loop.

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

In addition to `GetMessage`, there is also a `PeekMessage`, which enables a thread to look into its message queue without actually dequeuing a message. I'm not going to go into detail here, since message loops have been around a long time and are well documented in other books (e.g., in the classic *Programming Windows*, by Charles Petzold, see Further Reading). What I am going to cover, however, is what happens when a thread with a message queue has a call stack that has left the message loop and suddenly needs to block for some reason. In such cases, we often want to pump for messages to avoid the kinds of problems described earlier. Note that often a better design is to transfer the wait to a separate thread—for example, using techniques described in Chapter 16, Graphical User Interfaces—but let's assume for the following discussion that this approach is not possible.

To handle the block and pump for messages situation, there are two wait APIs very similar to those we saw earlier: `MsgWaitForMultipleObjects` and `MsgWaitForMultipleObjectsEx`. These functions allow us to wait for a set of handles while simultaneously pumping for messages.

```
DWORD WINAPI MsgWaitForMultipleObjects(
    DWORD nCount,
    const HANDLE * pHandles,
    BOOL bWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask
);
DWORD WINAPI MsgWaitForMultipleObjectsEx(
    DWORD nCount,
    const HANDLE * pHandles,
    DWORD dwMilliseconds,
    DWORD dwWakeMask,
    DWORD dwFlags
);
```

The difference between these and the ordinary wait APIs is simple: if a new message arrives in the thread's message queue before the wait is satisfied, the API returns so that the caller can process the new message. Everything you learned about the `WaitForMultipleObjectsEx` API earlier applies here: the return value can be `WAIT_OBJECT_0 + i`, where *i* is the index of the `HANDLE` that was signaled and falls in the range of 0 to `nCount - 1`, inclusive, `WAIT_ABANDONED_0 + i`, `WAIT_TIMEOUT`, `WAIT_IO_COMPLETION`, or `WAIT_FAILED`. The single new return value that indicates a message has arrived is `WAIT_OBJECT_0 + nCount`. Notice this returns a value that is one greater than the legal range when a specific object is signaled.

The `dwWakeMask` argument is used to specify what type of messages will cause the wait to return. `QS_ALLINPUT` will wake up when any message arrives. Please consult the Windows SDK documentation for details on the other available options, as there are legitimate cases where you might want to limit the type of messages you will process. To ensure the wait is alertable wait, the `MsgWaitForMultipleObjectsEx` API can be used, passing a `dwFlags` argument containing the value `MWMO_ALERTABLE`.

When the wait returns because a message has arrived, you must process messages in the queue by running the window's message loop. If you do not, future calls to this (and most related) API(s) will ignore existing messages because they are no longer considered "new." Similarly, when `PeekMessage` is used, the message seen is not considered "new" any longer either. Passing the flag value `MWMO_INPUTAVAILABLE` to `MsgWaitForMultipleObjectsEx` will process messages that already exist in the queue, overriding the default behavior (noted above) to only return when a new

message arrives: any message in the queue, new or otherwise, will cause the wait to return. All of these corner cases make for some pretty complicated boilerplate code, so most applications tend to rely on a single wait routine that is common to the entire code base and reused from one application to the next. Here is one (simplified) example.

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

DWORD DoWait(const HANDLE * pHandles, int cHandles,
             DWORD dwMilliseconds, BOOL bAlertable)
{
    DWORD dwRet;
    DWORD dwStart = GetTickCount();
    DWORD dwElapsed = 0;

    while (TRUE)
    {
        // Now do the actual wait.
        dwRet = MsgWaitForMultipleObjectsEx(cHandles,
                                            pHandles,
                                            dwMilliseconds - dwElapsed,
                                            QS_ALLINPUT,
                                            bAlertable ? MWMO_ALERTABLE : 0);

        if (dwRet == WAIT_OBJECT_0 + cHandles)
        {
            // At least one message has arrived. Drain the queue.
            MSG msg;
            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
            {
                if (msg.message == WM_QUIT)
                {
                    PostQuitMessage((int)msg.wParam);
                    dwRet = WAIT_TIMEOUT;
                    break;
                }

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

        // If a quit message was posted, quit.
        if (dwRet == WAIT_TIMEOUT)
            break;
    }
    else if (dwRet != WAIT_IO_COMPLETION)
```

```
{  
    // If not an APC, we will break and return the value.  
    break;  
}  
  
// We have to readjust the time, verify we haven't timed out;  
// then just loop back around to try the wait again.  
dwElapsed = GetTickCount() - dwStart;  
if (dwMilliseconds < dwElapsed)  
{  
    dwRet = WAIT_TIMEOUT;  
    break;  
}  
}  
  
return dwRet;  
}  
  
int wmain(int argc, wchar_t * argv[])  
{  
    HANDLE handles[5];  
    for (int i = 0; i < 5; i++)  
        handles[i] = CreateEvent(NULL, TRUE, FALSE, NULL);  
  
    DWORD dwWaitRet = DoWait(handles, 5, 1000, TRUE);  
    printf("Wait returned: %u\r\n", dwWaitRet);  
  
    for (int i = 0; i < 5; i++)  
        CloseHandle(handles[i]);  
  
    return 0;  
}
```

Notice that we break under a couple of circumstances. If the wait returns a timeout, we can return immediately. If the wait returns and indicates that we have a message, we will drain the message queue. Note that when we encounter a quit message, we must exit the wait entirely. We've overloaded the WAIT_TIMEOUT return value, but for application-wide routines it is a good idea to use something else. The idea is that the caller must return, and so on, and we will get back to the top-level modal loop quickly, which will quit the program. As shown earlier, we will just go back around and reissue the wait if an APC happened. Otherwise, we simply return the code returned by the wait API, for example, a successful wait, abandoned mutex, and so forth.

We only described wait-any waits above and for good reason. It's not that you can't do a wait-all wait—the APIs certainly do support it. In the case of `MsgWaitForMultipleObjects`, you must specify TRUE as the value for `bWaitAll`, and for `MsgWaitForMultipleObjectsEx`, you supply a `dwFlags` argument containing the value `MWMO_WAITALL`. However, this brings up a very thorny issue.

If you didn't stop to think of it earlier, did you wonder why the value returned during a wait-any wait when a message arrives is `WAIT_OBJECT_0 + nCount`? It's subtle. The implementation of the message wait APIs just append an internal event handle to the `pHandles` array supplied as input, increment the count by one, and then pass that to the standard `WaitForMultipleObjectsEx` wait API instead. This is why you can only supply one less than `MAXIMUM_WAIT_OBJECTS` handles for a message wait. Why does this matter? If you specify a wait-all wait, the wait will not return when all of the handles in your array are signaled; instead, it must wait for all of them to be signaled as well as a new message to arrive in the thread's message queue. This is typically not what you want and can easily lead to an application that seems frozen and will only wake up when the user nudges the mouse.

The CLR helps to avoid this problem by throwing an exception when you call `WaitHandle.WaitAll` on a Single Threaded Apartment (STA) thread, because the CLR always pumps messages automatically (we'll look at that soon). But if you're writing native code, you'll have no luck and need to be careful.

`CoWaitForMultipleHandles`. It is inconvenient to have to write the preceding boilerplate message pumping code in all of your GUI and COM programs. Because of this very reason, on Windows 2000 and later, there is a special `CoWaitForMultipleHandles` API defined in `objbase.h` and exported from `OLE32.LIB`.

```
HRESULT CoWaitForMultipleHandles(
    DWORD dwFlags,
    DWORD dwTimeout,
    ULONG cHandles,
    LPHANDLE pHandles,
    LPDWORD lpdwIndex
);
```

The function signature is very similar to `MsgWaitForMultipleObjects`. The `dwFlags` argument may contain 0 or more of the flags `COWAIT_WAITALL` (0x01) or `COWAIT_ALERTABLE` (0x02). As you may well imagine, the first specifies that a wait-all (rather than the default of wait-any) is desired, and the latter ensures that pending APCs are dispatched by the OS kernel. This function encapsulates poorly documented, mysterious logic that will automatically pump certain classes of messages. Specifically, when the wait occurs on a Single Threaded Apartment (STA), COM RPC messages are processed, and only a subset of the possible windowing messages are processed, via the `MsgWaitForMultipleObjectsEx` function. When called from a thread in a different apartment type, the call simply passes through to the `WaitForMultipleObjectsEx` API.

When to Pump Messages. Deciding when to pump messages is seldom straightforward. Not doing so, in the best case, is completely harmless (if a message never arrives during the wait). In the worst case, it can cause a deadlock that brings the program to its knees. Somewhere in the middle fall performance issues, which can vary between minor impacts to throughput (in the case of, say, COM on the server) or GUI responsiveness, and major impacts that destroy a server's performance or give users the impression that their GUI is hung, causing them to kill the application, possibly indirectly corrupting data in the process.

At the same time, pumping causes reentrancy. Reentrancy is caused when some logically unrelated piece of work enters on top of the existing callstack. If you pump messages during a blocking operation, this code seems to execute "in the middle" of the wait. If there is any thread specific state established at the time this reentrancy occurs, application behavior can go haywire, often leading to state corruption. For example, if a mutex is held when reentrancy occurs, it will be accidentally shared between the code that was active before the reentrancy and the reentrant code itself, due to mutex recursion. The decision to pump and risk reentrancy must be made carefully and must include consideration and precautions to ensure that application state invariants are prepared to handle the possibility of reentrancy.

The decision of whether to pump is often also informed by the length of a blocking operation. If you're doing GUI programming, you really ought to avoid all blocking on the GUI thread (as already noted). In some

circumstances, however, the overhead required to marshal work to a separate thread versus a short expected wait time may mean that staying on the GUI thread and doing a little pumping is appropriate. (Beware! This is a slippery slope!) These cases really ought to be rare. Often what seems like a short wait time can turn out to be forever under unexpected circumstances, such as trying to resolve a DNS entry when your user's network cable has just become unplugged. Most GUI frameworks will automatically pump messages when modal dialog boxes are shown. With COM it's seldom so straightforward, because the sole purpose of sending and pumping for messages is for cross-thread synchronization. And so, in order to avoid deadlocks, pumping is typically inescapable.

For sophisticated applications, choosing when to pump on a case-by-case basis is reasonable, but for most applications, deciding to always (or never) pump messages on threads with message queues can simplify your life quite a bit. A popular approach is to pump COM messages, but not GUI messages, as we saw with the `COWaitForMultipleHandles` API. This at least homogenizes the categories of failures you are apt to see in your code base, and lets you opt-in specific call sites after the fact in response to testing and bugs. The CLR similarly chooses to always pump messages when it's on a GUI or COM STA thread, as in `COWaitForMultipleHandles`, which brings us to the next topic: how the CLR waits.

Managed Code

Now we turn to the way in which managed code interoperates with Windows kernel synchronization. Everything mentioned here is, effectively, a thin veneer over everything we just discussed in the context of native code.

A Common Base Class: `WaitHandle`

The CLR directly exposes four out of the five kernel synchronization objects we are interested in for this chapter: mutexes, auto-reset events, and manual reset events, and semaphores. Each kernel object is represented by an instance of a different `System.Threading.WaitHandle` subclass. `WaitHandle` houses all common waiting functionality; in other words, it provides the managed equivalent to Win32's `WaitForSingleObject`, et. al.

```
System.Threading.WaitHandle
    EventWaitHandle
        AutoResetEvent
        ManualResetEvent
    Mutex
    Semaphore
```

The wait methods of interest on the `WaitHandle` class are:

```
public virtual bool WaitOne();
public virtual bool WaitOne(int millisecondsTimeout, bool exitContext);
public virtual bool WaitOne(TimeSpan timeout, bool exitContext);

public static bool WaitAll(WaitHandle[] waitHandles);
public static bool WaitAll(
    WaitHandle[] waitHandles,
    int millisecondsTimeout,
    bool exitContext
);
public static bool WaitAll(
    WaitHandle[] waitHandles,
    TimeSpan timeout,
    bool exitContext
);

public static int WaitAny(WaitHandle[] waitHandles);
public static int WaitAny(
    WaitHandle[] waitHandles,
    int millisecondsTimeout,
    bool exitContext
);
public static int WaitAny(
    WaitHandle[] waitHandles,
    TimeSpan timeout,
    bool exitContext
);
```

The instance method, `WaitOne`, is used to wait for a single object to become signaled. The `WaitAll` and `WaitAny` static methods wait for all of the objects in the array or any single object in the array to become signaled, respectively. Both APIs validate the array input and throw various exceptions if the array is `null`, any of the elements are `null`, or if there are duplicates found in the array. Each of the APIs throws an `AbandonedMutexException` to indicate that one of the elements refers to a mutex that has been abandoned (which we still haven't explained but will soon.)

Each of the waiting APIs supports an optional timeout argument, specified as either an `int` or a `TimeSpan` value. The `System.Threading.Timeout` class has a single constant (of type `int`), `Infinite`, which can be passed to indicate that the call will never timeout. This is the default behavior of the no timeout versions of these APIs, that is, those overloads that take no parameters. The `WaitOne` and `WaitAll` methods return a value of `true` to indicate that the return was caused by the object(s) becoming signaled, or `false`, if the timeout was exceeded before the object(s) became signaled. A timeout value of `0` (or new `TimeSpan(0)`) will simply check the object's or set of objects' status and return immediately without blocking. Because `WaitAny` uses the return value to indicate the index of a signaled object, it will return the constant value `WaitHandle.WaitTimeout` if the timeout was exceeded.

The timeout overloads of these methods have a mysterious `exitContext` argument. This is used for COM interoperability and controls whether the current synchronization context is exited before waiting or not. If you're a COM programmer, you may recognize the danger of deadlock if you wait without exiting the synchronization context. Otherwise, you should pass `false`. It's cheaper because the call doesn't incur a conditional context exit and reentrance before and after the wait and will have no noticeable effect on your program's correctness.

`WaitHandle` itself does not have a finalizer. Instead, it has a private `SafeWaitHandle` that encapsulates the Win32 `HANDLE` that is being wrapped. This object has a critical finalizer that will close the handle when all references to the safe handle have been dropped. You can still access the raw handle as an `IntPtr` via the `WaitHandle.Handle` property, but this has been deprecated because `IntPtr` handles have been proven to lead to security problems. Relying on the critical finalizer to clean up unused kernel objects is wasteful and eats up finite system resources, so you should take care to call `Dispose` or `Close` on the `WaitHandle` (both of which do the same thing) when you're finished using it.

How the CLR Waits

The CLR controls the mechanics of waiting so that you don't have to worry about many of the things mentioned earlier, such as restarting the wait after

APCs have occurred, pumping for messages on GUI and COM STA threads, and doing all the error prone timeout adjustments. In fact, because the CLR uses one common waiting routine whenever you block, regardless of whether it's due to a call to `WaitHandle.WaitOne`, `WaitAny`, `WaitAll`, `Thread.Join`, or any blocking calls on managed locks, such as `Monitor` or `ReaderWriterLock`, the CLR waits consistently for all managed code. Thanks to this, CLR hosts and custom `SynchronizationContext` implementations can override the CLR's waiting logic to perform bookkeeping or to make scheduling decisions.

On Windows 2000 or later, the CLR calls directly to the COM `CoWaitForMultipleHandles` API reviewed previously. On older OSs, the CLR uses some handwritten message pumping code that calls `MsgWaitForMultipleObjectsEx` when the wait occurs on an STA thread and `WaitForMultipleObjectsEx` otherwise. These waits are alertable. Both the pre-Windows 2000 and Windows 2000 behaviors prefer to pump COM RPC messages and not all GUI messages. If you wish to explicitly pump GUI messages in managed code, there are GUI framework-specific APIs to do so: for example, `System.Windows.Forms.Application.DoEvents` in Windows Forms and `System.Windows.Threading.Dispatcher.PushFrame` in Windows Presentation Foundation.

Finally, knowing precisely what the CLR is doing might tempt you to call the native wait APIs directly with P/Invoke. The fact that you have fine-grained control over how waiting actually happens might be attractive, but it is a bad idea. Everything mentioned here is effectively an implementation detail and is subject to change as the CLR evolves. Moreover, if you bypass the CLR's internal wait logic, the CLR is unable to cooperate with thread interruptions, aborts, and hosts. There have been instances of .NET APIs themselves that do this, but they tend to get cleaned up over time.

Interruption

When a managed thread has begun waiting or sleeping, it will be blocked in the kernel and its state will be `WaitSleepJoin`. If some other thread determines that the thread needn't wait any longer, it can be awakened with a call to the `ThreadInterrupt` instance method.

```
public void Interrupt();
```

Provided that the target thread is waiting by cooperation with the CLR itself, calling this API will unblock the thread and raise a `ThreadInterruptedException`. If a thread isn't waiting when the call is made, the next subsequent waits will trigger the exception. If the thread never waits, the interruption request may go entirely unnoticed. One caveat is worth noting: on .NET 2.0 and greater, thread interruptions aren't processed if the target thread is blocked in a catch or finally block. While interruption is safer than using asynchronous thread aborts (see Chapter 3, Threads), it is still generally unsafe to use against arbitrary code. Interrupts are implemented inside the CLR, so the potential points at which an interruption may be processed are carefully controlled and limited to blocking calls. Compare this to asynchronous thread aborts, which may occur almost anywhere. However, much of the code written in the .NET Framework, third party libraries, and applications may not have been written to deal correctly with the possibility of interruption exceptions being thrown from wait calls. If you decide to use interruption, you should carefully test that the code surrounding all of the interruptible blocking points in the code will continue to function correctly in the face of exceptions.

Asynchronous Procedure Calls (APCs)

Each thread has an asynchronous procedure call (APC) queue into which any thread in the process may place a new APC entry. An entry is a function-pointer/argument pair, which is run in the context of the thread when it next enters an alertable wait state. APCs can be enqueued across threads. The kernel uses APCs for many interrupt-like activities, and user-mode code can use them to hijack a blocked thread.

Two kinds of APCs exist: **kernel-mode** and **user-mode**. Most, but not all, APCs in practice run in kernel-mode and are like interrupts in that they asynchronously interrupt execution of a thread any time it's in user-mode (and only at specific interrupt request levels [IRQLs] in kernel-mode). This kind of APC is generally only interesting to people writing device drivers.

Whenever a thread performs an alertable wait, by passing a bAlertable argument of TRUE to one of the wait APIs shown above (assuming the handle[s] being waited for haven't been signaled), the kernel will automatically dispatch all of the thread's outstanding APCs before blocking. Similarly, calling SleepEx with a bAlertable argument value of TRUE also dispatches the thread's APCs. Dispatching the thread's APCs means that all APC pairs (*fp*, *arg*) in the queue—where *fp* is the function pointer and *arg* is the argument, each supplied when the APC was queued—are invoked: **fp*(*arg*). APCs are called in strictly FIFO order and run in the context of the thread queue from which the APC was taken.

In the case of both the wait APIs and SleepEx, the functions return a value of WAIT_IO_COMPLETION after running all of the thread's APCs, and the caller must then decide what to do. As we saw earlier, often this means just readjusting a timeout counter and retrying the original wait or sleep operation. If some thread is already in a wait state and another thread asynchronously places an APC into its queue, then the target thread will become runnable and placed into the scheduler's queue. It will then dispatch the APC as soon as it is scheduled.

User-mode APCs are somewhat rare in practice, but are used in some parts of Win32 itself, the most notable of which is asynchronous file I/O. (To find out more on asynchronous file I/O, refer to Chapter 15, Input and Output.) User-mode APCs are also exposed directly to Win32 programmers as of Windows 2000 via the QueueUserAPC function and can be used as a synchronization mechanism between threads.

```
DWORD WINAPI QueueUserAPC(
    PAPCFUNC pfnAPC,
    HANDLE hThread,
    ULONG_PTR dwData
);
typedef VOID (CALLBACK * PAPCFUNC)(ULONG_PTR dwParam);
```

The arguments *pfnAPC* and *dwData* represent the function-pointer/argument pair, and the *hThread* argument specifies the thread queue into which the APC will be placed. The callback function type has a VOID return type and a single *dwParam* parameter; the argument passed during callback invoke is the *dwData* pointer supplied at APC creation time.

In some circumstances, APCs can represent a lightweight interthread communication mechanism. If you know the HANDLE of a thread you wish to signal, and that thread has performed an alertable wait, then queueing an APC is often significantly quicker than waking the target thread by using kernel objects (as we are about to review). It does require kernel transitions on the caller and callee, but direct thread-to-thread communication is faster than the general purpose kernel objects that must handle a variety of other difficult conditions.

That said, APCs should be used with extreme care. They introduce a form of reentrancy, which can cause reliability problems in both native and in managed code alike. The thread performing the alertable wait has no control over what the APC actually does. This means, for instance, that the APC could wait for things alertably, dispatching more APCs on the thread (recursively) if these are alertable waits too. This can lead to messy situations because you may end up with a single stack that is a hodgepodge of multiple logical activities.

Other problems abound. If the APC waits for a mutex object that the thread already owns, then the APC will be granted access to it even though data protected by the mutex might be in an inconsistent state due to recursion. (See the section on mutexes in a few pages for details on mutex recursion.) If the APC triggers an exception, it will possibly rip through the entire call stack present at the time of the original alertable wait, unless the authors had the foresight to wrap all calls to `WaitForSingleObjectEx`, and so forth inside a `_try/_catch` block and somehow managed to intelligibly respond, such as reissuing the wait. This is seldom feasible because reentrancy is unpredictable.

In managed code, there are unique problems. If you P/Invoke to `QueueUserAPC`, the APC might be subsequently dispatched when managed code can't be run, such as while certain critical regions of code in the CLR are executing. This could lead to deadlocks in cases where nonrecursive locks are used. And it might even happen in the middle of a garbage collection, while the GC is blocked. And then who knows what will happen? Finally, this can introduce security vulnerabilities into your code because, unlike proper mechanisms of queuing asynchronously work, the CLR will not have a chance to capture and restore a security context.

Using the Kernel Objects

Now that we've reviewed the basics that apply to all kernel objects, let's drill into each of the synchronization specific objects: mutexes, semaphores, auto- and manual-reset events, and waitable timers, in that order.

Mutex

The mutex—also referred to as the mutant in the Windows kernel—is a kernel object that is meant solely for synchronization purposes. A mutex's purpose is to facilitate building the *mutually exclusive* (hence the abbreviated name *mut-ex*) critical regions of the kind that were introduced in Chapter 2, Synchronization and Time. The mutual exclusion property is accomplished by the mutex object transitioning between the nonsignaled and signaled states atomically. When a mutex is in the signaled state, it is available for acquisition; that is, there is no current owner. A subsequent wait will atomically transfer the mutex into a nonsignaled state. It is atomic because the Windows kernel handles cases in which multiple threads wait on the same mutex simultaneously; that is, only one will be permitted to initiate the transition, while the other will see the mutex as nonsignaled. When a mutex is nonsignaled, there is a single thread that currently owns the mutex.

Mutex ownership is based on the physical OS thread used to wait on the mutex in both native and managed code. This allows Windows to provide errors in cases where a thread erroneously tries to release a mutex when it isn't the current owner. In other synchronization primitives, such as events, this condition isn't caught although it (usually, but not always) represents an error in the program. For systems in which logical work might migrate between separate threads, or where multiple pieces of logical work might share the same physical thread, this can pose problems. Such is the case for fibers, as described in Chapter 9, Fibers, because multiple fibers can be multiplexed onto the same OS thread and can even migrate between them over time. The CLR denotes the acquisition and release of affinity through the use of the `Thread.BeginThreadAffinity` and `EndThreadAffinity` APIs to notify hosts when affinity has been acquired and released, corresponding to the acquisition and release of a mutex object, respectively, allowing hosts to deal with this situation.

As an illustration, here are two side-by-side code snippets that use a mutex to build a critical region: the left is written in C++ using Win32 and the right is C#.

```
HANDLE hMutant = CreateMutex(...);
...
WaitForSingleObject(hMutant, INFINITE);
try
{
    // The critical region.
}
finally
{
    ReleaseMutex(hMutant);
}
...
CloseHandle(hMutant);
```

```
Mutex mutant = new Mutex();
...
mutant.WaitOne();
try
{
    // The critical region.
}
finally
{
    mutant.ReleaseMutex();
}
...
mutant.Close();
```

Notice that in native code, a mutex is referred to by its HANDLE, while in managed code, a mutex is referred to by an instance of the Mutex class. The Mutex class derives from the common kernel object type System.Threading.WaitHandle in the .NET Framework. All error checking has been omitted from the native example for brevity, although a real program should check the return value of each API call. Let's now review the mutex APIs in detail.

Creating and Opening Mutexes

To create a new mutex kernel object in Win32, you use either CreateMutex or, as of Windows Vista, CreateMutexEx.

```
HANDLE WINAPI CreateMutex(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    BOOL bInitialOwner,
    LPCTSTR lpName
);
HANDLE WINAPI CreateMutexEx(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    LPCTSTR lpName,
    DWORD dwFlags,
    DWORD dwDesiredAccess
);
```

Each function returns a HANDLE to the created mutex object. If bInitialOwner is TRUE in the case of CreateMutex, or if dwFlags contains the value CREATE_MUTEX_INITIAL_OWNER in the case of CreateMutexEx, then the

resulting mutex object will have been created with the calling thread as the owner, and the mutex will be in a nonsignaled state. This ensures another thread in the system cannot locate the mutex (e.g., via a name lookup) before the caller is able to acquire the mutex, if that is desired.

Both APIs take an optional security descriptor to control subsequent access to the created mutex object. You can pass `NULL` if you don't have special security attributes, as is often the case. The `lpName` argument can be used to name the mutex. If you don't require a name, `NULL` can be passed as the argument. This is only useful if you intend to share the mutex across processes, or if you need to look up the mutex by name later on. Because any program on the machine can create a mutex with the same name you have chosen (by accident or otherwise), you should carefully name them and ensure they are properly protected by ACLs. Despite your best efforts, programs exist that will dump named mutexes on the machine. Specifying security attributes is also recommended when naming a kernel object. Finally, `dwDesiredAccess` is used to specify a certain set of access rights desired by the thread, which gets stored in the process handle table. We will omit any detailed discussion of kernel object security in this book. Please refer to existing books on this topic (see Further Reading, Brown) for thorough explanations and tutorials.

Either of these functions can fail. If the failure is catastrophic, the return value will be `NULL`, and `GetLastError` must be used to retrieve detailed information about it. If a name is given, and a mutex already exists under the given name (machine-wide), the return value will be a `HANDLE` to this existing mutex. This ensures many threads can race with one another to create a mutex with the same name, and only one mutex object will be shared among them. But in this case, `GetLastError` will then return `ERROR_ALREADY_EXISTS`, allowing you to detect this case. This is an important condition to code for when you specify that the caller should be the initial owner of the mutex. In the case that the mutex already exists, this request is ignored and the mutex will not be acquired before returning. If your code blindly proceeds as though it owns the mutex, the result will be equivalent to a race condition.

There is an equivalent to all of this in the .NET Framework. To create a new mutex object, you instantiate a new `Mutex` object using one of its constructors. This is a thin wrapper on top of the Win32 APIs shown previously.

```
public Mutex();
public Mutex(bool initiallyOwned);
public Mutex(bool initiallyOwned, string name);
public Mutex(bool initiallyOwned, string name, out bool createdNew);
public Mutex(
    bool initiallyOwned,
    string name,
    out bool createdNew,
    MutexSecurity mutexSecurity
);
```

The simple no argument overload always creates a new mutex object initialized to a signaled state. The second overload, which takes an `initiallyOwned` flag, does the same, except that it will create the mutex in a nonsignaled state with the current thread as the owner, if `initiallyOwned` is true. (If it's `false`, behavior is the same as the no argument overload.) As soon as you start to use named mutexes, things become more complicated. If you specify a `name` argument and a mutex already exists with that same name, the new mutex object will reference that kernel object. Otherwise, a new kernel object is created for you. The methods with an output parameter `createdNew` indicate which case occurred; that is, a value of `true` means the mutex didn't already exist and was created, while `false` means a reference to an existing mutex kernel object has been returned. The `mutexSecurity` argument can be used to specify the desired access control list for the resulting mutex object, which clearly only applies when creating a new mutex and is ignored otherwise.

Just as with the Win32 APIs, if you specified an `initiallyOwned` value of `true`, and yet `createdNew` ends up being `false`, the mutex object will not be owned by the calling thread. It is crucial you check this value and acquire the mutex before proceeding, otherwise your critical region may not enjoy mutual exclusion, depending on which thread creates the mutex first. Safe code typically looks a bit like this:

```
bool createdNew;
Mutex mutex = new Mutex(true, "...", out createdNew);
if (!createdNew)
    mutex.WaitOne();
... critical region, release, etc. ...
```

As with any `HANDLE` APIs in Win32, the handle returned from `CreateMutex` must be closed eventually with the `CloseHandle` API. As soon

as the last handle to the mutex is closed, the kernel object manager will destroy the object and reclaim its associated resources. The .NET Framework's `Mutex` class implements `IDisposable`: calling either `Close` or `Dispose` will eagerly release the sole handle when you know for sure you're done using it. The handle is protected by a critical finalizer, ensuring it will always be closed even if you forget to do so yourself, but eagerly closing it is a good practice and alleviates GC finalization pressure.

Sometimes you might know that a mutex object already exists under some name. Perhaps all mutexes used by your program are initialized during the program's startup routine, for example, such that the existing mutex couldn't be found by name, it would represent a program error. Instead of relying on the `CreateMutex` and `CreateMutexEx` APIs and `Mutex` constructors to do the right thing and having to check the error codes and return values described above, you can open the existing object directly with dedicated APIs.

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

The `OpenMutex` function returns `NULL` if the mutex kernel object cannot be found under the given name, and `GetLastError` will return `ERROR_FILE_NOT_FOUND`. The `dwDesiredAccess` parameter, as with `CreateMutex`, and so forth, indicates what permissions the resulting `HANDLE` should have. And `bInheritHandle` specifies whether child processes created by the current process can inherit and use the `HANDLE`.

You can do the same thing in managed code via `Mutex`'s `OpenExisting` static APIs.

```
public static Mutex OpenExisting(string name);
public static Mutex OpenExisting(string name, MutexRights rights);
```

Both methods throw a `WaitHandleCannotBeOpenedException` if no mutex kernel object was found in the system under the given name. The `MutexRights` argument, as with `dwDesiredAccess` for `OpenMutex`, specifies what rights the resulting `Mutex` object reference must have.

(Note that in the initial release of Windows Server 2003, there was a bug [see MS KB article 889318] that allowed two mutexes with the same name

to be created at the same time. This happened if two threads were racing to call `OpenExisting` and `CreateMutex` simultaneously: the `OpenExisting` would fail to see the mutex created by the other thread, and then, if called quickly enough, the subsequent call to `CreateMutex` would create another mutex under the same name. The results of this are disastrous because programs think they are using mutexes to achieve mutual exclusion but aren't. This was fixed in SP1 of Windows Server 2003, and the CLR Mutex object has a special case [only active on the affected versions of Server 2003] to work around this: it acquires an internal machine-wide mutex that, in effect, serializes all calls to create or open mutexes across the whole machine.)

Acquiring and Releasing Mutexes

Because mutexes facilitate mutual exclusion by the way that they atomically transition from the signaled to nonsignaled state, a mutex is acquired by waiting on it. This is done with any of the wait APIs described earlier in this chapter, that is, `WaitForSingleObject`, `WaitForMultipleObjects`, and so forth, in native code, and `WaitHandle.WaitOne`, `WaitAny`, or `WaitAll` in managed code. When the API returns successfully, the mutex has been acquired by the current thread and marked as nonsignaled. No other thread will be able to acquire the mutex until the owning thread releases it, transitioning the mutex back into a signaled state. In Win32, releasing the mutex is done with the `ReleaseMutex` API.

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

And in the .NET Framework, this is just a method call to the `ReleaseMutex` instance method on the `Mutex` class.

```
public void ReleaseMutex();
```

If the calling thread does not own the mutex, the Win32 API will return FALSE and `GetLastError` will return a value of `ERROR_NOT_OWNER` (288L). The .NET Framework throws an exception of type `ApplicationException` for the same condition.

Once a mutex has been released, it becomes signaled again, and other threads may acquire it. As described earlier, if there are any threads waiting for the mutex, the kernel uses a FIFO algorithm to track waiters and, hence,

which thread to wake up. Windows will wake only one of the waiting threads, since waking multiple threads would lead to all but one having to rewait anyway. Mutexes are fair in the sense that when a thread is wakened from a wait, it is guaranteed to be the next thread to acquire the mutex. This ensures that no other thread can sneak in and enter the mutex before the awakened thread becomes scheduled. While this might sound like a nice feature, it can lead to an increased rate of lock convoys, a phenomenon described more in Chapter 11, Concurrency Hazards. Priority boosts, as described in Chapter 4, Advanced Threads, increase the chance of the thread getting scheduled in a timely manner, which helps to alleviate the occurrence of lock convoys, but only slightly.

Effectively all locks on Windows were fair prior to Windows Server 2003 R2 and Windows Vista. In the newer operating systems, many locks, such as CRITICAL_SECTIONS and kernel pushlocks, have been made unfair to improve scalability and to help reduce convoys. Mutexes remain unaffected, however. We discuss this more in the next chapter.

The mutex object supports recursive acquires. That means that if the owning thread waits on the mutex, the wait is satisfied immediately, even though the object is nonsignaled. An internal recursion counter is maintained, starts at 0, and is incremented for each mutex acquisition. For each successful wait on the mutex, a paired call to release the mutex must be made to decrement this counter accordingly. Only when the mutex's recursion counter drops back to the original value of 0 will the kernel object become signaled and available to other threads, and any waiting threads are awakened. Recursion may seem like a convenient feature, but it turns out to produce brittle designs that can lead to reliability problems. Please refer to Chapter 11, Concurrency Hazards, for more details on recursion in general.

Abandoned Mutexes

Throughout this chapter, we've encountered a few circumstances in which the topic of abandoned mutexes arose, that is, in the return values of the wait APIs. We've deferred a detailed discussion until now. An abandoned mutex is a mutex kernel object that was not correctly released before its owning thread terminated. This can happen for any number of reasons.

Perhaps there is a bug in somebody's code and they forgot to release the mutex (or didn't release it enough times, in the case of recursive acquires). Or maybe they remembered to use a try / finally block, but for some reason, the finally block didn't get a chance to execute. This could happen if they are using a machine-wide mutex in a program that gets terminated abruptly, for example, with `ExitProcess` or by acquiring and releasing it from a CLR background thread that was destroyed during process exit. As we reviewed in Chapter 4, Advanced Threads, there are many cases in native and managed code where finally blocks are not run during process shutdown, and, therefore, any finally blocks on the stack that would have released the mutex won't get a chance to run. An abandoned mutex is problematic because it indicates a potential problem with the state protected by that mutex: some code never finished running the critical region, and, therefore, may have left partial state updates and corruption in its wake.

As soon as the mutex is abandoned, no other thread would be able to acquire it without help from the OS, because it's still marked as being owned. This is called orphaning and is discussed more in the next chapter (particularly since most synchronization primitives don't tolerate orphaning in the same way that mutexes do). The OS deals with this problem fairly elegantly. If a mutex is abandoned with waiting threads, a waiting thread will be awakened as though the abandoning thread released it. However, when this thread wakes up, it will be told that the mutex has been abandoned via the return value. If no waiting thread was awakened, the next thread to wait on the mutex is notified. Specifically, the Win32 single object wait functions `WaitForSingleObject` and `WaitForSingleObjectEx` will return `WAIT_ABANDONED` and the multiple object APIs `WaitForMultipleObjects` and `WaitForMultipleObjectsEx` will return `WAIT_ABANDONED_0 + i`, where *i* is the index of the abandoned mutex in the array of `HANDLE`s. In managed code, `WaitHandle`'s wait APIs will throw an `AbandonedMutexException`. In the case of a `WaitHandle.WaitAny` or `WaitAll`, the index of the mutex (from the array argument passed to the API) is captured in the exception's `MutexIndex` property and the `Mutex` object itself is accessible from the `Mutex` property. Despite receiving an error code or exception, when an abandoned mutex is discovered, the calling thread will have successfully acquired the mutex. This is important—it means the thread must

release the mutex when it completes the critical region, just as with any successful acquire.

Be careful when using a wait-all style wait on an array that contains more than one mutex. The `WAIT_ABANDONED_0 + i` scheme is only capable of communicating the first abandoned mutex encountered in the array. And because the CLR's `AbandonedMutexException` builds on top of this same basic support, it too can only communicate one such mutex in the `Mutex-Index` property. If several mutexes were abandoned, you will only be told about the first one, possibly masking a severe data corruption problem.

In any case, you must worry about abandoned mutexes. Abandonment is often an indication that a thread failed to finish updates it was making to shared state, possibly leaving this state corrupted. Similarly, for machine-wide mutexes, any resources or cross-machine state that the mutex protects is now suspect. What can you do in response? In some cases, you can verify the integrity of state by checking data invariants. If you can prove that the state is valid—or you can repair the state if it was indeed found to be damaged—then the program can typically proceed as normal. Often this is not easily determinable, however, and you may instead ask the user to verify that state is OK, ask them to restart the process or, in the case of machine-wide state, reboot the machine to fix things. If the corruption has to do with persistent state, the recovery task is sadly often much more tricky to orchestrate.

Semaphore

The basic counting semaphore idea was mentioned in Chapter 2, Synchronization and Time. In summary, threads may perform a take or put operation on a semaphore, atomically decreasing or increasing its current count, respectively. When a thread tries to take from a semaphore that already has a count of 0, the thread blocks until the count becomes non-0. This allows a special kind of critical region that is not mutually exclusive; rather, a specific number of threads is permitted to be inside the region. It turns out that more sophisticated patterns are possible too: it is not necessary to use them solely for critical regions, as we'll see later with an example implementation of a bounded buffer data structure. Note that, unlike mutexes, semaphores are never considered to be "owned" by a

specific thread. One thread can safely put and another thread can take from the same semaphore, for example.

Semaphores are typically used to protect resources that are finite in capacity. For example, you might have a pool of database connections fixed in size and need to regulate access such that more connections than are available are not requested at once. Similarly, you might have a shared in-memory buffer with a fluctuating size but need to guarantee only as many threads as there are available buffer items access to the buffer at once. Semaphores are not a replacement for the kind of data synchronization necessary for avoiding concurrency hazards. Semaphores with a count greater than 1 do not guarantee mutual exclusion, but rather help to implement common control synchronization patterns like producer/consumer.

The rules for when a thread may acquire a semaphore generally map to kernel objects: when the count is non-0, the semaphore is signaled, and once the count reaches 0, the semaphore becomes nonsignaled. Windows supports two additional features. First, a semaphore can be given a maximum count, which prevents threads from adding to a semaphore if its count has already reached the maximum. Second, a thread may put an arbitrary count back into the semaphore, rather than being limited to just putting a count of 1. As the semaphore transitions from nonsignaled to signaled, the Windows kernel will wake as many waiting threads as the count specified and no more. For instance, when you release N counts to the semaphore, Windows will wake up, at most, the first N waiting threads found in the wait queue. If there are fewer than N threads waiting, say M, then only M threads are awakened, and the next $N - M$ threads to wait on the semaphore will succeed in taking from it without having to wait. As with all other kernel objects, waiting threads are kept in a FIFO order. All of our previous discussions about APCs apply to semaphores too, meaning that this FIFO ordering is regularly disturbed and that you shouldn't take any sort of dependency on it.

Creating and Opening Semaphores

Creating and opening a semaphore kernel object is done similar to mutexes, as shown earlier. Because we already thoroughly discussed this topic

above, there is no need to do it again. Therefore, the following discussion will describe only the details specific to semaphores.

The `CreateSemaphore`, `CreateSemaphoreEx` and `OpenSemaphore` APIs can be used to create a new (optionally named) semaphore or open an existing one by name.

```
HANDLE WINAPI CreateSemaphore(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTSTR lpName
);
HANDLE WINAPI CreateSemaphoreEx(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount,
    LONG lMaximumCount,
    LPCTSTR lpName,
    DWORD dwFlags,
    DWORD dwDesiredAccess
);
HANDLE WINAPI OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName
);
```

Both `CreateSemaphore` APIs take a `lpSemaphoreAttributes` argument to specify the access control on the resulting object and a `lpName` argument if you wish to share and access the semaphore by name. Either or both arguments can be `NULL` if you do not care about assigning object security or naming. As with `CreateMutexEx`, the `CreateSemaphoreEx` API is new to Windows Vista. But its `dwFlags` argument is reserved, meaning that you must always pass `0`; thus the only advantage it provides over `CreateSemaphore` is that you can specify the `dwDesiredAccess` mask, which represents the rights granted to the resulting `HANDLE` that is returned.

In the .NET Framework, any one of `System.Threading.Semaphore`'s constructors can be used to create a new semaphore object. Or, as with `Mutex`, one of the static `OpenExisting` overloads can be used to open an existing semaphore kernel object by name.

```
public Semaphore(int initialCount, int maximumCount);
public Semaphore(int initialCount, int maximumCount, string name);
public Semaphore(
    int initialCount,
    int maximumCount,
    string name,
    out bool createdNew
);
public Semaphore(
    int initialCount,
    int maximumCount,
    string name,
    out bool createdNew,
    SemaphoreSecurity semaphoreSecurity
);

public static OpenExisting(string name);
public static OpenExisting(string name, SemaphoreRights rights);
```

When you create a new semaphore object, you must always specify an initial and maximum count. In the `CreateSemaphore` APIs, this is accomplished with `lInitialCount` and `lMaximumCount`, respectively, while `Semaphore`'s constructors offer `initialCount` and `maximumCount` parameters. As noted in the introduction to this section, a semaphore is signaled so long as its current count is non-0. The initial count given is the semaphore object's current count once it has been created, and the maximum count will ensure any attempts to increment the semaphore's count above the maximum number will fail. (The maximum is inclusive: that is, it is legal for a semaphore to take on the value of its maximum.) For obvious reasons, the initial count may not be greater than the maximum.

As with mutex objects, if you try to create a new semaphore with the same name as an existing semaphore kernel object on the machine, the resulting reference will refer to the existing semaphore rather than a new one. In such a case, `GetLastError` will return `ERROR_ALREADY_EXISTS` for `CreateSemaphore` or `CreateSemaphoreEx`, and the `createdNew` output parameter for the managed `Semaphore`'s constructor will be set to false. This situation is not nearly as important to check for as with mutexes because the calling thread doesn't "own" the semaphore, but it does mean the specified counts will have been ignored. This may or may not be a problem for your code; it depends on the situation.

Taking and Releasing Semaphores

To “take 1” from the semaphore, in other words to decrement the semaphore’s count by 1, you wait on it using one of the mechanisms seen earlier: in other words, `WaitForSingleObject`, `WaitForMultipleObjects`, and so forth, or `WaitHandle.WaitOne`, `WaitAny`, or `WaitAll`. As noted earlier, semaphores do not rely on thread affinity. Thus, when the wait is satisfied, the count will have been decremented by 1, but there is no residual evidence that the calling thread was actually the one to decrement the count. If the thread is meant to do something meaningful, and then put back the count it took from the semaphore, it is imperative that the thread doesn’t crash before finishing. Because there is no thread affinity, there is no concept of an “abandoned semaphore” either; such corruption could lead to hangs, data integrity problems, and so on. Moreover, there is no concept of recursion, as there is with mutexes, because each wait will decrement from the semaphore’s current count. It is also not possible to take more than 1 from the count at once.

To “release 1” back to the semaphore in Win32—in other words to increment its count—you use the `ReleaseSemaphore` API. Because semaphores have no notion of owners (as mutexes do), there isn’t any restriction on what threads are permitted to increment the semaphore’s count. In fact, it’s common to have schemes where one thread is taking and another thread is releasing to the same semaphore, as we see later. The `ReleaseSemaphore` function takes an argument, `lReleaseCount`, which specifies a nonnegative number representing by what delta to increment the semaphores count. Unlike taking, which only allows you to take one count at a time when a wait is issued, releasing the semaphore can increment the count by an arbitrary number with the `lReleaseCount` parameter.

```
BOOL WINAPI ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    LPLONG lpPreviousCount
);
```

The `lpPreviousCount` argument can either be `NULL` or a pointer to a `LONG`, in which case the value of the semaphore’s count (before the increment) is stored into the location. The call to `ReleaseSemaphore` returns `TRUE` if the

increment succeeded and FALSE otherwise. If the current count plus the value of lReleaseCount would have caused the semaphore's count to exceed its maximum, the return value will be FALSE and GetLastError will return ERROR_TOO_MANY_POSTS. In this case, the semaphore's count will not have been modified, and lpPreviousCount will not contain any information about its current count.

In the case of managed code, you use the `Release` instance method on the `Semaphore` type to put back into the semaphore. There are two overloads.

```
public int Release();
public int Release(int releaseCount);
```

The no argument overload releases only one back to the semaphore, while the other allows you to pass in a nonnegative count as the `releaseCount` argument. Both overloads return the semaphore's count to what it was just prior to the release operation. If the release would have caused the semaphore's current count to exceed its maximum, a `SemaphoreFullException` is thrown and the semaphore's state will not be modified.

A Mutex/Semaphore Example: Blocking/Bounded Queue

Let's see an example of a queue data structure built using a single mutex and two semaphores. The semantics we want are that attempting to dequeue from an empty queue will block until data becomes available (i.e., a producer enqueues data), and attempting to enqueue into a full queue will block until space becomes available (i.e., a consumer dequeues data). This is a standard **blocking/bounded queue data structure**, and we'll look at some additional ways to implement it in Chapter 12, Parallel Containers. The mutex is used to achieve mutual exclusion so that state modifications are done safely, and the semaphores are used for control synchronization purposes. The semaphore makes this task relatively easy because protecting access to resources that are finite in capacity is the semaphore's purpose.

It's worth stating that there are many more efficient ways to implement this code. Depending on how much the production and consumption of items costs, the kernel transition overheads required to manipulate the

mutex and semaphore objects could quickly dominate you're resulting performance. In any case, this simple example will help to illustrate the behavior of these objects.

Here is an implementation of these ideas in C#.

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

public class BlockingBoundedQueue<T>
{
    private Queue<T> m_queue = new Queue<T>();
    private Mutex m_mutex = new Mutex();
    private Semaphore m_producerSemaphore;
    private Semaphore m_consumerSemaphore;

    public BlockingBoundedQueue(int capacity)
    {
        m_producerSemaphore = new Semaphore(capacity, capacity);
        m_consumerSemaphore = new Semaphore(0, capacity);
    }

    public void Enqueue(T obj)
    {
        // Ensure the buffer hasn't become full yet. If it has, we will
        // be blocked until a consumer takes an item.
        m_producerSemaphore.WaitOne();

        // Now enter the critical region and insert into our queue.
        m_mutex.WaitOne();
        try
        {
            m_queue.Enqueue(obj);
        }
        finally
        {
            m_mutex.ReleaseMutex();
        }

        // Note that an item is available, possibly waking a consumer.
        m_consumerSemaphore.Release();
    }

    public T Dequeue()
    {
        // This call will block if the queue is empty.
        m_consumerSemaphore.WaitOne();
```

```
// Dequeue the item from within our critical region.  
T value;  
m_mutex.WaitOne();  
try  
{  
    value = m_queue.Dequeue();  
}  
finally  
{  
    m_mutex.ReleaseMutex();  
}  
  
// Note that we took an item, possibly waking producers.  
m_producerSemaphore.Release();  
  
return value;  
}  
}
```

We used two semaphores for this example. The producer takes from one of them, which we'll call the producer semaphore, before acquiring the mutex and enqueueing an item. This is initialized to whatever the queue's capacity should be in the constructor. This semaphore achieves the effect of blocking the producer once the queue becomes full and happens inside of Enqueue. A consumer must release this semaphore after it has taken an item, inside of Dequeue, indicating to the producer that space has become available for it to enqueue a new item, in case it has reached 0. The second semaphore, which we'll call the consumer semaphore, is taken from by the consumer before dequeuing an element inside of Dequeue. This one's count corresponds to the number of items in the queue, and so it is initialized to 0 at the start. When the queue is empty, the consumer will block on it; the producer releases this semaphore after adding a new item to indicate to consumers that the queue is no longer empty. We use the mutex in both Enqueue and Dequeue to ensure that modifications to the underlying Queue<T> object are done in a thread safe manner.

Auto- and Manual-Reset Events

Windows provides two special event object types to facilitate coordination between threads: auto-reset and manual-reset events. (You'll sometimes hear these kernel object types referred to as synchronization and notification events, respectively, inside the Windows kernel and in device driver

programming.) An event object, like any other kernel object, is always in either the signaled or nonsignaled state. In usual event terminology, these states map to set and reset, respectively. I'll use the kernel object terminology in subsequent chapters when referring to events abstractly I'll typically prefer to use the terms set and reset.

To summarize the differences between the two event types: when an auto-reset has been signaled, only one thread will see this particular signal. When a thread observes the signal by waiting on the event, it is automatically transitioned back to the nonsignaled state. In this sense, an auto-reset event is like a mutex, with the sole difference being that auto-reset events have no notion of ownership and, hence, do not use thread affinity or recursion. This means that any thread can subsequently set the event, unlike a mutex, which requires that only the owner thread release it. If there are waiting threads when the auto-reset event transitions into a signaled state, Windows will select the first thread in the waiter queue to wake and will only wake up a single thread. All of the previous information about fairness and FIFO ordering applies. If there are no waiting threads at the time the signal arrives, then the first subsequent thread to wait on the object will return right away without blocking, atomically transitioning the event to a nonsignaled state. The manual-reset event, on the other hand, remains signaled until it is manually reset with an API call. In other words, the event is "sticky" and persistent (just like a traditional latch). This allows multiple threads to wait on the same event and observe the same signal, which is often useful for one-time events. All waiting threads are released at the time of a set.

As with mutex kernel objects, Win32 APIs are available to create and interact with these objects through their `HANDLEs`, and the .NET Framework exposes their capabilities through the `AutoResetEvent` and `ManualResetEvent` classes, joined at the hip by the common (concrete) base class, `System.Threading.EventWaitHandle`. `EventWaitHandle` is a subclass of the abstract base class `WaitHandle`. You work with instances of the two separate events types with basically the same set of APIs—to create, open, set, reset, and wait on the event—although there are some substantial differences regarding how the separate object types respond to signals and waiting. Note that the two subclasses of `EventWaitHandle` are only there as a convenience: you can instantiate and deal with `EventWaitHandle` objects directly if you prefer, as we'll see below.

Creating and Opening Events

Creating and opening events is identical to what we've already reviewed for semaphores and mutexes. Like semaphores, we will review just the details specific to events in this section. To create a new event object, or to find an existing one by name, you can use the `CreateEvent`, `CreateEventEx`, and `OpenEvent` APIs.

```
HANDLE WINAPI CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    BOOL bManualReset,
    BOOL bInitialState,
    LPCTSTR lpName
);
HANDLE WINAPI CreateEventEx(
    LPSECURITY_ATTRIBUTES lpEventAttributes,
    LPCTSTR lpName,
    DWORD dwFlags,
    DWORD dwDesiredAccess
);
HANDLE WINAPI OpenEvent(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpName
);
```

In the case of `CreateEvent`, the `bManualReset` argument specifies whether an auto-reset (FALSE) or manual-reset (TRUE) event should be created. `CreateEventEx` (new to Windows Vista) uses the `dwFlags` bit flags argument to specify this same information: if the argument value contains `CREATE_EVENT_MANUAL_RESET`, the event will be a manual-reset, and otherwise it will be auto-reset. This is the only valid flag that you can pass inside of `dwFlags`. The `bInitialState` argument specifies whether the event should be created in the signaled (TRUE) or nonsignaled (FALSE) state. The other parameters should be familiar by now: `lpEventAttributes` for optional access control, `lpName` to optionally name the object, and `dwDesiredAccess` to specify the resulting `HANDLE`'s access rights, new to Windows Vista. And `OpenEvent` works the same way that `OpenMutex`, and so on do.

To create an event in managed code, you have an option. An option is to instantiate one of the two derived classes `AutoResetEvent` and `ManualResetEvent`. Each has only a single constructor available.

```
public AutoResetEvent(bool initialState);
public ManualResetEvent(bool initialState);
```

Or you can instantiate an instance of the common base class `EventWaitHandle` via one of its several constructors, specifying either `EventResetMode.AutoResetEvent` or `ManualResetEvent` as the `mode` argument to indicate which kind of event you would like.

```
public EventWaitHandle(
    bool initialState,
    EventResetMode mode
);
public EventWaitHandle(
    bool initialState,
    EventResetMode mode,
    string name
);
public EventWaitHandle(
    bool initialState,
    EventResetMode mode,
    string name,
    out bool createdNew
);
public EventWaitHandle(
    bool initialState,
    EventResetMode mode,
    string name,
    out bool createdNew,
    EventWaitHandleSecurity eventSecurity
);
```

The simplest constructor overload accepts just the `initialState` argument, to specify whether the resulting event will be nonsignaled (`false`) or signaled (`true`) by default, and the `mode`, as described previously. The rest works the same way as the other kernel object types. The `name` parameter allows you to name the event so it can be subsequently looked up and shared, `eventSecurity` allows you to supply the security attributes for the created object, and the output parameter `createdNew` is set to `false` if an event already existed under the given name. The only reason to use `EventWaitHandle` directly is when you need to name the object or specify security attributes, since the `AutoResetEvent` and `ManualResetEvent` types don't support them. Using the more specific types has the advantage that you can see from a variable's type what kind of event is being used, whereas you

need to know where an `EventWaitHandle` was constructed to determine this (i.e., the mode isn't accessible via a property or anything similar).

Opening an existing event by name can be done with `EventWaitHandle`'s static `OpenExisting` method.

```
public static EventWaitHandle OpenExisting(string name);
public static EventWaitHandle OpenExisting(
    string name,
    EventWaitHandleRights rights
);
```

There's one slight glitch possible when you use named events. If the event already exists by name, then returned `HANDLE` from `CreateEvent` or `CreateEventEx` will point to the existing event rather than a new one. `GetLastError` will return `ERROR_ALREADY_EXISTS`, as with the other object types. Similarly, the `EventWaitHandle` constructor will set `createdNew` to `false`. The state of the event may not necessarily be in the state requested. It gets worse; there is no guarantee that the event returned is even the right kind. For example, if you requested a manual-reset event, but an auto-reset event was found under the same name, then the resulting reference will point at an auto-reset event. This can subsequently lead to errors and deadlocks.

Setting and Resetting Events

Events are signaled explicitly with the `SetEvent` Win32 API and can be reset to nonsignaled with `ResetEvent`.

```
BOOL WINAPI SetEvent(HANDLE hEvent);
BOOL WINAPI ResetEvent(HANDLE hEvent);
```

In managed code, you use the `EventWaitHandle`.`Set` and `Reset` instance methods.

```
public bool Set();
public bool Reset();
```

Setting the event transitions it to the signaled state, while resetting the event transitions it to the nonsignaled state, with the effects mentioned earlier depending on the kind of event. Unlike other kernel types such as mutexes and semaphores, an auto-reset event can be set multiple times

with no effect. Redundant calls to set the event when it's already signaled are effectively ignored. The Win32 APIs can fail, in which case they return FALSE and GetLastError retrieves the error information. Although the .NET Framework APIs are typed as returning bools, it's an anomaly: all failures are communicated through exceptions.

There is also a Win32 PulseEvent API that is deprecated and should not be used in new code. There is no support for it in managed code. A pulse is equivalent to a SetEvent immediately followed with a ResetEvent. In the case of a manual-reset event, any threads waiting at the time of the pulse are released; for an auto-reset event, at most one thread that is waiting when the event is pulsed will be released. PulseEvent is unreliable because threads often momentarily wake up and then rewait for many reasons on Windows. As we saw with user-mode APCs earlier, it's not uncommon for a thread to exit its wait only to reenter it after a tiny window of time during which it runs an APC. If a thread wakes up for such an event just prior to the pulse, the pulsed event will possibly return back to a nonsignaled state before the thread has a chance to rewait on the event. This consistently leads to problems, most often manifesting as deadlocks. For these reasons, you should avoid the API altogether. The only reason it is brought up in this book is to help you debug and maintain legacy code that uses it. And perhaps now you'll rewrite the next such piece of code you run across to use a more reliable mechanism.

Wait-All and Auto-Reset Events

The wait-all style of wait, specified with the WAIT_ALL flags value for the Win32 wait APIs or WaitHandle.WaitAll in managed code, interoperates closely with the object signaling mechanisms in the kernel. One might imagine that this was implemented as a loop that waits individually for each event, returning once each has been signaled, but this is not really how it works. The reason is subtle. In the case of auto-reset events, this naïve design would consume auto-reset event signals before all of the events had been signaled; not only would this possibly starve other threads that are prepared to process some subset of them, but should a thread time out before all of the events have been signaled, it must ensure none of them are consumed. To achieve this behavior, Windows ensures that no events are consumed until all events being waited for are in a signaled state, and only

then are they all consumed atomically. This also means that, although each event may become signaled during the wait, if they aren't ever never all signaled at any one time, the waiting thread will never actually wake up.

Events and Priority Boosts

A thread waiting on a Windows event enjoys a temporary priority boost of +1 when the wait is satisfied. This is often good because it helps to ensure threads that have been waiting are given preference to run. This is particularly important in responsive scenarios where the signaling of an event means a thread needs to process some information, possibly to update a GUI. Boosting can, however, also negatively impact scalability for some relatively common scenarios. If the waiting and setting threads are at the same priority and there are fewer CPUs than runnable threads, then it is possible that the act of setting an event will boost the waiting thread so that it immediately preempts and overtakes the setting thread. On single-CPU machines, in fact, this is guaranteed when the setter and waiter threads are of equal priority. This is perhaps fine, unless the thread setting the event holds on to resource that the waiting thread will need—such as a lock. In this case, the waiting thread will wake up in response to the event, get boosted so it preempts the setting thread, and find out immediately that it must wait again. The setting thread will then need to be rescheduled so that it can release the lock. This may again cause the waiting thread to be boosted (since most locks use events internally). And clearly this problem may actually repeat if the setting thread still owns resources the waking thread needs.

Here is a graphic illustration of this scenario.

Why is this so bad? Each context switch costs thousands of cycles. So when this situation happens, there are at least three context switches involved instead of one: (1) for the waking thread to overtake the setting

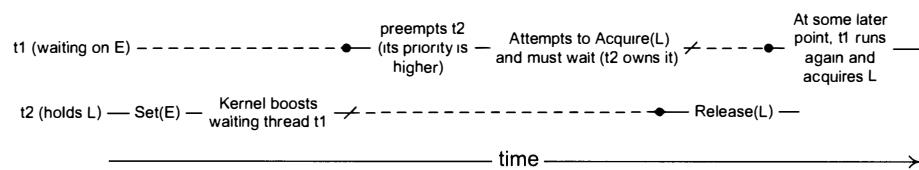


FIGURE 5.1: Timeline illustration of priority boosts in action

thread, (2) for the waking thread to go back to sleep and the setting thread to be resumed, and (3) for the waking thread to finally wake up and make forward progress. These unnecessary context switches are simply wasted cycles that could have been used to execute actual application logic. Wasted cycles are bad.

The following code example demonstrates this phenomenon in code.

```
ManualResetEvent mre = new ManualResetEvent(false);
object lockObj = new object();

Thread t1 = new Thread(delegate()
{
    Console.WriteLine("t1: waiting");
    mre.WaitOne();

    Console.WriteLine("t1: woke up, acquiring lock");
    lock (lockObj)
        Console.WriteLine("t1: acquired lock");
});

t1.Start();
Thread.Sleep(1000); // Allow 't1' to get scheduled

lock (lockObj) {
    Console.WriteLine("t2: setting");
    mre.Set();
    Console.WriteLine("t2: done w/ set, leaving lock");
}

t1.Join();
```

Thread t1 just waits on the event, and thread t2 sets the event while it still holds a lock that t1 will try to acquire as soon as it wakes up. Running this program on a single CPU machine consistently shows that t1 and t2 briefly ping-pong between each other once the event is set.

```
t1: waiting
t2: setting
t1: woke up, acquiring lock
t2: done w/ set, leaving lock
t1: acquired lock
```

Fixing these problems is not straightforward. In general, we'd prefer to avoid boosting the waking thread until all of the resources it needs to run are available. Using wait-all to acquire all such resources at once is

sometimes an option, but doesn't work for cases in which access to the raw kernel object is not permitted (as is the case with CLR monitors). Waiting to signal the event until such resources have been released is often an attractive solution, but it often comes with additional baggage because it opens you up to various race conditions. We'll become more familiar with such issues as we look at how to build event-based blocking queues later in this chapter. We discuss that when we get to the `SignalObjectAndWait` API, since an understanding of this API is required to build the queue.

Waitable Timers

The last kernel object type we'll look at in this chapter is the waitable timer. It's fairly common that a thread needs to wait for a certain period of time, or until a specific date or time has arrived. You can get by with sleeping—as we saw in the previous chapter—but Windows offers first-class kernel support for this. As its name implies, the waitable timer object allows a thread to wait and be awakened at a later date/time and optionally on a periodic recurring interval after that. So, for example, a thread can sleep until 7/31/2009 and then be awakened on an hourly basis afterwards. When a timer becomes signaled, we say that it has “expired.” Timers support both manual- and auto-reset modes, just as events do. A manual-reset timer allows multiple threads to wait on it and must be reset by hand, while an auto-reset timer wakes up only one waiting thread and automatically (and atomically) resets back to the nonsignaled state after releasing a single thread. A timer with a recurrence interval will then become signaled again the next time it expires.

The Win32 and .NET Framework thread pools offer support for timers to make it easier to manage waiting threads, timer expirations, and so on. This is useful because you typically don't want to require one thread per timer object. One solution to this problem is to use wait-any style waits so that a single thread can wait for many timers. But when a timer expires, you also probably don't want to hold up observing expirations for other timers that the thread is responsible for waiting on, so you might want to queue the work to some set of threads whose sole responsibility is to execute callbacks in response to timer expirations. There are other optimizations that come up too, like reducing the number of waits by clumping timer

expirations together, and so on. The thread pools handle all of this, as we describe in Chapter 7, Thread Pools. Although knowing about the kernel waitable timer support is useful, most programmers will want to use the thread pools instead.

Also note that the .NET Framework doesn't offer direct support for waitable timers. It uses them in the implementation of its thread pool timer support (exposed through the `System.Threading.Timer` object), but does not expose any public APIs to work directly with the kernel object itself. Therefore, everything we are about to see applies only to native code.

Creating and Opening Timers

As with the other kinds of kernel objects we've already looked at, there are a set of create functions to generate a new timer object and a function to open an existing timer.

```
HANDLE WINAPI CreateWaitableTimer(
    LPSECURITY_ATTRIBUTES lpTimerAttributes,
    BOOL bManualReset,
    LPCTSTR lpTimerName
);
HANDLE WINAPI CreateWaitableTimerEx(
    LPSECURITY_ATTRIBUTES lpTimerAttributes,
    LPCTSTR lpTimerName,
    DWORD dwFlags,
    DWORD dwDesiredAccess
);
HANDLE WINAPI OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    LPCTSTR lpTimerName
);
```

When creating a new timer with `CreateWaitableTimer`, the `bManualReset` argument specifies whether the timer is auto-reset (FALSE) or manual-reset (TRUE). This is specified with the `CreateWaitableTimerEx` API (new to Vista) by passing `CREATE_WAITABLE_TIMER_MANUAL_RESET` in the `dwFlags` argument; its presence results in a manual-reset event, else it is auto-reset. The `lpTimerAttributes` parameter is used to specify access control on the object, and `lpTimerName` can be used to optionally name a timer. If an existing timer with the provided name exists, the `HANDLE` will refer to it and `GetLastError` returns

ERROR_ALREADY_EXISTS. OpenWaitableTimer works just like the other open APIs we reviewed previously.

Setting and Waiting

We have said nothing about the expiration period when creating a new timer object. The result is that, even after creating the timer object, no timer has been scheduled for execution. You do that with the SetWaitableTimer function.

```
BOOL WINAPI SetWaitableTimer(
    HANDLE hTimer,
    const LARGE_INTEGER * pDueTime,
    LONG lPeriod,
    PTIMERAPCROUTINE pfnCompletionRoutine,
    LPVOID lpArgToCompletionRoutine,
    BOOL fResume
);
```

Clearly, `hTimer` is the waitable timer object `HANDLE` returned from the create or open method for which a new expiration is to be set. The `pDueTime` and `lPeriod` arguments specify the timer's expiration policy; `pDueTime` points to a 64-bit `LARGE_INTEGER` structure, which must actually be a `FILETIME` structure. This allows you to specify an absolute date or relative offset at which the timer will first expire. But because it's a `FILETIME`, this requires additional background discussion, which we will get to soon. The `lPeriod` is just the number of milliseconds between timer expirations, beginning with the `pDueTime` date. It may be 0, in which case the timer will fire only once at `pDueTime`, that is, there will be no recurrence. The `fResume` argument may be set to `TRUE` if the timer should still fire if the system has transitioned into low-power mode or `FALSE` if the timer should not fire in this case.

You can call `SetWaitableTimer` on the same timer object multiple times. This enables you to change the next due date and recurrence of an existing timer and is the only way to reset a manual reset timer, that has already fired, back to nonsignaled. (Auto-reset timers automatically transition back to nonsignaled when a thread waits on one.) There is also a `CancelWaitableTimer` routine that just takes a `HANDLE` to a timer object and stops the timer from firing again in the future.

You may optionally supply `pfnCompletionRoutine` and `lpArgToCompletionRoutine` argument values, though often they are just `NULL`. If `pfnCompletionRoutine` is non-`NULL`, the APC will be queued onto the thread that originally called `SetWaitableTimer` when the timer expires. Once that thread issues an alertable wait, it will dispatch the timer APC function call(s) that have queued up. If an APC function is provided and the calling thread exits before the timer expires, the timer is canceled.

This function pointer refers to a function of the signature.

```
VOID CALLBACK TimerAPCProc(  
    LPVOID lpArgToCompletionRoutine,  
    DWORD dwTimerLowValue,  
    DWORD dwTimerHighValue  
) ;
```

As you probably guessed, the `lpArgToCompletionRoutine` parameter passed to `SetWaitableTimer` is passed through transparently to the APC routine. The `dwTimerLowValue` and `dwTimerHighValue` arguments to the APC routine correspond to the fields of a `FILETIME` structure representing the time at which the timer became signaled.

A Brief Tangent on Using FILETIMEs. Now let's conclude our discussion of waitable timers with a look at how to go about specifying the `pDueTimer` argument. If you're already familiar with `FILETIMEs`, feel free to skip ahead to the next section. Most Win32 programmers are used to specifying timeouts and various synchronization-related times with millisecond based `DWORD` values representing relative offsets from the current time. But `SetWaitableTimer` (and, as we'll see in Chapter 7, Thread Pools, various Windows thread pool APIs) deal in terms of `FILETIMEs` instead. This is done for two reasons: `FILETIMEs` allow you to specify absolute dates, and relative `DWORD` milliseconds don't; this is how Windows implements waits and timeouts throughout the kernel, so using `FILETIMEs` directly saves some translation overhead.

A `FILETIME` is a 64-bit structure comprised of two `DWORDs`, a high and low date. Together these encode the number of 100 nanosecond units of time elapsed since 1/1/1601.

```
typedef struct _FILETIME {  
    DWORD dwLowDateTime;  
    DWORD dwHighDateTime;  
} FILETIME, * PFILETIME;
```

Notice that `SetWaitableTimer` takes a pointer to a `LARGE_INTEGER` (a.k.a. `_int64`, `LONGLONG`, `LONG64`, and so forth) and not an actual `FILETIME`. It's not safe to simply cast a `FILETIME *` to a `LARGE_INTEGER *`. The reason is subtle. `FILETIME`s consist of two separate 32-bit values; therefore, the start of the `FILETIME` structure itself is not required to be aligned on an 8-byte boundary. But `LARGE_INTEGER` offers the `QuadPart` field, which is a true 64-bit value, and thus its start needs to be aligned on an 8-byte boundary. Casting a `FILETIME *` to a `LARGE_INTEGER *` may create a misaligned pointer and will cause exceptions when dereferenced on platforms that require alignment, such as IA64. (Note that the reverse is OK—that is, casting a `LARGE_INTEGER *` to a `FILETIME *`.) Worse, if you're not actively testing on such platforms today, you'll be creating some nasty portability issues with your code in the future, possibly without even knowing it.

There are a few techniques to get around this issue. In many cases, we will be setting fields of the structure individually, in which case it's easiest to start with a `LARGE_INTEGER`. Like `FILETIME`, `LARGE_INTEGER` offers two individual 32-bit fields, `LowPart` and `HighPart`, to set the parts independently; or you can set the `QuadPart` value directly if you want to store all 64 bits at once. You can also either copy bytes from the `FILETIME` structure to a separate `LARGE_INTEGER` via `memcpy` or, alternatively, you can use the VC++ alignment compiler directive, that is, `__declspec(align(8))`, on the `FILETIME` variable to guarantee alignment, in which case it's safe to perform the cast.

It would be nice if the internal representation of `FILETIME` was an implementation detail, but you will have to munge it in order to use waitable timers (and other APIs in the thread pool, including timer callbacks and registered waits). What's worse, there are no easy-to-use system APIs that create relative-offset `FILETIME` values from existing absolute-offset `FILETIME`s, so we'll have to do a little hacking to create the right values.

Let's tackle the simple case, where you want the timer to begin executing right away. Just initialize your `LARGE_INTEGER` to 0.

```
LARGE_INTEGER li = {0L};  
SetWaitableTimer(..., &li, ...);
```

You could instead initialize a FILETIME's fields to 0, but that requires the extra steps mentioned above to copy bits around or to align the data structure:

```
_declspec(align(8)) FILETIME ft = {0,0};  
SetWaitableTimer(..., reinterpret_cast<LARGE_INTEGER *>(&ft), ...);
```

Both work roughly equivalently. The timer begins firing right away.

As mentioned earlier, you can specify either an absolute or a relative value for the due time. To represent an absolute date in the future, you'll have to construct a FILETIME with a valid representation of the date you desire. Because the structure's encoding is an implementation detail, you'll want to consult other system APIs to create one. You can grab a FILETIME off of a file, for example, by accessing its creation date, but that's probably not going to be useful (given that it has probably been created sometime in the past). The easiest way to get started is to use a SYSTEMTIME, set its fields as appropriate, and then convert it to a FILETIME with the SystemTimeToFileTime API.

```
typedef struct _SYSTEMTIME  
{  
    WORD wYear;  
    WORD wMonth;  
    WORD wDayOfWeek;  
    WORD wDay;  
    WORD wHour;  
    WORD wMinute;  
    WORD wSecond;  
    WORD wMilliseconds;  
} SYSTEMTIME, * PSYSTEMTIME;  
  
BOOL SystemTimeToFileTime(  
    const SYSTEMTIME * lpSystemTime,  
    LPFILETIME lpFileTime  

```

As a simple example, say we wanted to schedule a timer to fire at midnight on 5/6/2027. We could do that as follows.

```

SYSTEMTIME st = {0};
ZeroMemory(&st, sizeof(SYSTEMTIME));
st.wYear = 2027;
st.wMonth = 5;
st.wDay = 6;

__declspec(align(8)) FILETIME ft;
SystemTimeToFileTime(&st, &ft);

SetWaitableTimer(..., reinterpret_cast<LARGE_INTEGER *>(&ft), ...);

```

Alternatively, you could use the `GetSystemTime` API to obtain an already initialized `SYSTEMTIME` set to the current date and time, manipulate it as needed by adding offsets, and then use `SystemTimeToFileTime` to convert it into a `FILETIME`.

```
void GetSystemTime(LPSYSTEMTIME lpSystemTime);
```

However, manipulating `SYSTEMTIME`s with arithmetic is tricky because you have to handle the plethora of date/time validation corner cases, such as knowing how many days are in a particular month and so on. That brings us to the discussion of how to specify relative times.

If the value provided is negative, it is interpreted as a relative (nonnegative) number of 100 nanosecond units from the current time. How do you go about getting a negative `LARGE_INTEGER`? That's simple. You can set its `QuadPart` to a negative value. Since most people are used to specifying relative offsets in milliseconds quantities, we'll do the same. We must first convert milliseconds to 100 nanosecond units, which we do by multiplying milliseconds by 1,000 (to get microseconds) and then multiplying that by 10 (to get 100 nanoseconds):

```

DWORD milliseconds = ...;
LARGE_INTEGER li = { -((LONG64)milliseconds * 1000 * 10) };
SetWaitableTimer(..., &li, ...);

```

You could also initialize a `FILETIME` structure similarly, though it takes a little extra effort. (This is mentioned here because some related thread pool APIs use `FILETIME`s instead of `LARGE_INTEGER`s, as we will see in Chapter 7, Thread Pools.) You can probably figure it out based on an understanding of the binary representation of two's compliment numbers: if the most significant bit in `dwHighDateTime` is turned on, then the number is

considered to be negative, and the rest of the number must be specified in two's compliment representation.

Unless you enjoy thinking about binary representation in your code, the easiest approach to getting a negative value into a FILETIME structure is to use a 64-bit data type and copy by hand the high and low bits back into the FILETIME's dwHighDateTime and dwLowDateTime parts, respectively. Here is a simple function that does all of the bit-blitting for us. It takes a pointer to a FILETIME and number of milliseconds, specified as a DWORD, and initializes the FILETIME's fields

```
void InitFileTimeWithMs(PFILETIME pft, DWORD dwMilliseconds)
{
    LARGE_INTEGER cv;
    cv.QuadPart = -((LONG64)dwMilliseconds * 1000 * 10);
    pft->dwLowDateTime = cv.LowPart;
    pft->dwHighDateTime = cv.HighPart;
}
```

Signaling an Object and Waiting Atomically

Recall Table 5.1 from earlier in this chapter that some kernel objects are signaled only by the kernel—such as the process and thread objects—and that programs have little direct control over transitions between the signaled and nonsignaled states. Many other objects, such as those meant for synchronization, require you to manually trigger the transitions using object specific and wait APIs. `SignalObjectAndWait` is alternative way to signal these kinds of objects directly.

```
DWORD WINAPI SignalObjectAndWait(
    HANDLE hObjectToSignal,
    HANDLE hObjectToWaitOn,
    DWORD dwMilliseconds,
    BOOL bAlertable
);
```

This API accommodates situations in which you must signal an object and begin waiting for another one atomically. Although this isn't overly common, it's not rare either: there are many interesting cases in which it's a requirement for avoiding missed wake-ups and corresponding deadlocks. We'll see such a case shortly. Condition variables offer first class

support for this pattern; we will return to this topic when we look at CLR monitors and Windows condition variables in Chapter 6, Data and Control Synchronization.

`SignalObjectAndWait` is available on Windows as of Windows NT 4.0 and, hence, cannot be used on Windows 9x, requiring `_WIN32_WINNT` to be defined as `0x0400` or higher. Calling this function has a similar effect as calling the corresponding object specific signal API on `hObjectToSignal`, that is, `ReleaseMutex` if it's a mutex, `ReleaseSemaphore` (with a count argument of 1) if it's a semaphore, or `SetEvent` if it's an event. (This is like calling the respective object's API once and only once. For mutexes that have been acquired recursively, for example, calling `SignalObjectAndWait` will decrement the recursion counter by one—it won't do the work needed to make the mutex completely available to other threads, and so it's not guaranteed to become signaled.) After signaling the object, the API then blocks until either `hObjectToWaitOn` becomes signaled, the timeout specified by `dwMilliseconds` is exceeded (if not `INFINITE`), or an APC is dispatched (if `bAlertable` is `TRUE`). The most interesting aspect of this function is that it appears as though the thread enters the wait state for `hObjectToWaitOn` before it signals `hObjectToSignal`, which you couldn't actually do on your own without help from the Windows kernel.

The return value is mostly the same as with the other wait functions described earlier: `WAIT_OBJECT_0` if the wait succeeds, `WAIT_TIMEOUT` if the specified timeout expires, `WAIT_ABANDONED` if `hObjectToWaitOn` is a handle to a mutex that has been abandoned, `WAIT_IO_COMPLETION` if an APC interrupts the wait, or `WAIT_FAILED` to indicate that the wait (or possibly signaling `hObjectToSignal`) has failed. There are some notable differences, however. With a couple of exceptions, the `hObjectToSignal` object will have been signaled, even if the wait failed, timeout expired, or an APC got dispatched. But sometimes a `WAIT_FAILED` return value indicates that signaling `hObjectToSignal` itself failed. You can check `GetLastError` for return codes ordinarily returned by the object specific signaling APIs to determine this. For instance, `GetLastError` will return `ERROR_TOO_MANY_POSTS` if `hObjectToSignal` was already full semaphore.

You must be very careful with error conditions. Because `hObjectToSignal` will have typically been signaled by the time an error is discovered (i.e., if it occurs while waiting on `hObjectToWaitOn`), then you can no longer achieve

the atomicity that was sought by using `SignalObjectAndWait` in the first place. This is a fundamental problem that recovering from often requires extra synchronization. It typically can't be handled as you would a normal wait, for example, subtracting time from the timeout and reissuing a `WaitForSingleObject` on `hObjectToWaitOn`. In some cases, you even have to turn around and rewait on `hObjectToSignal` so that you can reacquire it and proceed.

In managed code, there are three method overloads on the `WaitHandle` class that provide this same exact functionality.

```
public static bool SignalAndWait(
    WaitHandle toSignal,
    WaitHandle toWaitOn
);
public static bool SignalAndWait(
    WaitHandle toSignal,
    WaitHandle toWaitOn,
    int timeoutMilliseconds,
    bool exitContext
);
public static bool SignalAndWait(
    WaitHandle toSignal,
    WaitHandle toWaitOn,
    TimeSpan timeout,
    bool exitContext
);
```

These call the `SignalObjectAndWait` Win 32 function internally. If the timeout expires while waiting for the `toWaitOn` object, this method returns `false`. Error conditions and abandoned mutexes are represented the same way they are with the object specific APIs.

Unfortunately there is one known discrepancy: if the `toSignal` object represents a semaphore whose count has already reached its maximum, `SignalAndWait` throws an `InvalidOperationException` instead of the expected `SemaphoreFullException`. All of the other exception types are consistent with the kernel object specific methods.

A Motivating Example: A Blocking Queue Data Structure with Events

Let's look at an example where you might use events for coordination purposes and where the ability to signal and wait atomically comes in handy. Imagine we want to build a queue type that blocks when a consumer tries

to take from an empty queue. This is a standard blocking queue and is much like our example earlier that uses semaphores with the difference that we omit blocking producers when some fixed capacity has been reached. We will begin by building such a data structure out of an auto-reset event and then explore how to accomplish the same behavior with a manual-reset event. In both cases, we will use a mutex to guarantee thread safe access to state.

Using events rather than semaphores can lead to slightly more efficient code because it doesn't require as many context switches. This approach is substantially more complicated and error prone. We'll have to use the `SignalObjectAndWait` API to write a deadlock free version. The examples are written in C# to avoid things such as memory management, which distract from the core concurrency behavior we're interested in exploring. The ideas translate easily to C++.

With Auto-Reset Events. We use a single auto-reset event for this data structure. When a consumer notices the queue is empty, it will wait on the event. And whenever a producer creates a new item, it will signal the event so that a single waiting consumer wakes up and processes any items found in the queue. Here is some sample code that accomplishes this.

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

public class BlockingQueueWithAutoResetEvents<T>
{
    private Queue<T> m_queue = new Queue<T>();
    private Mutex m_mutex = new Mutex();
    private AutoResetEvent m_event = new AutoResetEvent(false);

    public void Enqueue(T obj)
    {
        // Enter the critical region and insert into our queue.
        m_mutex.WaitOne();
        try
        {
            m_queue.Enqueue(obj);
        }
        finally
        {
            m_mutex.ReleaseMutex();
        }
    }
}
```

```
}

// Note that an item is available, possibly waking a consumer.
m_event.Set();
}

public T Dequeue()
{
    // Dequeue the item from within our critical region.
    T value;
    bool taken = true;
    m_mutex.WaitOne();
    try
    {
        // If the queue is empty, we will need exit the
        // critical region and wait for the event to be set.
        while (m_queue.Count == 0)
        {
            taken = false;
            WaitHandle.SignalAndWait(m_mutex, m_event);
            m_mutex.WaitOne();
            taken = true;
        }

        value = m_queue.Dequeue();
    }
    finally
    {
        if (taken)
        {
            m_mutex.ReleaseMutex();
        }
    }
}

return value;
}
}
```

Most of this is straightforward. The consumer checks that `m_queue.Count != 0` before removing an item from the queue. If the queue is empty, the thread must wait for a producer to set the event. Clearly the consumer needs to exit the mutex before waiting, otherwise no producer would be able to enter its critical region and enqueue data. As soon as the consumer wakes up, it must acquire the mutex again. The check for the queue being empty is done in a loop because although the thread has awakened because a producer enqueued data, it is quite possible that another consumer will

call `Dequeue` in the meantime. This thread acquires the mutex before the awakened thread and dequeues the element. We must ensure in this case that the awakened thread sees that the queue is empty and goes back to waiting again.

We have to be careful to avoid deadlocks in this design. These might be caused by threads going to sleep and not being told properly that new items have arrived. (This problem, referred to as “lost wakeups,” is described at great length in Chapter 11, Concurrency Hazards; it is perhaps the most common control synchronization pitfall that people face.) To avoid deadlocks in this particular case, we must ensure that when an empty queue is noticed (while the mutex is still held), the consumer releases the mutex and waits on the event atomically, accomplished with the call to `WaitHandle.SignalAndWait`.

To illustrate better why this is necessary, imagine for a moment that the consumer replaced the `SignalObjectAndWait` call with two independent calls to `ReleaseMutex` and then `WaitForSingleObject` instead.

```
m_mutex.ReleaseMutex();
m_event.WaitOne();
```

All it takes is three threads, one producer and two consumers, and bad luck to encounter a deadlock due to a missed signal.

t0 (consumer)	t1 (consumer)	t2 (producer)
<code>ReleaseMutex(g_hMutex);</code>	<code>ReleaseMutex(g_hMutex);</code>	<code>SetEvent(g_hSyncEvent);</code>
<code>WaitForSingleObject(...);</code>	<code>WaitForSingleObject(...);</code>	<code>SetEvent(g_hSyncEvent);</code>
		<code>...</code>

Given this program schedule, either t0 or t1 is now doomed to (possibly) wait forever. Why? Because the producer set the event twice before any thread was waiting on the event, only one thread observed the fact that a new item has been published. Remember that an auto-reset can either be signaled or nonsignaled: there is no concept of multiple signals (as with a semaphore). Therefore, only one of the threads will see the event in a

signaled state when it eventually waits on it, even though the producer has set it multiple times. The consumers can't release the mutex after performing the wait because the wouldn't be able to enqueue new data, also causing a deadlock. Using `SignalObjectAndWait` in this case prevents deadlock prone schedules like this one. This is the main reason building this data structure out of events is trickier than building it with a semaphore.

There are still some issues with the `SignalObjectAndWait` approach to this problem, which we have touched on previously. Because the thread doing a wait may temporarily wake up due to an APC, it may not be in the wait queue when `SetEvent` is called, leading to the possibility of a missed event and an ensuing deadlock. This problem is similar to the `PulseEvent` problem mentioned earlier. For this reason, you must be very careful when using this pattern and should never pass `TRUE` for `bAlertable`.

In fact, this problem is lurking within this code as written. Because the CLR uses alertable waits internally while it executes the `SignalAndWait` and automatically reissues the wait, a consumer may be temporarily removed from the event's wait queue to execute an APC. Say there are two consumers and both have temporarily gone off and begun executing APCs. If two producers come along, there will be two calls to set the event. But only one of the consumers will observe this event when they return to waiting, which automatically transitions the event to a nonsignaled state, meaning the second consumer will miss the event. In native code, you can work around this issue by passing `FALSE` to `bAlertable` when calling `SignalObjectAndWait`. In managed code, however, there's not much you can do. As written, this code can cause deadlock under rare but certainly possible circumstances.

Some simple optimizations can be made in this example: if we keep a counter of the number of waiting consumers—that is, it is incremented under the protection of a mutex prior to waiting and decremented when it wakes up—then producers can avoid signaling the event when no threads are waiting, leading to fewer kernel transitions. As it stands, each producer

call incurs three transitions: one to acquire the mutex, one to signal the event, and one to release the mutex. With this optimization, it would be reduced to just two.

With Manual-Reset Events. Alternatively, we can use a manual-reset event to implement our queue. This can be more intuitive than using auto-reset events and also avoids the problem of lost wake-ups caused by APCs. Instead of notifying waiters each and every time a new item is produced, we will have two states for our queue: empty and nonempty. And then our single manual-reset event will be kept in synch with these states, that is, nonsignaled and signaled, respectively. Whenever a consumer sees an empty queue, it waits on the event. When a consumer takes the last item from the queue, it resets the event so that it is nonsignaled. And finally, when a producer adds an item to an empty queue, it sets the event (i.e., state transition empty to nonempty).

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

public class BlockingQueueWithManualResetEvents<T>
{
    private Queue<T> m_queue = new Queue<T>();
    private Mutex m_mutex = new Mutex();
    private ManualResetEvent m_event = new ManualResetEvent(false);

    public void Enqueue(T obj)
    {
        // Enter the critical region and insert into our queue.
        m_mutex.WaitOne();
        try
        {
            m_queue.Enqueue(obj);

            // If the queue was empty, the event should be
            // in a signaled set, possibly waking waiters.
            if (m_queue.Count == 1)
                m_event.Set();
        }
        finally
        {
            m_mutex.ReleaseMutex();
        }
    }
}
```

```
public T Dequeue()
{
    // Dequeue the item from within our critical region.
    T value;
    bool taken = true;
    m_mutex.WaitOne();
    try
    {
        // If the queue is empty, we will need exit the
        // critical region and wait for the event to be set.
        while (m_queue.Count == 0)
        {
            taken = false;
            m_mutex.ReleaseMutex();
            m_event.WaitOne();
            m_mutex.WaitOne();
            taken = true;
        }

        value = m_queue.Dequeue();

        // If we made the queue empty, set to non-signaled.
        if (m_queue.Count == 0)
            m_event.Reset();
    }
    finally
    {
        if (taken)
        {
            m_mutex.ReleaseMutex();
        }
    }
}

return value;
}
```

This example is strikingly similar to the first attempt above. We avoid setting the event unless the producer has just transitioned from an empty to a nonempty queue, which can provide some performance benefits. However, we now have to make the call to set the event inside the critical region, to avoid deadlocks caused by race conditions between producers and consumers. The consumer must also reset the event if it transitions the queue to empty. Notice that we didn't need to use the `SignalAndWait` API in the consumer, though we certainly could have. It's not necessary because manual-reset events are "sticky," and, thus, we will not miss any events.

This queue data structure will likely lead to fewer kernel transitions than the earlier auto-reset event version. For a queue that usually has items in it, the only kernel transitions required are those needed for the mutex acquisition and releases. The worst case, which is worse than the average case for the auto-reset event queue, is when the queue is constantly transitioning between empty and nonempty, since each operation requires a kernel transition. But even in this worst case situation, the number of transitions on enqueue and dequeue is equivalent to the number needed in the semaphore based queue that we built earlier in this chapter.

Debugging Kernel Objects

As our last topic having to do with kernel objects in this chapter, let's explore briefly how to debug kernel objects. Because kernel object state is kept in kernel-mode memory and because there aren't any user-mode APIs to find out what threads are waiting for a mutex or which thread currently owns it, you'll have to resort to a debugger like WinDbg for most of this information. WinDbg is of course extremely powerful, and, thus, we'll only scratch the surface of what you are able to do with it.

Perhaps the most useful debugger feature is the !handle command. If you have an object handle, you can dump detailed information about it with '!handle <handle> f'. In this command text, <handle> is the actual numeric handle for the thread, and f instructs the debugger to print detailed information about the object rather than just a summary. Here is an example of this command run against a manual-reset event whose handle is 0x7e8.

```
0:000> !handle 0x7e8 f
Handle 7e8
  Type          Event
  Attributes    0
  GrantedAccess 0x1f0003:
                Delete,ReadControl,WriteDac,WriteOwner,Synch
                QueryState,ModifyState
  HandleCount   2
  PointerCount  4
  Name          <none>
Object Specific Information
  Event Type Manual Reset
  Event is Waiting
```

Notice that everything leading up to the “Object Specific Information” section is general to all kernel object types. Dumping information about a mutex will contain information about whether it is currently owned, a semaphore will provide the current and maximum count for the object, and so on. WinDbg stops short of providing other useful information such as the threads that owns a particular mutex, what threads are waiting for which objects, and so forth because this information is stored inside kernel-mode data structures. You can use the Kernel Debugger, KD.EXE—which is provided with the same Debugging Tools for Windows package that contains WINDBG.EXE—to access this information.

To start a kernel debugging session for the local machine run KD.EXE /KL. Once inside, you can run the !process command to retrieve information about the process in which you are interested. Running '!process <handle> 2' will print out detailed information about each thread in the system, including what kernel object it is waiting on (if any). Moreover, if a thread is waiting on a mutex that is currently owned, that thread’s kernel memory location is shown. As an example, here is an entry for a thread waiting for a currently-owned mutex.

```
THREAD 80172040 Cid 10f0.20c8 Teb: 7efdd000 Win32Thread: 00000000
WAIT: (UserRequest) UserMode Alertable
      8306aa00 Mutant - owning thread 822240c8
```

In this example, thread that lives at memory location **80172040**, whose user-mode visible process ID is **10f0** and thread ID is **20c8** (separated by a dot in the “Cid”), has performed an alertable wait in user-mode on a mutex (a.k.a. mutant). This mutex is currently owned by the thread at **822240c8** and lives at address **8306aa00**. It’s often useful to do user- and kernel-mode debugging side by side for the same process because they both offer useful but different ways of accessing kernel object information.

Where Are We?

This chapter covered a fair bit of ground. In addition to offering services to create and schedule threads, as we saw in Chapters 3 and 4, the Windows kernel also offers support for synchronization between threads. What you’ve seen in this chapter—the ability to wait in a myriad of ways on any

kernel object, several kernel objects themselves (mutexes, semaphores, events, and waitable timers)—will be fundamental to all concurrent programs you encounter. Many services are layered on top of them. So even if you don't end up calling `CreateMutex` or `WaitForMultipleObjectsEx` directly, you are probably using them deep down in the implementation of whatever higher-level API you're coding against.

In that light, the next chapter will focus on some useful user-mode abstractions that are built on top of these kernel facilities. These APIs aim to make the more common synchronization patterns easier and often provide superior performance. Knowing all about these low-level kernel facilities will enable you to use them appropriately when the higher-level programming models don't quite meet your needs exactly. And let's face it, life is usually simpler when you know what's going on underneath it all, particularly when debugging and diagnosing problems.

FURTHER READING

- J. Beveridge, R. Wiener. *Multithreading Applications in Win32: The Complete Guide to Threads* (Addison-Wesley, 1997).
- D. Box. *Essential COM* (Addison-Wesley, 1998).
- K. Brown, T. Ewald, C. Sells, D. Box. *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications* (Addison-Wesley, 1999).
- K. Brown. *Programming Windows Security* (Addison-Wesley, 2000).
- J. M. Hart. *Windows System Programming*, Third Edition (Addison-Wesley, 2005).
- C. Petzold. *Programming Windows*, Fifth Edition (MS Press, 1998).
- J. Richter. *Programming Applications for Microsoft Windows* (MS Press, 1999).
- M. Russinovich, D. A. Solomon. *Microsoft Windows Internals: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000*, Fourth Edition (MS Press, 2004).

6

Data and Control Synchronization

In the last chapter, we saw that the Windows kernel intrinsically supports several kinds of synchronization through kernel objects. What wasn't emphasized, however, was that you seldom want to use kernel objects directly as your primary synchronization mechanism. The simplest reason for this is cost. They cost a lot in time due to the kernel transitions required to access and manipulate them, and in space due to the various auxiliary OS data structures that are required to manage instances, such as the process handle table, kernel memory, and so forth. At the same time, if your program must truly wait for some event of interest to occur, you ultimately have no choice but to use a kernel object in one form or another. Even so, it's usually preferable to use a higher level construct, which abstracts away the use and management of such kernel objects.

Win32 and the .NET Framework both offer mechanisms that perform this kind of abstraction, typically using lazy allocation techniques and, in some cases, pooling them to reuse a single kernel object among multiple instances of higher level concurrency abstractions over time. This approach leads to an appreciable reduction in space and time by deferring all allocations to the latest point possible and by amortizing kernel transitions by incurring them only when absolutely necessary. In addition to offering equivalent functionality with better performance, these platform abstractions also codify common

coding patterns that you would otherwise have to build by hand using only kernel objects such as shared-mode locks and first class condition variables.

Here is a list of the synchronization primitives we'll review in this chapter.

- Win32 **CRITICAL_SECTIONS** provide a more efficient mutual exclusion mechanism for native code when compared to mutexes. Roughly, they are equivalent in functionality to mutex kernel objects and support recursive acquires. Entering and leaving critical sections occurs entirely in user-mode except for the (rare, one hopes) cases where lock contention is encountered, in which case a true kernel object will be used to wait.
- CLR locks—accessed via the `Monitor` class's static `Enter`, `Exit`, and `TryEnter` methods, the C# `lock` keyword, or the VB `SyncLock` keyword—are effectively the managed equivalent to **CRITICAL_SECTIONS**. Each CLR object implicitly has a lock associated with it and can, therefore, stand in as a separate lock object. These are also lightweight, using a pointer sized header in the target object until contention is encountered, which, as with **CRITICAL_SECTIONS**, lazily allocates a kernel object. And even then, internal kernel objects are pooled and reused among many locks.
- Win32 “slim” reader/writer locks (i.e., SRWLocks) are new to Windows Vista and Server 2008 and offer both exclusive and shared lock modes, the latter of which can be used for read-only operations. Shared mode allows multiple threads performing reads to acquire the lock simultaneously. This is safe and usually leads to higher degrees of concurrency and, hence, better scalability. These are even lighter-weight to work with than **CRITICAL_SECTIONS**: in addition to executing almost entirely in user-mode, SRWLocks are the size of a pointer and do not even use standard kernel objects internally for waiting.
- There are two CLR reader/writer lock types: `ReaderWriterLock` and `ReaderWriterLockSlim`, both of which reside in the `System.Threading` namespace. The former dates back to version 1.1 of the .NET Framework, while the latter is new to 3.5 (i.e., Visual Studio 2008); the

new lock effectively deprecates the older one because it is lighter weight and addresses several design shortcomings of the older lock. This lock is still heavier weight than CLR locks and Vista's SRWL lock, however, because it is composed of multiple fields and uses a kernel object to wait.

- Win32 `CONDITION_VARIABLE`s are abstractions that support the classic notion of a condition variable. A condition variable allows one or more threads to wait for the occurrence of an event and integrates with both `CRITICAL_SECTIONS` and SRWLS, allowing you to atomically release a lock and begin waiting on a condition variable, thus eliminating tricky race conditions. These are new to Windows Vista and Server 2008. As with the SRWL, they are pointer-sized and do not use traditional kernel objects for waiting.
- CLR condition variables are exposed through `Monitor`'s `Wait`, `Pulse`, and `PulseAll` methods. Managed condition variables integrate with the CLR's mutually exclusive locking support exposed via `Monitor`, and, therefore, any managed object can be used as a condition variable too. As with the Vista condition variables, waiting will atomically release and wait on a monitor. Each condition variable reuses a kernel object associated with the managed thread and maintains a simple wait list and is, thus, very lightweight.

The remainder of this chapter will focus on the exploration of using these synchronization abstractions. Based on our taxonomy of data and control synchronization established in Chapter 2, Synchronization and Time, the first four primitives are for data synchronization, while the latter two are meant for control synchronization.

Mutual Exclusion

The most basic kind of data synchronization is mutual exclusion, where only one thread is permitted to be "inside" a critical region at a given time. This is exactly what the mutex kernel object offers. Let's turn our attention to two user-mode primitives that achieve a similar effect: Win32 critical sections and CLR locks, in that order. These are the most common

form of synchronization for concurrent native and managed programs, respectively.

Win32 Critical Sections

A critical section is a simple data structure (`CRITICAL_SECTION`, defined in `Windows.h`) that is used to build critical regions. (It's easy to get "critical section" confused with "critical region" given the similar names. While this isn't terrible, you should distinguish clearly between the abstract notion of a **critical region**—which is a code region in your program that enjoys mutual exclusion—and a **critical section**—which is a specific data structure used to implement critical regions.) Each critical section instance is local to a process, and multiple instances may be created; each section establishes a separate span of mutual exclusion, such that each distinct section is orthogonal to all others. In other words, a thread that has acquired critical section A does not in any way prevent another thread from acquiring an entirely separate critical section B. This is similar to how the acquisition and release of different mutex kernel objects does not interfere with one another.

When one thread has acquired ownership of a given section, no other thread is permitted to acquire that same section until it has been released. Attempts to do so result in the acquiring thread waiting for the section to become available, using a combination of spinning and an underlying auto-reset kernel object managed by the critical section. Critical sections are used in native code only. Because managed code often P/Invokes into or utilizes native code by way of mixed-mode assemblies, not to mention the CLR VM's direct use of native libraries, however, it's certainly possible for critical regions to be acquired and released on managed threads.

Allocating a Section

Critical sections are often statically associated with fragments of the program logic, in which case it is usually most convenient to allocate your `CRITICAL_SECTION` in the program's statically allocated memory. This corresponds nicely to coarse-grained locking, as per previous discussion. This usually means defining a C++ class static field or a global variable of type `CRITICAL_SECTION` and placing initialization logic into your program's startup logic or DLL's main function for library code. Such statically

allocated locks are typically used to protect large portions of the program, which are comprised primarily of static or global state. This corresponds to coarse-grained locking (see Chapter 2, Synchronization and Time).

In other cases, a critical section may be associated with a dynamically allocated data structure, such as a critical section per node in a tree data structure, in which case the `CRITICAL_SECTION` is typically allocated as a member inside the data structure's memory. In some cases, such a critical section is considered coarse-grained, for example, if it protects a larger collection of data, while in many cases dynamic allocation is used to produce finer-grained locks that are attached to individual bits of data. For example, if we had a tree data structure, we might allocate a single lock to protect all nodes, that is, coarse-grained locking; or we may wish to allow fine-grained locking of individual nodes by giving each its own critical section.

Notice that in neither example was the `CRITICAL_SECTION` object referred to by a pointer. This is common—that is, allocating the critical section “inline,” either in static or dynamic data—although you can alternatively allocate and free the `CRITICAL_SECTION` objects dynamically via `malloc`, `free`, `new`, and/or `delete`. This decision is entirely in your hands. The only hard requirement is that you never copy or attempt to move the critical region's memory after initialization. The implementation of critical sections assumes the address of the data structure remains constant and uses its address as the key into some internal OS data structures. Address movement can cause some undesirable things to happen to your program, ranging from crashes to data corruption.

When allocating a critical section embedded within a data structure, you might worry about the size of the section because it bloats the data structure. As of Windows Vista, a `CRITICAL_SECTION` object is 24 bytes on 32-bit architectures and 40 bytes on 64-bit systems. The variance is due to some internal pointer-sized information such as handles. The size is apt to change from release to release and even on different architectures, so you should certainly never depend on it. Nevertheless, it can at least be used as a guideline to help decide whether to use fine- or coarse-grained locks.

Initialization and Deletion

Because a critical region holds on to kernel resources internally and demands specific initialization and data layout, you must initialize each critical section

before it is first used. This is accomplished via the `InitializeCriticalSection` function or the `InitializeCriticalSectionAndSpinCount` function, which can be used to control the spin waits used by the section. There is also an `InitializeCriticalSectionEx` function that is new in Windows Vista. To avoid leaking resources, you must call the `DeleteCriticalSection` function once you no longer need to use the section. The signatures for these functions are as follows.

```
VOID WINAPI InitializeCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
VOID WINAPI InitializeCriticalSectionAndSpinCount(
    LPCRITICAL_SECTION lpCriticalSection,
    DWORD dwSpinCount
);
BOOL WINAPI InitializeCriticalSectionEx(
    LPCRITICAL_SECTION lpCriticalSection,
    DWORD dwSpinCount,
    DWORD Flags
);
VOID WINAPI DeleteCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection
);
```

Each takes a pointer to the memory location containing a `CRITICAL_SECTION` to initialize or delete. We'll discuss the `dwSpinCount` arguments for `InitializeCriticalSectionAndSpinCount` and `InitializeCriticalSectionEx` in more depth later in this section. The `Flags` argument to `InitializeCriticalSectionEx` can take on the value `CRITICAL_SECTION_NO_DEBUG_INFO`, which may be used to suppress the creation of internal debugging information. Note that you must take care to ensure that only one thread calls the initialization or deletion functions at any one time on any particular critical section and that the calling thread does so when no thread still owns the critical section object. Failing to heed this advice can lead to unexpected behavior. Initialization can fail with an `ERROR_OUT_OF_MEMORY` exception if the allocation of an internal auto-reset event did not succeed, although as of Windows 2000 the event is lazily allocated unless explicitly requested at initialization time. We dig into this topic momentarily.

When a critical section is allocated in the program's static memory, it is commonplace to do the initialization and deletion in the program's startup

and shutdown logic. For a reusable DLL this usually entails placing code in the library's DllMain function.

```
#include <windows.h>

CRITICAL_SECTION g_crst;

BOOL WINAPI DllMain(HINSTANCE hinstDLL,
                     DWORD fdwReason, LPVOID lpvReserved)
{
    switch (fdwReason)
    {
        case DLL_PROCESS_ATTACH:
            InitializeCriticalSection(&g_crst);
            break;
        case DLL_PROCESS_DETACH:
            DeleteCriticalSection(&g_crst);
            break;
    }
}
```

On the other hand, if the critical section is an instance member of a class, we might do this initialization and deletion from the constructor and destructor, respectively.

```
#include <windows.h>
class C
{
    CRITICAL_SECTION m_crst;
public:
    C()
    {
        InitializeCriticalSection(&m_crst);
    }
    ~C()
    {
        DeleteCriticalSection(&m_crst);
    }
};
```

Neither of these examples demonstrates any sort of error handling logic for situations in which initialization fails. A real program would have to deal with these conditions. But before discussing the specific kinds of failures that might be seen during initialization—since there's background and tangent information that we need to review, we'll first review the basics of entering and leaving critical sections.

Entering and Leaving

Once you have initialized a critical section, you are ready to use it to denote the boundaries of your critical regions using `EnterCriticalSection` and `LeaveCriticalSection`. As you'd expect, each of these functions also takes a `LPCRITICAL_SECTION` argument.

```
VOID WINAPI EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
VOID WINAPI LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
```

As soon as the `EnterCriticalSection` call returns, the current thread "owns" the critical section. This ownership is reflected in the state of the critical section object itself. If a call to `EnterCriticalSection` is made while another thread holds the section, the calling thread will wait for the section to become available. This wait may last for an indefinite amount of time, depending on the amount of time the owning thread holds the section. (There is a `TryEnterCriticalSection` API we'll review that avoids blocking during contention.) And the "wait" is optionally comprised of a bit of spin waiting (more on that later), which is then abandoned in favor of a true wait on an auto-reset event kernel object internally if the lock doesn't become available in a reasonable amount of time. Once the owning thread leaves the critical section, the waiting thread will either acquire the lock (if it is spinning) or be awakened (via the event signaling) and attempt to acquire the lock as soon as it has been scheduled. If many threads are waiting for a given critical section when it becomes available, the selection of the thread to wake is entirely based on the OS's quasi-FIFO auto-reset event wait list, as described more in Chapter 5, Windows Kernel Synchronization.

Although `EnterCriticalSection`'s signature appears to indicate that it cannot fail, as with `InitializeCriticalSection`, it may throw an `ERROR_OUT_OF_MEMORY` exception under some rare circumstances on Windows 2000 only. This is because the auto-reset event is usually lazily allocated upon its first use (as of Windows 2000), that is, the first time contention occurs on the lock, which can fail if the machine is low on resources. We'll describe why failure isn't possible on new OSs along with some historical perspective in a bit.

Critical sections support recursive acquires. That is to say, if the current thread holds the section when `EnterCriticalSection` is called, an internal recursion counter is incremented and the acquisition immediately succeeds. When `LeaveCriticalSection` is subsequently called, the recursion counter is decremented by 1; only when this counter reaches 0 is the section actually exited, made available to other threads, and any waiting threads awakened. Recursion is possible because the critical section tracks ownership information, enabling it to determine whether the calling thread is the current owner. While recursion may seem like a generally convenient feature, it does come with some unique challenges because it is very easy to accidentally recursively acquire a lock and depend (incorrectly) on certain state invariants holding. We review this issue more in Chapter 11, Concurrency Hazards.

Leaving an Unowned Critical Section. It is a very serious bug to try to leave a critical section that isn't owned by the current thread. In all cases, this indicates a programming error, and, if it ever occurs, there is no immediate indication that something has gone wrong. There is no error code or exception. Despite the appearance that all is well, a ticking time bomb has been left behind.

If the critical section is completely unowned at the time of the erroneous call to `LeaveCriticalSection`, all future calls to `EnterCriticalSection` will block forever. This effectively deadlocks all threads that later try to use this critical section. If the section is owned by another thread when the unowning thread tries to leave it, the current owner is still permitted to reacquire and release the lock recursively. But once the owner exits the lock completely, the lock has become permanently damaged: subsequent behavior is identical to the case where no owner was initially present. In other words, all subsequent calls to `EnterCriticalSection` by any thread in the system will block indefinitely.

Ensuring a Thread Always Leaves the Critical Section. We usually want to ensure `LeaveCriticalSection` is called no matter the outcome of the critical region itself. Please first recall the warnings about reliability and the possibility of leaving corrupt state in the wake of an unhandled exception

stemming from a critical region. Assuming we're convinced we do want this behavior, we can use a try/finally block.

```
EnterCriticalSection(&m_crst);
__try
{
    // Do some critical operations...
}
__finally
{
    LeaveCriticalSection(&m_crst);
}
```

While this certainly does the trick and is a fairly simple pattern to follow, it's easy to accidentally slip in a call to some function that might throw exceptions after the EnterCriticalSection but before the try block. If an exception were thrown from such a function, the finally block will not run, leading to an orphaned lock and subsequent deadlocks.

Instead of writing this boilerplate everywhere, we can use a C++ holder type (see Further Reading, Meyers). A holder is a stack allocated object that manages a resource and takes advantage of C++'s implicit destructor invocation at the end of the scope in which it's used for cleanup.

```
#include <windows.h>
class CrstHolder
{
    LPCRITICAL_SECTION m_pCrst;
public:
    CrstHolder(LPCRITICAL_SECTION pCrst)
    {
        m_pCrst = pCrst;
        EnterCriticalSection(m_pCrst);
    }
    ~CrstHolder()
    {
        LeaveCriticalSection(m_pCrst);
    }
};
```

Allocating a holder and deleting it will perform lock acquisition and release, respectively. This holder can then be used anywhere we need to create a critical region. For example, we can now go ahead and change our try/finally example to use the holder instead.

```
{  
    CrstHolder lock(&m_crst);  
    // Do some critical operations...  
}
```

Holder types typically lead to much cleaner code and allow you to consolidate any extra logic you need now or in the future. For instance, you may want to log lock acquisitions and releases or perform some kind of lock hierarchy validation, and so forth, which this approach enables you to do. But holders still aren't perfect. A legitimate argument against them is that too many of the synchronization details are hidden by using a holder. It's very easy to (accidentally) extend the lifetime of the critical region by not scoping its life correctly, which is why we introduced an explicit C++ scope block around the critical region above using extra curly braces.

Avoiding Blocking: TryEnterCriticalSection and Spin Waiting. Because blocking can be expensive, it is often profitable to avoid it. There are two techniques offered by critical sections to avoid blocking: (1) a `TryEnterCriticalSection` function, which tries to acquire the critical section but simply returns `FALSE` (rather than waiting) if it is unavailable, and (2) the capability to spin briefly before falling back to waiting on the kernel object. Let's look at both of these techniques in turn.

The `TryEnterCriticalSection` API looks just like `EnterCriticalSection`, except that it returns `BOOL` instead of `VOID`.

```
BOOL WINAPI TryEnterCriticalSection(  
    LPCRITICAL_SECTION lpCriticalSection  
) ;
```

As already mentioned, this function just checks whether the lock is available, and, if so, acquires it, returning `TRUE`; otherwise, it returns `FALSE` immediately. The caller has to check the value and execute the critical region code, if the return was `TRUE`, and do something else otherwise. This is useful if the thread has other useful work to do instead of wasting valuable processor time by blocking, for example:

```
while (!TryEnterCriticalSection(&m_crst))  
{  
    // Keep myself busy doing something else...  
}
```

```
_try
{
    // Do some critical operations...
}
_finally
{
    LeaveCriticalSection(&m_crst);
}
```

Critical sections always employ some amount of spinning to avoid blocking on multiprocessor machines. In Chapter 14, Performance and Scalability, we will examine custom spin-wait algorithms more closely and look into the math that explains why spinning can often dramatically benefit scalability. Briefly, however, spinning can lead to fewer wasted CPU cycles than waiting. If the critical section becomes available while a thread is spin-waiting, the thread never has to block on the internal event. Blocking such as this requires at least two context switches for a thread to acquire the lock, each of which costs several thousands of cycles: one switch occurs when the thread begins waiting and the second occurs when the thread must wake up to acquire the lock once it has subsequently become available. And a real wait involves at least one kernel transition. If the time spent spinning is less than the time spent switching, avoiding blocking can improve throughput markedly. On the other hand, if the critical section doesn't become available while spinning, the thread will have wasted real CPU cycles (and power) by spinning—cycles that would have otherwise gone to context switching out the thread and letting another thread run. Therefore, all use of spin waiting must be done very carefully and thoughtfully.

`EnterCriticalSection` will, by default, not perform any spinning because each critical region has a default spin count of 0. As we saw earlier, you can specify an alternative spin count instead with the `dwSpinCount` argument to `InitializeCriticalSectionAndSpinCount` or `InitializeCriticalSectionEx` API. This count is the maximum number of loop iterations `EnterCriticalSection` will spin for internally before lazily allocating and falling back to blocking on its event. Alternatively, or in addition to using initialization to set the spin count, it also can be modified later after the section has been initialized with the `SetCriticalSectionSpinCount` API.

```
DWORD WINAPI SetCriticalSectionSpinCount(
    LPCRITICAL_SECTION lpCriticalSection,
    DWORD dwSpinCount
);
```

Spin count arguments are always ignored on single-threaded machines, that is, the critical section's count will always be the default of 0 because spinning makes no sense in such cases. Also note that the high-order bit for `InitializeCriticalSectionAndSpinCount`'s `dwSpinCount` argument is ignored because it has been overloaded on some operating systems to request pre-allocation of the kernel event. Thus, the maximum spin count that can be specified is `0x7fffffff`. This code initializes a critical section with a spin count of 1,000.

```
InitializeCriticalSectionWithSpinCount(&m_crst, 1000);
```

If we later wanted to change the spin count to 500, we could just do the following:

```
DWORD dwOldSpin = SetCriticalSectionSpinCount(&m_crst, 500);
```

Notice that the `SetCriticalSectionSpinCount` function returns the old spin count; so in this example `dwOldSpin` would equal 1,000 after making the call.

Getting the spin count right is an inexact science and can have effects that differ from machine to machine. MSDN documentation recommends 4,000 based on experience from the Windows heap management team. On average, something around 1,500 is a more reasonable starting point, but this is something that should be fine-tuned based on scalability testing. Although it is possible to change the spin count after initialization with `SetCriticalSectionSpinCount`, perhaps dynamically in response to statistics gathered during execution, the spin count is usually a constant value decided during performance testing.

Windows Vista has a new dynamic spin count adjustment feature. While this is used inside the OS, it is an undocumented feature. It's possible that this feature will be officially documented and supported in an upcoming Windows SDK, but that may not happen, so I wouldn't recommend taking a dependency on it. If the `InitializeCriticalSectionEx` API is used,

passing a Flags value containing the RTL_CRITICAL_SECTION_DYNAMIC_SPIN value, the resulting critical section will use a dynamic spinning algorithm. Note that this value is defined in WinNT.h, not Windows.h, so you'll have to include that to access this functionality.

```
#include <windows.h>
#include <winnt.h>
// ...
CRITICAL_SECTION crst;
InitializeCriticalSectionEx(
    &crst, 0, RTL_CRITICAL_SECTION_DYNAMIC_SPIN);
```

When a critical section is initialized this way, the spin count supplied is completely ignored. Instead, the spin count will begin at some reasonable number and be dynamically adjusted by the OS based on whether spinning historically yields better results than blocking. The goal of this dynamic adjustment algorithm is to stabilize the spin count and to stop spinning altogether if the spinning does not statistically prevent the occurrence of context switches. While interesting, this is an experimental feature, which is probably why it's undocumented, and it's not clear if it provides any significant value to make it worth considering for use in your programs.

Low Resource Conditions

As mentioned earlier, under some circumstances the initialization of a critical section may attempt to allocate a kernel object. This allocation may fail due to low resources, leading to an ERROR_OUT_OF_MEMORY exception being thrown. Critical sections are quite different in this regard from most of the Win32 library because most other APIs will return FALSE or an error code to indicate allocation failure rather than using an exception. This is slightly annoying, because many native programmers prefer return codes to exceptions and, therefore, have to treat this as a special case or perform some translation. Worse, many don't realize it can happen, leading to reliability holes (i.e., due to unhandled exceptions in very rare and hard-to-test-for circumstances). In Vista, the new InitializeCriticalSectionEx API conforms to Win32 standards and, instead, returns FALSE to indicate failure.

Woes of Lazy Allocation. And, as also already mentioned, subsequent calls to `EnterCriticalSection` and `LeaveCriticalSection` on Windows 2000 also can throw SEH `ERROR_OUT_OF_MEMORY` exceptions as well. The reason is subtle. The kernel team made a change in the move to Windows 2000 so that critical sections would lazily allocate the kernel object the first time it was needed (i.e., when a thread needs to wait) versus the previous behavior of always allocating one during section initialization. The reason that lazy allocation was preferred is that kernel objects are heavyweight; allocating one for initialized, but unused, critical sections increases the cost of each section itself and hence the overall pressure on the system, including some consumption of nonpageable kernel memory. Particularly around the Windows 2000 time frame, many more people were writing multi-threaded code primarily for server SMP programs. It's relatively common now to have hundreds or thousands of critical sections in a single process. And many critical sections are used only occasionally (or never at all), meaning that the auto-reset event often isn't used. Requiring that kernel resources always be allocated up front became a rather large scalability limitation. But the addition of lazy initialization suddenly meant that the first time thread tried to enter a critical section already owned by another thread (with a failed spin wait) required the auto-reset kernel event to be allocated on the spot. This allocation can fail.

What's worse, you can't recover from this exception. On most OSs, the `CRITICAL_SECTION` data structure is left in a corrupt and unusable state. And it gets worse. `LeaveCriticalSection` also can fail under some even more obscure circumstances: if `EnterCriticalSection` fails, throwing an out of memory exception, a subsequent call to `LeaveCriticalSection` would notice the damaged state and respond by attempting to allocate the event. This too could fail, causing even more corruption and confusion.

Dealing with this condition effectively means that any call to enter or leave a critical section on Windows 2000 must be wrapped inside a try/catch block, which is unrealistic. A slight mitigation to this issue was made available in Windows 2000: a flag could be passed to the `InitializeCriticalSectionAndSpinCount` API to request that Windows pre-allocates the event. To pre-allocate the event at initialization time with this function, turn on the high-bit of the `dwSpinCount` argument.

```
CRITICAL_SECTION crst;  
InitializeCriticalSectionAndSpinCount(&crst, 0x80000000);
```

This is a bit of a hack, since it overloads a parameter for an entirely different purpose from its primary use. But it does the trick; that is, subsequent calls to `EnterCriticalSection` and `LeaveCriticalSection` cannot fail due to out of memory conditions. However, changing all `InitializeCriticalSection` calls to `InitializeCriticalSectionAndSpinCount` calls is tedious, and most programmers didn't even know about this problem, including many of the programmers on the Windows team. The fact is, most programs that used critical sections still used the old APIs and were vulnerable to these reliability problems, even many years after Windows 2000 shipped. All the addition of this capability did was push the fundamental reliability vs. scalability decision back onto the developer—it wasn't a real fix.

Keyed Events to the Rescue. As of Windows XP, this is no longer an issue. Windows contains a new kernel object type, called a **keyed event**, to handle low-resource conditions. Keyed events are hidden inside the kernel and are not exposed directly, though we'll see that they are used heavily in the new Windows Vista synchronization primitives (as with condition variables and slim reader/writer locks). And they are used by `EnterCriticalSection` when memory is not available to allocate a true event.

There is one keyed event, named `\KernelObjects\CritSecOutOfMemoryEvent`, that is shared among all critical sections in the process when memory becomes too low to allocate dedicated events. Each process has a `HANDLE` to this event; this is apparent if you run `!handle` from a debugger, for example, because every process will have one. There is no need for your program code to initialize or create the object; it's always there and always available, regardless of the resource situation on the machine.

How do keyed events work? A keyed event allows threads to set or wait on it, just like an ordinary Windows event. But having only a single, global event would be an inadequate solution to the critical section problem: we effectively need a single event per critical section. To solve this dilemma, any time a thread waits on or sets the event it must specify a "key," `K`. This key is any legal pointer-sized value and represents some abstract, unique identifier for the event in question. When a thread sets an event for some

key value K, only a single thread that has begun waiting on K is awakened (similar to an auto-reset event). And only waiters in the current process are awakened, so K is isolated between processes, although the keyed event object is not. Conveniently, memory addresses are very good pointer-sized unique identifiers, which is precisely how critical sections, condition variables, and slim reader/writer locks use them. You get an arbitrarily large number of abstract events in the process (bounded by the addressable bytes in the system), but without the cost of allocating a true event object for every address needed.

If N waiters must be awakened, the same key K must be set N times. So to simulate a manual-reset event, the list of waiters needs to be tracked in an auxiliary data structure. (Although not an issue for critical sections, this is needed to support reader/writer locks and condition variables.) This gives rise to a subtle corner case; if a setter finds the wait list associated with K to be empty when it sets the event, it must wait for a thread to arrive. Yes, that means the thread setting the event can wait too. Why? Because without handling this case, there would be extra synchronization needed to ensure a waiter didn't record that it was about to wait (e.g., in the critical section bits), the setter to see this and set the keyed event (and leave), and, finally, the waiter to start waiting on the keyed event without seeing that the event was set. This would lead to a missed pulse and a possible deadlock.

Let's return to the lazy allocation problem with critical regions. After keyed events were introduced, a critical section that finds it can't allocate a dedicated event due to low resources will wait on the `CritSecOutOfMemoryEvent` keyed event, using the critical section's address in memory as the key K. And a subsequent releaser will have to set the global keyed event at address K.

Given all of this, you might wonder why keyed events haven't replaced ordinary event types. There are admittedly some drawbacks to them. First, the implementation in Windows XP was somewhat inefficient. It maintained the wait list as a linked list, so finding and setting a key required an O(n) traversal. Here n is the number of threads waiting globally in the system on the single event, without any isolation between different key values of K. The head of the list is in the keyed event object itself, and entries in the linked list are threaded by reusing a chunk of memory on the waiting

thread's ETHREAD data structure for forward- and back-links, cleverly avoiding any dynamic allocation (aside from the ETHREAD memory, which is already allocated at thread creation time). But given that the event is shared physically across the entire machine, using such a design for all critical sections globally would not have scaled very well. This sharing can also result in contention that is difficult to explain, since threads have to use synchronization when accessing the list. Most low-resource conditions are transitory in nature anyway—that is, a machine encounters such a condition only temporarily, before the user kills the offending application or service—so this temporary performance degradation is much better than the risk of reliability problems. But these are the basic reasons that critical sections still allocate and use a traditional event in the common case.

Keyed events have improved quite a bit in Windows Vista. Instead of storing waiters in a linked list, they now use a hash table keyed by the key K, trading the possibility of hash collisions (and hence, some amount of contention unpredictability) in favor of improved lookup performance. This improvement led to performance good enough that it allows them to be used as the sole event mechanism for the new Vista slim reader/writer lock, condition variable, and one-time initialization APIs. None of these new features use traditional events—they use keyed events exclusively, which is why the new primitives are so lightweight, often taking up only a pointer-sized bit of data and not requiring any dedicated kernel objects whatsoever.

The improvement that keyed events offer to reliability and the alleviation of HANDLE and nonpageable pressure is overall very welcome and will pave the way for new synchronization OS features in the future. They are accessible most directly with the condition variable APIs because they internally wrap access to the keyed event object. We'll get to those in a few more sections.

Debugging Ownership Information

There is a lot of debugging information available for critical sections if you know where to look. The basic information available includes the identity of the owning thread, recursion count, and HANDLE to the kernel object used for waiting, among other things. Assuming you haven't initialized your

CRITICAL_SECTION with the CRITICAL_SECTION_NO_DEBUG_INFO flag, there's even more information available, such as the total number of times a section has been entered, experienced contention, and so on. A detailed overview of these structures is outside of the scope of this book, although there is quite a bit of information accessible programmatically for purposes of building debuggers, profilers, and the like. See Further Reading, Pietrek and Osterlund, for some additional details.

The Microsoft kernel debuggers provide extensive information about critical sections, including which locks are held by what threads. For example, the !locks command in Windbg will print out information about all of the locks that are currently owned in the process.

```
0:000> !locks

CritSec ntdll!LdrpLoaderLock+0 at 77805340
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread    d84
EntryCount       0
ContentionCount  0
*** Locked

CritSec image00400000+cf80 at 0040cf80
WaiterWoken      No
LockCount        0
RecursionCount   1
OwningThread    e50
EntryCount       0
ContentionCount  0
*** Locked

Scanned 36 critical sections
```

By default, only critical sections that are currently owned will be shown. Notice that the owning thread's OS ID is easily accessible in the output, which can be matched up with thread IDs in a kernel debugging session (i.e., with the !threads command) or in the output of the ~ thread listing command. You can specify that all locks, regardless of ownership status, be printed with !locks -v. Also note that dumping the TEB information for threads with the !teb command also lists a count of the current number of locks owned by a particular thread.

CLR Locks

The CLR provides “monitors” as the managed code equivalent to critical regions and Win32’s critical sections. Any CLR object can be used as a monitor, which can be accessed through the `System.Threading.Monitor` class’s static methods. There’s no need to initialize or delete a monitor explicitly. You allocate the object on the GC heap and the CLR will take care of any initialization and management of internal data structures needed to support synchronization.

Each monitor is logically comprised of two things: a critical section and a condition variable. Physically, the monitor does not include a Windows `CRITICAL_SECTION`, but it behaves much as though it does. We will defer discussion of the condition variable aspect of monitors until later in this chapter and focus for now on how to make use of its mutually exclusive locking capabilities.

Note also that managing a monitor object is just like managing any other kind of object in an object-oriented system. Encapsulation is important so as not to accidentally leak the target of synchronization, enabling users of your type to interfere with internal synchronization. This is why it’s generally seen as a bad practice to lock on `this` inside of an instance method. And, as with Win32 critical sections, you can decide to associate monitors with static variables or as fields of individual objects. At first it might seem convenient that you can lock on any CLR object, but it’s almost always a better idea to explicitly manage locks as you would native critical sections. Synchronization is difficult to begin with, and being thoughtful and disciplined about how locks are managed, what they protect, and so forth, is very important. Explicitly walling off your objects meant for synchronization from the rest is a good first step in this direction.

Entering and Leaving

The `Monitor.Enter` static method acquires the monitor associated with the object passed as an argument and the `Monitor.Exit` method leaves it.

```
public static void Enter(object obj);
public static void Exit(object obj);
```

If the target monitor, `obj`, is already held by another thread when you call `Enter`, the calling thread will block until the owning thread releases it.

The CLR uses Win32 events to implement waiting, which get allocated on demand and pooled among monitors. Because monitors use kernel objects internally, they exhibit the same roughly-FIFO behavior that the OS synchronization mechanisms also exhibit (described in the previous chapter). Monitors are unfair, so if another thread sneaks in and acquires the lock before an awakened waiting thread tries to acquire the lock, the sneaky thread is permitted to acquire the lock. Trying to call `Exit` on a monitor, `obj`, that is not held by the current CLR thread causes a `System.Threading.SynchronizationLockException` exception to be thrown. The monitor itself still remains in a completely valid state.

CLR monitors support recursive acquires by maintaining an internal recursion counter, so if a thread owns the monitor when a call to `Enter` is made, the acquisition succeeds and the counter is incremented. When `Exit` is called, this counter is decremented. Once it hits 0, the monitor is released, waiting threads are awakened, and other threads may freely acquire it. Each call to `Enter` must, therefore, have only one matching call to `Exit`. As mentioned earlier, recursion can cause some subtle problems, because it is dangerous to rely on invariants that would normally hold at critical region boundaries.

Ensuring a Thread Always Leaves the Monitor. As discussed earlier with Win32 critical sections, you'll typically want to use a try/finally block to guarantee your lock is released, even in the face of an exception. And, as also already noted, this sometimes is dangerous to do. An exception from within a critical region often implies that data protected by that region has (possibly) become corrupt, so releasing the lock is usually the wrong thing to do. It's often too cumbersome and time consuming to take the extra effort to validate state invariants for the extremely rare case that an exception occurs, so most programs simply don't do it.

Using a try/finally might look something like this:

```
object monitorObj = new object();

// ... elsewhere ...

Monitor.Enter(monitorObj);
try
```

```
{  
    // Do some critical operations...  
}  
finally  
{  
    Monitor.Exit(monitorObj);  
}
```

This ensures that, so long as the call to `Enter` succeeds, the call to `Exit` will always be made, no matter what happens in the critical region. Asynchronous exceptions threaten the reliability of even this code, because an exception can theoretically arise between the call to `Enter` and the entrance into the try block. We'll examine this situation in more detail just a little bit later. Because this pattern is so common, the C# and VB languages offer keywords to encapsulate this pattern. In C#, we can use the `lock` keyword.

```
object monitorObj = new object();  
  
// ... elsewhere ...  
  
lock (monitorObj)  
{  
    // Do some critical operations...  
}
```

This example is functionally equivalent to the previous one. In fact, the same IL is emitted by the C# compiler in both cases. In Visual Basic, you can use the `SyncLock` keyword.

```
Dim monitorObj As Object = new Object()  
  
' ... elsewhere ...  
  
SyncLock monitorObj  
    ' Do some critical operations...  
End SyncLock
```

To support the `synchronized` keyword in Java (for J#), which is used as a method modifier indicating callers of the method implicitly acquire/release the target monitor, there is a method-level attribute that can be used. In `System.Runtime.CompilerServices` you'll find the

`MethodImplAttribute` type. You can annotate any method definition with it, passing the `MethodImplOptions.Synchronized` flag to its constructor, and the CLR will automatically acquire and release a monitor when calls are made to it. Note that this method of synchronization is effectively deprecated and only described for educational purposes—that is, in case you run across code that is already using it.

For example, in J# we might write some function `f` to be synchronized.

```
synchronized void f()
{
    // Do some critical operations...
}
```

This is simply translated into the following.

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
void f()
{
    // Do some critical operations...
}
```

Note that this attribute is usable from any CLR language, not just J#, although most languages do not support the `synchronized` keyword itself.

The next question is, what monitor is acquired and released? For instance methods, the monitor is the instance on which the call was made. Thus, the preceding code is effectively equivalent to wrapping `f`'s body in `lock(this) { ... }`. For static methods, the monitor is the `Type` object on which the method is defined. Thus, if `f` were marked static and was on some type `T`, it would be equivalent to wrapping the method body in `lock(typeof(T)) { ... }`. While this might look nice at first glance, both instance and static methods use dangerous practices. Locking on `this` is discouraged because it exposes synchronization details; and locking on a CLR `Type` object can cause some surprisingly strange behavior because `Types` can be shared across AppDomains (more on that later).

Avoiding Blocking: TryEnter and Spin Waiting. The `Monitor` class also offers a `TryEnter` method to avoid blocking, or to block for only a certain period of time before giving up. Two of the three overloads accept a timeout—either

with an integer count of the milliseconds or a `TimeSpan` value—and all return `true` or `false` to indicate whether the lock was acquired.

```
public static bool TryEnter(object obj);
public static bool TryEnter(object obj, int millisecondsTimeout);
public static bool TryEnter(object obj, TimeSpan timeout);
```

If the `TryEnter` overload without a timeout is called, or the timeout argument is 0 or new `TimeSpan(0)`, then the method will test if the monitor is available and, if not, return `false` immediately without waiting. Otherwise, the method will block for approximately the timeout specified as an argument. (Timer resolutions vary across platforms, and, because the thread must be placed back into the OS thread scheduler to run after the timeout has expired, precisely when the thread is rescheduled for execution depends heavily on the current load of the machine.) Using `TryEnter` is a good approach to test locks for availability, choosing to spend time on some other activity instead of blocking and periodically checking back to discover when it has become available. Note that `TryEnter` is generally not good as a deadlock prevention technique, although this is perhaps its most popular (mis)use.

To use a nonblocking or timeout acquire, you have to throw out the language keywords and go back to using the `Monitor` class directly.

```
object monitorObj = new object();

// ... elsewhere ...

while (!Monitor.TryEnter(monitorObj))
{
    // Keep myself busy...
}

try
{
    // Do some critical operations...
}
finally
{
    Monitor.Exit(monitorObj);
}
```

The CLR monitor employs a small amount of spinning internally before a true wait is used. The spin-wait algorithm uses a fixed spin

count, and, unlike Win32 critical sections, you cannot change it. To your advantage, the CLR team has spent many hours of development and testing effort trying to come up with one spin count that works well, on average, and across many diverse workloads and architectures. At the same time, the general-purpose nature of this approach can be a disadvantage for extreme circumstances, including cases where you do not want to spin (such as when writing code for battery-powered devices). We'll see in subsequent chapters how to build custom spin wait algorithms in managed code.

On a single-CPU machine, the monitor implementation will do a scaled-back spin-wait: the current thread's timeslice is yielded to the scheduler several times by calling `SwitchToThread` before waiting. On a multi-CPU machine, the monitor yields the thread every so often, but also busy-spins for a period of time before falling back to a yield, using an exponential back-off scheme to control the frequency at which it rereads the lock state. All of this is done to work well on Intel HyperThreaded machines. If the lock still is not available after the fixed spin wait period has been exhausted, the acquisition attempt falls back to a true wait using an underlying Win32 event. We discuss how this works in a bit.

Note that all of these are implementation details and, thus, may change in future runtime releases. While it's doubtful the CLR would stop spinning entirely, minor changes to the algorithm itself are highly likely.

Value Types. If you pass an instance of a value type to `Monitor.Enter`, you are apt to be disappointed. A value type must be boxed before a lock can be acquired on it because `Enter`'s parameter is typed as `object` (and because lock information is held in the object header, which values do not have). Each time you box the same value, you have (implicitly) created an entirely separate and distinct object. Therefore, different threads boxing the same value get different boxed objects, and, hence, locking on them does not achieve any sort of mutual exclusion whatsoever.

The C# and VB compilers tell you if you try to pass a value to the `lock` or `SyncLock` keyword. In fact, they refuse to compile your code. C# reports an error message "error CS0185: 'T' is not a reference type as required by the lock statement," as does VB "error BC30582: 'SyncLock' operand cannot be of type 'T' because 'T' is not a reference type." If you're calling the

Monitor APIs directly, however, the compiler won't catch this problem, so you will need to be careful.

Locking on Types and AppDomain-Agile Objects. I mentioned earlier that locking on Type objects is a dangerous practice (in the context of discussing `MethodImplAttribute`). It's dangerous for much of the same reason that locking on publicly accessible objects is dangerous, at least in a reusable library: breaking lock encapsulation and, in some cases, exposing your code to accidental deadlocks. The latter is worse because deadlocks might span multiple AppDomains, which are typically thought of and treated as strongly isolated sandboxes.

First, why is it so bad to expose synchronization details to callers of your API? It's bad for the same reason exposing any implementation detail is considered poor object oriented programming. But what's worse, if you're creating a public library and your caller can access the same locks used internally within your code, the liveness of your code is left at the mercy of their responsibility. If they acquire one of these locks (for whatever reason, accidental or malicious), then your library code will contend with their code for locks. If they forget to release the lock, this can cause deadlocks in your code. If they manage to release the lock while your library thinks it is still held by the thread, they are apt to expose some new bugs that you never thought existed, possibly even leading to security vulnerabilities. (This can happen in some convoluted callstacks consisting of virtual methods interwoven between library and user code.) And worse, you'll wonder what the cause was when you receive a bug report and probably spend hours investigating only to come up empty handed.

For this last reason alone, you should never use a publicly exposed object as the target of a monitor acquisition in reusable library code. This was hinted at previously. But let's make it very explicit: if you ever run across a public class that contains statements such as `lock(this) { ... }`, it's a bug. No questions asked.

Locking on Type objects is far worse, for a very subtle reason. When an object is passed across an AppDomain boundary, it must be marshaled. Usually this is done by making a copy of the object (to keep state between AppDomains isolated), though in some cases a proxy to the same object can

be created (for `MarshalByRefObjects`). After marshaling an object in these two cases, code in either AppDomain can safely lock on the resulting object without interfering: one AppDomain locks on the original object, while the other locks on either a copy of the object or a proxy to it (with its own monitor). But there's a poorly documented case that can break this isolation: the CLR supports another marshaling mechanism, referred to informally as "marshal-by-bleed." With this marshaling mechanism, references in separate domains can refer to the same CLR object in memory. If code in the two AppDomains locks on one such object, they will be locking on precisely the same object, with exactly the same monitor. And they will clash with each other.

A lot of code and CLR infrastructure assumes isolation between AppDomains, that is, that code in one AppDomain can't corrupt state that is observable by another, totally independent, AppDomain. This is why many add-in frameworks and hosts like SQL Server can be confident that failures from one domain can be reliably dealt with by unloading the domain rather than the entire process. As soon as you start using marshal-by-bleed objects as the target of `Monitor.Enter`, you're possibly invalidating this entire set of assumptions.

What kind of objects enjoy marshal-by-bleed semantics? Domain neutral `Type` objects—as well as other reflection types (e.g., `MethodInfo`, and so forth) representing domain neutral assembly artifacts—present a nasty situation where the same objects are shared across all AppDomains in the process. By default, the only assembly that is loaded domain neutral is `mscorlib.dll`, although this can be overridden by configuration and policy, either at the host or program level. This is bad because there needn't be any inter-AppDomain communication for a single reference to be bled: two unrelated pieces of code accessing `typeof(Int32)`, for example, will suddenly have a reference to the same object in memory. CLR strings are also marshal-by-bleed. A string argument to a remoted `MarshalByRefObject` method invocation might be bled, for instance, as can be process-wide interned string literals. The `System.Threading.Thread` object is also bled across domains.

If one AppDomain orphans the lock (forgets to release it), it could cause deadlocks in other AppDomains. Even without deadlocks, there will be

false conflicts, possibly impacting scalability in a way that is impossible to track down and understand. This deadlock situation can be observed by running this tiny program.

```
#define DOMAIN_NEUTRAL

using System;
using System.Reflection;
using System.Threading;

class Program
{
    private const string s(eventName = "__SharedEvent";

    // Conditionally turn on/off domain neutrality.
#if DOMAIN_NEUTRAL
    [LoaderOptimization(LoaderOptimization.MultiDomain)]
#endif

    static void Main()
    {
        EventWaitHandle wh = new EventWaitHandle(
            false, EventResetMode.ManualReset, s(eventName);

        // Hold the lock while we wait for the other AppDomain.
        Console.WriteLine("#1: acquiring lock");
        lock (typeof(Program))
        {
            // Queue work to happen in a separate AppDomain.
            AppDomain ad2 = AppDomain.CreateDomain("2");
            ThreadPool.QueueUserWorkItem(AppDomainWorker, ad2);

            // Now wait for the other AppDomain to signal us.
            Console.WriteLine("#1: waiting for event");
            wh.WaitOne();
            Console.WriteLine("#1: exiting lock");
        }
    }

    static void AppDomainWorker(object obj)
    {
        AppDomain ad = (AppDomain)obj;

        // Execute code in the specified AppDomain.
        ad.DoCallBack(delegate
        {
            EventWaitHandle wh = EventWaitHandle.OpenExisting(
                s(eventName);
```

```
// Acquire the lock. When running w/ domain neutrality,
// this will use the same lock as the AppDomain that is
// calling us. Otherwise, it will be independent.
Console.WriteLine("#2: acquiring lock");
lock (typeof(Program))
{
    Console.WriteLine("#2: lock acquired, setting event");
    wh.Set();
    Console.WriteLine("#2: exiting lock");
}
});
```

The `LoaderOptimizationAttribute` is used in this example to conditionally turn on domain neutral loading. You can turn off domain neutral loading by commenting out the definition of the `DOMAIN_NEUTRAL` symbol. When domain neutral loading is turned on, both domains will use a shared `Type` object as the target of the `lock(typeof(Program)) { ... }` statement. In this particular example, this leads to deadlock because the primary domain waits forever for the second domain to set an event, but the second domain waits for the primary domain to release the lock on `typeof(Program)`. A similar effect can be achieved by replacing `lock(typeof(Program)) { ... }` with `lock("foo") { ... }`, because by default "foo" is interned and shared across domains. Turning off domain neutral assembly loading causes each `AppDomain` to have a separate `Type` object, and, hence, they do not interfere.

This, in the author's opinion, is a bug in the CLR. This is actually a perfect example of a leaky abstraction provided by the CLR, and it's admittedly quite terrible that you need to know anything about it. But given that it's persisted for several releases already and that the cost of Microsoft fixing it is probably prohibitively expensive for compatibility reasons, it's likely to persist into the foreseeable future. The `DoNotLockOnObjectsWithWeakIdentity` VSTS 2005 code analysis rule looks for and warns you for some well-known cases, with the standard static analysis caveats.

Reliability and Monitors

The CLR uses various asynchronous exceptions, such as thread aborts, which can interrupt your code at any instruction. In earlier examples, we

used try/finally blocks to “guarantee” that a lock is released reliably, regardless of whether the outcome of the try block was success or failure (i.e., exceptional). Asynchronous exceptions complicate matters. Consider this snippet of code.

```
Monitor.Enter(monitorObj);
S0;
try
{
    S1;
}
finally
{
    Monitor.Exit(monitorObj);
}
```

No matter the successful or failed execution of S1, we can be assured that the monitor for obj will be exited. But what happens if S0 causes an exception? It should be obvious, but in this case, the try block will not have been entered and, therefore, the finally block will not run. And the monitor will be orphaned at that point, possibly leading to subsequent deadlocks on any threads that tried to acquire a lock on monitorObj.

Most developers realize this and don’t put any code between the call to `Monitor.Enter` and the try block. In fact, most people will use the C# lock or VB SyncLock statement to achieve this. But that doesn’t necessarily mean that a compiler won’t put any code there. S0 could be as simple as a NOP instruction in the assembly code generated by the CLR’s JIT compiler: in this case, all we need is an asynchronous thread abort to be generated while the thread’s instruction pointer is at this NOP instruction, and the abort would occur before the thread’s instruction pointer moves inside the try block. This has the same effect we described previously: `Monitor.Exit` doesn’t get called.

As a brief aside, `Monitor.Enter` is special. If it was written in managed code, a thread abort also could get triggered after it had acquired the lock but before it returned to the caller. This would suffer from the same problem. It turns out that, because `Monitor.Enter` is written as an `mscorwks.dll` native function, asynchronous thread aborts cannot interrupt it. Such code must poll for and give permission for a thread abort to occur. Managed code, on the other hand, can be interrupted at any instruction (except when

inside some special uninterruptible regions such as finally blocks or constrained execution regions). This is subtle, but key to making some of the guarantees we're about to discuss.

There is some good news. The C# code generation for the lock statement ensures there are no IL instructions between the CALL to Monitor.Enter and the instruction marked as the start of the try block, but only in nondebug builds (i.e., those for which /debug was not supplied to csc.exe). The X86 JIT correspondingly will not insert any machine instructions in between them either. And because any attempted thread aborts in Monitor.Enter are not polled for after the lock has been acquired and before returning, the soonest subsequent point at which an abort can happen is the first instruction following the call to Monitor.Enter. At that point, the thread's instruction pointer will already be inside the try block (the return from Monitor.Enter returns to the CALL+1), thereby ensuring that the finally block will always run if the lock was acquired. This might seem like an implementation detail, but the CLR team can't change it. Too many people have written code that would suddenly be exposed to subtle reliability bugs if it were changed.

CLR 2.0's X64 JIT did not guarantee this. In fact, in the X86 JIT used to generate machine code that always had a NOP instruction between the CALL and the instruction marking the try block in the jitted code. This is done for internal reasons, to make it easier to identify try/catch scopes during stack unwind. This means that, yes indeed, an abort can happen at S0 on 64-bit, even if it was empty in the original program. This was fixed in the 3.5 release. If you don't compile with optimization flags, your compiler is still apt to insert padding instructions (for debuggability reasons) that cause this problem to surface.

In the end, relying on this for correctness is a bad idea. Most people don't need to write code that will survive asynchronous thread aborts. If you are worried about such things, however, at least you now know the full story, including some of the limitations in the current implementation. You should always devise a fallback plan.

How Monitors Are Implemented

It's worth discussing briefly how monitors are implemented. Each CLR object has an object header, which is a double pointer-sized block of

memory that resides just prior to the address in memory to which an object reference points. The contents of this memory are used by the CLR to manage various bits of information. If you've ever called `GetHashCode` on an object (whose `GetHashCode` method hasn't been overridden), the runtime generated hash code is remembered in the object header as a lightweight way of ensuring that it doesn't change over time. COM interoperability information is also held here for certain objects.

What's interesting from the perspective of monitors is that half of the object's header also is used for a monitor's so-called **thin lock**: encoded in less than a naturally sized word is the ID of the CLR thread that currently owns the monitor and a recursion counter. This thin lock mechanism is nice because it's cheap to maintain and each object has this block of memory already allocated and easily reachable by subtracting a few bytes from its reference. It can't always be used due to something called *object header inflation*.

Clearly it's not possible to store a hash code, thin lock ownership information, and COM interoperability information in the same object header at once. An object's hash code is (approximately) a 4-byte integer, as is the thread ID, and yet we only have a naturally sized word available. Though the domain of both is constrained a little so that a few extra bits can be used, it's not constrained to less than what 2 bytes can represent: so if we only have 4 bytes in the header on a 32-bit system, we obviously can't cram both a hash code and thread ID into an object's header at once. Moreover, a thin lock only works if all we need to store is the owner ID and recursion count; if we ever need to allocate and store an event handle for waiting purposes, we will need more space. To deal with this, the CLR lazily **inflates** the object header, by allocating a sync block for the object if there isn't sufficient room in the object header for all of the information that needs to be stored. The sync block is taken from an ever-expanding pool of shared memory, and an index into this pool is stored in the object header. From that point on, anything previously stored in the object header goes onto the object's sync block, including lock information.

Once a monitor experiences contention, that is, a thread attempts to acquire an already owned lock and wasn't able to obtain it by spinning briefly, a Win32 auto-reset event will be allocated. The CLR pools these events along with its pool of sync blocks. When a GC is subsequently triggered, any

objects inspected are eligible to be **deflated**, which entails returning their sync block back to the pool of available blocks. This can be done so long as the sync block isn't needed permanently (e.g., for COM interop cases), and so long as it has not been marked **precious**, which happens anytime a thread owns the monitor, when a thread is actively waiting for it, or when at least one thread is waiting on the object's condition variable. Notice that orphaning monitors can, thus, lead to leaked event objects, because they will remain precious, until the monitor object itself becomes unreachable. When a sync block is reclaimed in this fashion, the next use of the monitor will use a thin lock, and certain reusable state is returned to the pool (as with the event object, so that the next monitor to need a sync block can reuse it).

Debugging Monitor Ownership

A number of useful debugging features exist for CLR monitors. Some of the following techniques can come in handy for interactive debugging or post-mortem analysis of crash dumps.

Using the SOS debugging extension, one can dump a list of objects in the GC heap that currently have thin locks associated with them. These are locks that have not been contended and that reside on objects whose headers still had sufficient space to store the thin lock information, as reviewed previously. After loading SOS in the Immediate Window of Visual Studio, type `!DumpHeap -thinlock` to print all thin locks currently in the heap.

```
> !DumpHeap -thinlock
Address      MT      Size
012b1c6c  790f9c18  12      ThinLock owner 3 (001aff48) Recursive 1
```

This sample output shows that the thin lock for the object at address `0x012b1c6c` is held by thread `0x001aff48` and that the thread has recursively acquired the lock once. Notice that a recursion count of `0` in the `!DumpHeap` command means that the lock is acquired but has not been acquired recursively. Somewhat confusingly, a value of `1` is sometimes used to represent the same information for other SOS commands. If there were many objects in the heap that presently have a thin lock, each would be shown on a separate line. If we dump information about an object directly with `!DumpObj` (or `!do` for short), we will see the same information printed

about the thin lock. For example, if we dump the object that holds the lock as seen above, we might see something like this:

```
> !do 012b1c6c
Name: System.Object
MethodTable: 790f9c18
EEClass: 790f9bb4
Size: 12(0xc) bytes

(C:\WINDOWS\...\mscorlib.dll)
Object
Fields:
None
ThinLock owner 3 (001aff48), Recursive 1
```

The thread ownership information (`0x001aff48`) is the address of an internal data structure, so it's not something you can easily correlate with a managed thread ID directly. Using the SOS `!Threads` command, you can trace the address back to the thread object itself by matching the `ThreadOBJ` address with the lock ownership information.

```
> !Threads
ThreadCount: 5
UnstartedThread: 0
BackgroundThread: 1
PendingThread: 0
DeadThread: 0
Hosted Runtime: no

      ID  OSID  ThreadOBJ  State   PreEmptive  GC Alloc  Lock
      ID  OSID  ThreadOBJ  State   Enabled    Context    Domain Count  APT  Exception
3692  1    e6c     001871a0  8a028  Enabled   00000000:00000000  0014f238  1   MTA
5568  2    15c0    0018a838  b228   Enabled   00000000:00000000  0014f238  0   MTA  (Finalizer)
2856  3    1750    001aff48  8b028  Enabled   00000000:00000000  0014f238  1   MTA
1180  4    49c     001b2780  b028   Enabled   00000000:00000000  0014f238  0   MTA
6104  5    17d8    001b76b0  8b028  Enabled   00000000:00000000  0014f238  0   MTA
```

The third row contains the managed thread with a `ThreadOBJ` address of `0x001aff48`, which is the thread from the above lock ownership dumps. So based on this, we now know that the thread with ID 3 currently owns the lock on object `0x012b1c6c`. You can also see that its Lock Count is 1, which represents the total number of distinct monitors the target thread holds (and does not take into account recursive acquires).

This is very useful, but we still haven't seen how to get debugging information about fat locks. Once a lock is inflated from thin to fat, it will no longer be reported by !DumpHeap -thinlock. Instead, you have to run the !SyncBlk command, optionally passing a specific sync block index as an argument. When called without arguments, the sync blocks for all objects that are currently actively locked by a thread are shown. !SyncBlk -all shows all sync blocks in the process, including those without current owners.

Imagine that, in the above example, a bunch of threads have entered the system and tried to acquire a lock on object 0x001b20c8 while thread ID 3 still owns it. This would inflate the lock to a fat lock, as could be then seen by running the !SyncBlk SOS command.

```
> !SyncBlk
Index SyncBlock MonitorHeld Recursion Owning Thread Info SyncBlock Owner
 5 001b218c          19          2 001aff78 b282856 012b1c6c
System.Object
-----
Total      11
CCW        0
RCW        0
ComClassFactory 0
Free       0
```

We can see here that 0x001aff78 still owns the lock on object 0x012b1c6c. We also see that the recursion count reflected is 2. Unfortunately the !SyncBlk command starts counting at 1, versus the !DumpHeap and !DumpObject commands which start counting at 0. In other words, a value of 1 means "no recursive acquires" instead of the value 0. Although neither !DumpHeap nor !DumpObject will report lock ownership information for inflated locks, !Threads will still account for fat lock acquisitions in its Lock Count column.

Reader/Writer Locks (RWLs)

So far we've been talking about mechanisms to achieve complete mutual exclusion. Often, mutual exclusion is a stronger guarantee than is

absolutely needed. That's OK, because it's still correct. Marking entire regions of code as critical regions, that is, mutually exclusive—no questions asked—can simplify things, leading to code that is easy to understand, maintain, and debug. With that said, it's sometimes preferable to take advantage of the fact that read/read conflicts are safe; this allows us to allow multiple concurrent readers to access shared data so long as there isn't a writer present. Because the number of reads typically outnumbers writes (the ratio is about 2.5 to 1 in mscorelib.dll, as one data point), allowing these reads to happen parallel with one another can dramatically improve the scalability of a piece of code. That's not to say this is always the case, but it often is.

That's where **reader/writer locks** (RWLs) enter the picture. While implementations vary quite a bit from one another in detail, RWLs have the following basic requirements.

- When a thread acquires the lock, it must specify whether it is a reader or writer.
- At most one writer can hold the lock at a given time (exclusive mode).
- So long as there is a writer, no readers may hold the lock.
- Any number of readers can hold the lock at a given time (shared mode).

Windows Vista now offers a “slim” RWL with these precise characteristics. The .NET Framework offers two, one of which has been available since the .NET Framework 1.1, while the other is new with 3.5. Although the latter supersedes the old one, we'll look at both in this section.

As a quick thought experiment, pretend we have a fully loaded server with 32 CPUs, and each CPU is executing a single request concurrently at all times. On a heavily loaded server, this is likely to be the case, that is, the server will have more work than it can perform at a given time. If the workload running on these threads spends 6 percent of its time reading some shared data, and 0.25 percent of its time writing that same shared data, then we would see a massive increase in throughput by using shared locks. (The other 93.75 percent of the time is spent doing something that does not

involve this shared data. It's very common, particularly for server programs, to share data minimally between requests.) Not all cases are this clear-cut and obvious, but choosing an extreme example can help to serve as an illustration.

Let's see why this is the case. If all locks were exclusive, then 6.25 percent of each thread's time would be spent inside of the critical region. Thirty-two times 6.25 percent is 2. Thus, at any given time, we expect there to be 2 threads wanting to be in the critical region. You might notice a problem with this. If at every unit of time only 1 thread can actually be inside of the lock, then this means we'll always have threads waiting for others to finish. As soon as the other thread finishes, 2 more threads will want to be in the region, and so on. There will be a continuous build-up of threads at the critical region, and it's possible that soon all 32 threads will be waiting for the lock. This is a phenomenon known as a **lock convoy**, and is treated in more detail in Chapter 11, Concurrency Hazards.

Now imagine, instead, that threads can acquire the lock in shared mode when they only need to read the shared data. Only 0.25 percent of the time will any thread need to hold the exclusive lock. Thirty-two times 0.25 percent is only 8 percent, which indicates there will be very little contention for the lock on average. The fact is that 6 percent of the time, a shared lock is needed may cause some degree of contention between the shared and exclusive threads—since shared acquisitions still need to wait for exclusive locks to be released—which is hard to capture in such a simplistic model. You can easily see how this turns an entirely non-scalable design into one that scales well. Again, few cases are so clear-cut, but most workloads exhibit similar characteristics to one degree or another.

Windows Vista Slim Reader/Writer Lock

The Windows Vista slim reader/writer lock (SRWL) is similar to the critical section data type we saw earlier. The key difference is that SRWLS support shared-mode locks in addition to exclusive-mode. But there are other interesting differences. SRWLS are lighter weight than critical sections due to: (1) using only a pointer-sized amount of memory (versus several pointers), and (2) relying exclusively on keyed events instead of allocating a per lock kernel event object. There are also some other basic

feature level differences between them that we'll cover later, such as SRWLS being nonrecursive.

As with the CRITICAL_SECTION, a SRWL instance is a simple structure, SRWLOCK, that can be allocated anywhere you choose. SRWLS are new to Vista, so you'll have to define a _WIN32_WINNT version of 0x0600 or greater before importing Windows.h to use them.

Before using a SRWLOCK instance, you have to initialize it with a call to InitializeSRWLock. Because SRWLS don't use any dynamically allocated events or memory internally, there is no need to delete them later on, and initialization ensures the right bit pattern is contained in memory.

```
VOID WINAPI InitializeSRWLock(PSRWLOCK SRWLock);
```

Once you have initialized the lock, threads can then begin acquiring in exclusive (write) or shared (read) mode with the AcquireSRWLockExclusive and AcquireSRWLockShared functions, respectively. Both accept a single argument of type PSRWLOCK, which is a type definition for SRWLOCK *, and have no return value. The corresponding functions ReleaseSRWLockExclusive and ReleaseSRWLockShared release the lock in the specified mode.

```
VOID WINAPI AcquireSRWLockExclusive(PSRWLOCK SRWLock);
VOID WINAPI AcquireSRWLockShared(PSRWLOCK SRWLock);
VOID WINAPI ReleaseSRWLockExclusive(PSRWLOCK SRWLock);
VOID WINAPI ReleaseSRWLockShared(PSRWLOCK SRWLock);
```

Attempted lock acquisitions will block if the lock is held by another thread in a mode that is incompatible at the time of the attempted acquisition: that is, if the thread is owned exclusively, all attempts block; if it is owned in shared mode, exclusive attempts block. Blocking is done with a nonalertable wait, and waiters are released in a roughly FIFO order, although the lock is unfair and will permit concurrent acquisition attempts to succeed. When the lock is released and both readers and writers are waiting, the lock will prefer to wake up waiting writer threads first. When there are no writers, all waiting reader threads are awakened.

Acquiring a SRWL in shared or exclusive mode will never fail due to low resource conditions, and, hence, there is no alternative API to pre-allocate internal data structures. Once a SRWL has been initialized, it's ready to use. The secret to SRWL's ability to work in low resource conditions is the

same secret to critical sections working in low resource conditions: keyed events. The substantial performance improvements made to keyed events in Windows Vista has made it possible to use them as the sole waiting mechanism for SRWLs. In fact, you might want to consider using SRWLs with exclusive-mode-only acquisitions and releases over Win32 critical sections, due to their lightweight nature. For small amounts of contention, a SRWL will actually outperform a critical region.

Unlike critical sections, SRWLs don't support nonblocking acquire APIs, such as `TryAcquireSRWLockExclusive`, for example. This would be a nice feature, but it has not yet been made available. SRWLs also use a spin-wait for a constant number of spins that is neither configurable nor dynamic, but that has been chosen for good average case performance, much like CLR monitors.

Also note that Vista SRWLs do not support changing the lock mode after the lock has been acquired. For example, "upgrading" from shared to exclusive or "downgrading" from exclusive to shared are fairly common features for RWLs, but (due to its lightweight nature), the Vista lock doesn't support either.

Here's an example of using one such lock.

```
class C
{
    SRWLOCK m_rwl;

public:
    C()
    {
        InitializeSRWLock(&m_rwl);
    }

    void SomeReadOperation(...)
    {
        AcquireSRWLockShared(&m_rwl);
        __try
        {
            // Do some critical read operations...
        }
        __finally
        {
            ReleaseSRWLockShared(&m_rwl);
        }
    }
}
```

```

void SomeWriteOperation(...)
{
    AcquireSRWLockExclusive(&m_rwl);
    __try
    {
        // Do some critical write operations...
    }
    __finally
    {
        ReleaseSRWLockExclusive(&m_rwl);
    }
}
};

```

As with critical sections, it often makes sense to use a holder class for SRWLs to ensure you don't forget a `__finally` somewhere. The same caveats apply: reliability should be a concern, and you must take care not to accidentally extend the hold time of your locks due a big scope.

```

class SRWLockHolder
{
    PSRWLOCK m_pSrwl;
    BOOL m_pShared;

public:
    SRWLockHolder(PSRWLOCK pSrwl, BOOL pShared)
    {
        m_pSrwl = pSrwl;
        m_pShared = pShared;
        if (pShared)
            AcquireSRWLockShared(m_pSrwl);
        else
            AcquireSRWLockExclusive(m_pSrwl);
    }

    ~SRWLockHolder()
    {
        if (pShared)
            ReleaseSRWLockShared(m_pSrwl);
        else
            ReleaseSRWLockExclusive(m_pSrwl);
    }
};

```

SRWLs do not support recursive exclusive lock acquisitions. If a thread has already acquired either the read or write lock for a particular SRWL, attempting to acquire either the read or write lock on the same thread

again will lead to deadlock. This is acceptable because, as mentioned previously, recursive acquisitions can lead to brittle design. But it can still cause difficulties for designs that would otherwise call for recursion. There's another subtle implication. Because the SRWL doesn't need to support recursive acquisitions, it also doesn't need to track ownership information. (This would be hard to do anyway due to its compressed size.) This last point helps to make SRWL ultra-slim, but also makes it harder to debug: unlike the CRITICAL_SECTION data structure, a SRWLOCK doesn't actually have an OS thread ID embedded in it. (You can wrap acquisitions and releases yourself to track this data if it's important.) But this can make debugging more painful. The lack of ownership information has another implication.

Recall the behavior of `LeaveCriticalSection` when called on a thread that doesn't currently own the lock. With some caveats, it leaves the CRITICAL_SECTION in a damaged state so that no future acquisitions on it will succeed. In the simple case, a call to `ReleaseSRWLockExclusive` or `ReleaseSRWLockShared` on a completely unowned SRWLOCK will raise an exception. The exception type is not public and is defined as `STATUS_RESOURCE_NOT_OWNED` in `NtStatus.h` with a value of `0xC0000264L`. That's OK. You seldom want to catch this anyway because it represents a program bug. But it helps to know the exception code when you're stuck in the debugger faced with an unhandled exception. Because the SRWLOCK doesn't track ownership information, a thread that doesn't even hold a lock can exit another thread's lock. The lock can't differentiate this case from a correct lock release; eventually some thread will notice that the lock is not held any longer when it tries to release it, and this will cause an exception. By this point, the source of the bug has been lost and must be reconstructed by analysis.

.NET Framework Slim Reader/Writer Lock (3.5)

As mentioned above, there are two reader/writer locks in the .NET Framework, both in the `System.Threading` namespace: `ReaderWriterLock` and `ReaderWriterLockSlim`. As the name implies, the latter is lighter weight (having been written in managed code), and should yield much better performance than the old one. (Note that the footprint of the new lock can, in

some cases, be greater than the old one due to the use of multiple event objects.) The new RWL is available in .NET Framework 3.5, whereas the old RWL has been available in the .NET Framework since 1.1. We'll focus primarily on the new one, and will describe it first, but will cover the old one for legacy reasons. If you're writing new code, you should be using the `ReaderWriterLockSlim` class.

To use this lock, you will need to allocate an instance using one of the two constructors: a no-argument overload and one that takes a `LockRecursionPolicy` value to control whether the resulting lock permits recursive acquires or not (the default is `NoRecursion`).

```
public ReaderWriterLockSlim();
public ReaderWriterLockSlim(LockRecursionPolicy recursionPolicy);
```

The lock type encapsulates several kernel events to perform waiting, and, thus, when you are done with the object, you can invoke `Dispose` to clean up any events that were allocated. (They are allocated lazily as needed, so they won't necessarily always be there.) This is optional but helps to alleviate pressure on the GC due to a reduction in finalizable objects.

Three Modes: Shared, Exclusive, and Upgrade

The new `ReaderWriterLockSlim` actually supports three lock modes, shared, exclusive, and upgrade, rather than the traditional two. There are corresponding methods `EnterReadLock` (shared), `EnterWriteLock` (exclusive), `EnterUpgradeableReadLock` (upgrade), and related methods `TryEnterXXLock`, and `ExitXXLock`, that do what you'd expect.

```
public void EnterReadLock();
public bool TryEnterReadLock(int millisecondsTimeout);
public bool TryEnterReadLock(TimeSpan timeout);
public void ExitReadLock();
public void EnterWriteLock();
public bool TryEnterWriteLock(int millisecondsTimeout);
public bool TryEnterWriteLock(TimeSpan timeout);
public void ExitWriteLock();
public void EnterUpgradeableReadLock();
public bool TryEnterUpgradeableReadLock(int millisecondsTimeout);
public bool TryEnterUpgradeableReadLock(TimeSpan timeout);
public void ExitUpgradeableReadLock();
```

As the names indicate, `EnterXXLock` will acquire the lock in the specified mode `XX`. `TryEnterXXLock` will also attempt to acquire the lock in mode `XX`, but will return `false` if the timeout period (in either milliseconds or a `TimeSpan`) expires before succeeding. The format for timeouts acts precisely as do monitors: that is, a `0` value or `new TimeSpan(0)` indicates that the lock should be acquired if available, but otherwise, the call returns right away without blocking; and `-1` (or `Timeout.Infinite`) indicates that the attempted acquisition should never timeout. `ExitXXLock` releases the lock in the specified mode. The lock tracks ownership ID information (using the managed thread ID), so trying to release a lock mode that hasn't been acquired by the calling thread results in a `SynchronizationLockException`.

Shared and exclusive mode should be familiar: shared is a typical read lock mode, in which any number of threads can acquire the lock in shared mode simultaneously, and exclusive is a typical mutual exclusion mode, in which no other threads are permitted to simultaneously acquire the lock in any of the other modes. The upgrade mode will probably be new to most people, though it's a concept that's well known to database practitioners and is the mode that enables deadlock free upgrades. When a thread has acquired the lock in upgrade mode, it should be treated as though it is an ordinary shared mode lock until the act of upgrading or downgrading has been initiated. We'll look at the differences more closely later.

There are corresponding properties, `IsReadLockHeld`, `IsWriteLockHeld`, and `IsUpgradeableReadLockHeld`, to determine whether the current thread holds the lock in the specified mode. These are very useful for asserting ownership (or lack of ownership) at certain interesting parts of your program. You can also query the `WaitingReadCount`, `WaitingWriteCount`, and `WaitingUpgradeCount` properties to see how many threads are waiting to acquire the lock in the specific mode, and `CurrentReadCount` to see how many concurrent readers there are. The `RecursiveReadCount`, `RecursiveWriteCount`, and `RecursiveUpgradeCount` properties tell you how many recursive acquires the current thread has made for the specific mode, assuming recursion has been enabled for the lock. All of these properties are good debugging aids and not things you'll need to access programmatically.

Upgrading

Let's look at the upgrade mode more closely now. This mode allows you to safely upgrade from shared to exclusive mode. To illustrate why it's generally not safe to upgrade from shared to exclusive mode, imagine we have two threads that hold the shared mode lock and simultaneously attempt to upgrade: each would have to wait for the other before upgrading to exclusive mode (because the lock may only be held in exclusive mode when there are no other owners in any other mode), which leads to deadlock. As we'll see, the old `ReaderWriterLock` type supports deadlock free upgrading by releasing the lock and reacquiring it, but this breaks atomicity and is a bad design (particularly since most people don't realize it happens). The new lock neither breaks atomicity nor causes deadlocks. This is achieved by allowing only one thread to be in the upgrade mode at once, though there may be any number of other threads in shared mode while a possible upgrader holds the lock.

Once the lock is held in the upgrade mode, a thread can then read state to determine whether to downgrade to shared or upgrade to exclusive. Ideally this decision should be made as fast as possible: holding the upgrade lock causes any new shared mode acquisitions to wait, though existing shared mode holders are permitted to remain active. To downgrade, after acquiring in upgrade mode you must call `EnterReadLock` followed by `ExitUpgradeableReadLock`; this permits other shared and upgrade mode acquisitions to complete that were previously held up by the fact that the upgrade lock was held. To perform an upgrade, you call `EnterWriteLock` while holding the upgrade lock; this may have to wait until there are no longer any threads that still hold the lock in shared mode, but will not cause deadlock.

Here's some code that illustrates conditionally upgrading or downgrading based on some program specific logic.

```
ReaderWriterLockSlim rwl = ...;
...
bool needsRelease = true;
rwl.EnterUpgradeableReadLock();
try
```

```
{  
    if (... we want to upgrade ...)  
    {  
        // Perform the upgrade:  
        rwl.EnterWriteLock();  
        try  
        {  
            ... write to state ...  
        }  
        finally  
        {  
            rwl.ExitWriteLock();  
        }  
    }  
    else  
    {  
        // Perform the downgrade:  
        rwl.EnterReadLock();  
        rwl.ExitUpgradeableReadLock();  
        needsRelease = false;  
        try  
        {  
            ... read from state ...  
        }  
        finally  
        {  
            rwl.ExitReadLock();  
        }  
    }  
}  
finally  
{  
    if (needsRelease)  
        rwl.ExitUpgradeableReadLock();  
}
```

Upgrade locks are not used in many cases, but often you need to hold a shared mode lock in order to read state that determines whether exclusive mode is required. Having a dedicated upgrade mode accommodates such cases.

Recursive Acquires

Another nice feature with the `ReaderWriterLockSlim` type is how it treats recursion. By default, all recursive acquires, aside from the upgrade and

downgrade cases already mentioned, are disallowed. This means you can't call `EnterReadLock` twice on the same lock from the same thread without first exiting the lock and similarly with the other modes. If you try, you get a `LockRecursionException` thrown. You can, however, turn recursion on at construction time: pass the enum value `LockRecursionPolicy.SupportsRecursion` to your lock's constructor, and recursion will be permitted. The chosen policy for a given lock is subsequently accessible from its `RecursionPolicy` property.

There's one special case that is never permitted, regardless of the lock recursion policy: acquiring an exclusive lock when a shared lock is held. This is dangerous and leads to the same shared-to-exclusive upgrade deadlocks that were mentioned earlier. The designers of this lock (of which I was one) didn't want to lead developers down a path fraught with danger. If you need this kind of recursion, it's a matter of changing your design to hoist a call to either `EnterWriteLock` or `EnterUpgradeableReadLock` (and the corresponding exit method[s]) to the outermost scope in which the lock is acquired. This leads to less scalability, but will at least remain live (i.e., it won't suffer from deadlock).

A Limitation: Reliability

First, unlike monitors and the old `ReaderWriterLock` the `ReaderWriterLockSlim` type does not cooperate with CLR hosts through the hosting APIs. This means a host will not be given a chance to override various lock behaviors, including performing deadlock detection (as SQL Server does). Thus, you should not use this lock if your code will be run inside SQL Server or another similar host.

Next, this lock is not currently hardened against asynchronous exceptions such as thread aborts and out-of-memory conditions (like monitor). (Note that this is not unique to this particular RWL: the old RWL suffers from this problem too.) If either one of these occurs in the middle of one of the lock's methods, the lock state can become corrupt, causing subsequent deadlocks, unhandled exceptions, and, due to the use of spin locks internally, a pegged 100 percent CPU. So if you're going to be running your code in an environment that regularly uses thread aborts or attempts to survive hard `OutOfMemoryExceptions`, this lock will probably not satisfy your

requirements. It doesn't even mark critical regions appropriately, so hosts that do make use of thread aborts won't know that the thread abort could put the AppDomain at risk; many hosts would prefer to wait, or immediately escalate to an AppDomain unload, if an individual thread abort is necessary while the thread is in a critical region. But in the case of `ReaderWriterLockSlim`, a host has no idea if a thread holds the lock because the implementation doesn't call `Begin-` and `EndCriticalRegion`. And the kind of problems I mentioned earlier in the context of thread aborts and orphaned monitors are always a risk with `ReaderWriterLockSlim` because the CLR never guarantees that there will be no instructions in the JIT generated code between the acquisition and entrance to the following try block, assuming a try/finally is used.

All of these problems sound a lot more severe than they are. Large swaths of .NET Framework libraries are not resilient to these severe conditions, so if the above text made `ReaderWriterLockSlim` sound special in this regard it was unintentional. It does, however, differ from the level of reliability provided for CLR monitors. In the end, most managed programs needn't worry about such things: only if you're proactively using things like constrained execution regions and have to achieve an extraordinarily high degree of reliability should you pay attention to these potential issues.

Motivation for a New Lock

The primary reason for the addition of a new RWL was that Microsoft wanted to provide an official reader/writer lock for the .NET Framework upon which people could rely for performance critical code. It was no secret that the old `ReaderWriterLock` type performs poorly, with around 6 times the cost of a monitor acquisition for uncontended write lock acquires. Consequently, most people avoided it entirely and would either use mutual exclusive locks, roll their own, or download one of the various locks that other people had written and published in articles, weblogs, and so on.

Second, there were a large number of flaws with the old lock's design. It had funny recursion semantics (and is in fact broken in a few COM interop related thread reentrancy cases) and has a dangerous nonatomic upgrade method, as noted above. All of these problems represent very fundamental flaws in the existing type's design, which made it unsalvageable.

The new lock eliminates all of the major adoption blockers that plagued the old one, such as deadlock free and atomicity preserving upgrades, and leads developers to program cleaner designs free of lock recursion. It also has better performance, roughly equivalent to `Monitor`. (When I say “roughly,” I mean that it’s within a factor of 2 times in just about all cases.) And the new lock favors letting threads acquire the lock in exclusive mode over shared or upgradeable-shared because writers tend to be less frequent than readers, meaning this policy generally leads to better scalability. Admittedly there are some reliability oriented downsides to the new lock, so some programmers writing hosted or low-level reliability sensitive applications may have to wait to adopt it. `ReaderWriterLockSlim` is suitable for most developers out there.

.NET Framework Legacy Reader/Writer Lock

The old RWL type `ReaderWriterLock` has been around since version 1.1 of the .NET Framework and is quite a bit like the new `ReaderWriterLockSlim`. You must allocate an instance and manage it as you would any other kind of lock. And this lock supports just the two traditional RWL lock modes: shared and exclusive. Note that, while resources are indeed used internally, this lock does not implement `IDisposable` and, therefore, there’s no way to proactively reclaim its resources. It is also implemented primarily in `mscorwks.dll` (internal to the CLR) and, therefore, holds on to some memory from the native memory heap, which is why it has a **critical finalizer** (a finalizer that is guaranteed to run in more cases).

The simplest usage pattern for this lock involves calling the `AcquireReaderLock` (shared) and/or `AcquireWriterLock` (exclusive) methods, along with the corresponding `ReleaseReaderLock` and/or `ReleaseWriterLock` methods.

```
public void AcquireReaderLock(int millisecondsTimeout);
public void AcquireReaderLock(TimeSpan timeout);
public void ReleaseReaderLock();
public void AcquireWriterLock(int millisecondsTimeout);
public void AcquireWriterLock(TimeSpan timeout);
public void ReleaseWriterLock();
```

Notice that there are no overloads without timeouts offered by `ReaderWriterLock`. As with all of the other timeout parameters we’ve seen, -1 (or

`Timeout.Infinite`) may be passed to indicate no timeout is desired. Also note another slight difference: unlike most timeout variants, these do not return a `bool`; instead, they will throw an `ApplicationException` if the acquisition does not succeed prior to the timeout expiring. If you attempt to release a lock mode that is not held by the calling thread, an `ApplicationException` will be thrown.

This lock also freely supports any kind of recursion you might attempt: shared-to-shared, exclusive-to-exclusive, shared-to-exclusive, and exclusive-to-shared. Note that shared-to-exclusive recursion is very dangerous for reasons already outlined: it is highly susceptible to deadlock. The lock offers properties to inquire as to the current state of the lock, `IsReaderLockHeld` and `IsWriterLockHeld`, which are useful when asserting ownership. If both the shared and exclusive lock are held by the current thread (due to recursion), `IsReaderLockHeld` will return `false` anyway.

There is another way of releasing ownership of the lock, the `ReleaseLock` method.

```
public LockCookie ReleaseLock();
```

This is used to release the lock completely in just a single method call, including all recursive calls made on the calling thread. It returns a `LockCookie` structure, which can be subsequently used to restore the entire sequence of recursive lock acquisitions later on with the `RestoreLock` method.

```
public void RestoreLock(ref LockCookie lockCookie);
```

This is a dangerous practice because, once the lock has been released, additional threads can sneak in and invalidate any invariants that held before the call to `ReleaseLock`. Similarly, the thread releasing the lock must ensure that invariants are consistent so that the state is not seen as being corrupted by other threads that may enter the lock. It is a much better practice to cleanly unwind and pair each recursive acquisition with a release. `ReleaseLock` and `RestoreLock` can be used in some very limited circumstances where you need to ensure a thread's acquisitions do not hold up progress in the system, such as when waiting for a COM synchronization context.

Upgrading

As noted before, the `ReaderWriterLock` type does support upgrading and downgrading, albeit in an inferior way. It has three methods for this purpose.

```
public void DowngradeFromWriterLock(ref LockCookie lockCookie);
public LockCookie UpgradeToWriterLock(int timeoutMilliseconds);
public LockCookie UpgradeToWriterLock(TimeSpan timeout);
```

Due to issues noted before with potential deadlocks for simple shared-to-exclusive upgrades, when a call to `UpgradeToWriterLock` is made, the shared mode lock is first released. If the timeout expires, an `ApplicationException` will be thrown. Otherwise, the lock will have been released and a write lock will have been acquired. The method returns a `LockCookie`, which must be used to downgrade back to the recursive state that was present before the upgrade. It is not sufficient to call `ReleaseWriterLock`.

There is a subtle “gotcha” lurking here. Because the lock is released entirely during an upgrade, other writer threads may acquire the lock, mutate state, and so forth, before the upgrade completes. Therefore, once the thread performing the upgrade is granted the exclusive lock, it must always validate that a writer hasn’t snuck in and invalidated the state that was read leading up to the decision to upgrade. This is done with the lock’s `WriterSeqNum` property. Each time an exclusive lock is granted, this number is incremented. Therefore, a thread must read it before upgrading and validate that it hasn’t changed once it successfully upgrades the lock. This can be done by hand or with the `AnyWritersSince` method.

```
ReaderWriterLock rwl = ...;
... elsewhere ...
rwl.AcquireReaderLock(Timeout.Infinite);
try
{
    while (true)
    {
        if (... need to upgrade ...)
        {
            int seqNum = rwl.WriterSeqNum;
            LockCookie uc = rwl.UpgradeToWriterLock(Timeout.Infinite);
            try
            {
                if (rwl.AnyWritersSince(seqNum))
                {

```

```
        // A writer snuck in. Our decision to upgrade
        // may now be invalidated, so we try again.
        continue;
    }
    ... perform write operations ...
}
finally
{
    rwl.DowngradeFromWriterLock(ref uc);
}
}
break;
}

... perform read operations ...
}
finally
{
    rwl.ReleaseReaderLock();
}
```

You don't always have to retry the whole operation if a writer sneaks in during an upgrade, but it's usually necessary in order to preserve atomicity. This is one of the biggest problems with the upgrade feature of the old `ReaderWriterLock`: deciding whether atomicity is compromised by this behavior is a tricky and error prone process.

Debugging RWL Ownership

There is minimal SOS support for legacy RWLs. The `SOS !Threads` command has a `Lock Count` column in which the number of locks currently held by the thread is displayed. This number also takes into consideration RWL shared and exclusive lock ownership. Unlike CLR monitors, where the count excludes recursive acquisitions, the count does in fact include recursive RWL acquisitions.

If you need to get specific information about what threads currently own the RWL, short of spelunking in CLR internal data structures, there isn't much you can do. If you are inspecting the RWL from the thread that owns either a read or the write lock, the public `IsReaderLockHeld` and `IsWriterLockHeld` properties will report back a value of true accordingly. If you're not on the holding thread, the RWL has a private field `_dwWriterID` that contains the managed thread ID of the current writing thread. This is the best you can do. Lock reader information is hidden completely, managed by the

runtime, and not even exposed through the RWL data structure's private fields visible in Visual Studio.

Condition Variables

Now that we've looked at the data synchronization mechanisms on the platform, let's turn to those that are meant for control synchronization. This includes Windows Vista and CLR condition variables. These facilities, along with Windows events, are powerful enough to accommodate just about any control synchronization scenario you will encounter.

Windows Vista Condition Variables

Condition variables codify a very common control synchronization pattern. A thread often needs to wait for the establishment of some program specific condition. Verifying that this condition has been met involves evaluating a predicate, which in turn involves reading shared state. Because shared state is involved, it's important to use data synchronization. Moreover, if the condition has not yet been established, other threads will need to use data synchronization to ensure they safely modify state associated with the condition under evaluation.

There's a race condition inherent in exiting a critical region associated with data synchronization and waiting for the occurrence of an event. As we saw in the last chapter, Windows provides the `SignalObjectAndWait` API to signal an object and wait on another atomically for these very cases. But as soon as you use a critical section or SRWL, you can't access this feature because the synchronization mechanisms are hidden, that is, you cannot "release" the lock by signaling a kernel object; the user-mode lock itself controls all of this.

That's where the new Windows Vista condition variable feature comes in handy. It integrates with both critical sections and SRWLS to enable waiting and signaling on a logical condition variable related to a particular lock. As with critical sections, condition variables are local to a process and, as with SRWLS, they are extremely lightweight: each one is the size of a pointer, and uses keyed events as the sole waiting and signaling mechanism, meaning no allocation of separate kernel event objects is required.

Condition variables are also implemented primarily in user-mode and only have to incur kernel transitions when definitely waiting or signaling. The implementation is careful to minimize the number of such transitions. Note also that condition variables are the closest thing to raw access to Windows kernel keyed events.

A condition variable is represented by an instance of the `CONDITION_VARIABLE` data type. You can have any number of variables for any single lock, each representing a different abstract condition. The contents of the variable must be initialized before its first use, using the `InitializeConditionVariable` API. It takes an argument of type `PCCONDITION_VARIABLE` which is just a shortcut for `CONDITION_VARIABLE *`.

```
VOID WINAPI InitializeConditionVariable(
    PCCONDITION_VARIABLE ConditionVariable
);
```

And, just like SRWLocks, there are no related resources to free. So, aside from destroying the memory containing the variable, you do not need to take extra steps for de-allocation.

Sleeping and Waking

Once you have a condition variable initialized, you can begin coordinating among threads. When a thread has acquired a critical section or SRWL and subsequently decides that some condition has not yet been met, it can atomically release the lock and wait for another thread to wake it via the condition variable. This is done with the `SleepConditionVariableCS` or `SleepConditionVariableSRW` function, depending on whether the thread is using a critical section or SRWL, respectively.

```
BOOL WINAPI SleepConditionVariableCS(
    PCCONDITION_VARIABLE ConditionVariable,
    PCRITICAL_SECTION CriticalSection,
    DWORD dwMilliseconds
);
BOOL WINAPI SleepConditionVariableSRW(
    PCCONDITION_VARIABLE ConditionVariable,
    PSRWLOCK SRWLock,
    DWORD dwMilliseconds,
    ULONG Flags
);
```

When either function is called on a PCCONDITION_VARIABLE, the lock (either CriticalSection or SRWLock) is released and the thread begins waiting on the condition variable, atomically. This ensures no other thread can quickly acquire the lock and wake threads associated with the condition variable before they have been registered in the keyed event's internal wait list. If the SRWL is held in shared mode, you must pass the value CONDITION_VARIABLE_LOCKMODE_SHARED as Flags. As soon as the condition variable is signaled, the waiting thread will wake up and reacquire the lock before this function returns. Attempting to sleep by releasing a lock that has not been acquired results in the same behavior (explained earlier) of trying to erroneously release that particular kind of lock.

The timeout value, dwMilliseconds, is interpreted just like any other timeout, that is, -1 (INFINITE) indicates "no timeout." However, there's something interesting about the timeout for waiting on condition variables. Because the function won't return until the lock has been reacquired, the thread may actually have to wait to perform that acquisition after timing out but before returning. And there is no timeout for that acquisition. So while you may prevent the thread from waiting forever on the condition itself, there's no way to control the timeout for the subsequent wait on the lock needed in order to return.

When a thread enables the condition on which one or more threads may be waiting, it must wake them. There are two functions: WakeConditionVariable (wake-one) and WakeAllConditionVariable (wake-all). As their names imply, the first function wakes at most a single thread from the condition variable's wait list, while the second wakes up all threads that have begun waiting on the condition variable. These are very similar to auto-reset and manual-reset kernel event objects and can be used in similar circumstances:

```
VOID WINAPI WakeConditionVariable(
    PCCONDITION_VARIABLE ConditionVariable
);
VOID WINAPI WakeAllConditionVariable(
    PCCONDITION_VARIABLE ConditionVariable
);
```

It's not necessary to hold a lock when calling these APIs, though it's safer to do so. If you do not hold a lock, then threads adding themselves to

the wait list may miss a wake (for example, wake-all would miss a thread that enqueues itself immediately after the wake). Waking while the lock is held avoids these problematic cases. With that said, it also suffers from the problem mentioned in the previous chapter: awakened threads will immediately attempt to reacquire the lock held by the waker, and they will have to immediately rewait for the lock itself. This can be less efficient, but is often the only way to preserve correctness.

You must also be careful when it comes to lock recursion and condition variables. If you have recursively acquired a lock (either a critical section or a SRWL shared mode lock) prior to calling sleep on a condition variable, the lock will be released only once before waiting on the variable. While it is not necessary that the call to wake waiting threads associated with a condition variable happen inside of a critical region, it's common that a lock must be acquired in order to enable the condition on which threads are waiting. Accidentally holding on to the lock is, therefore, a great recipe for deadlock.

A Motivating Example: A Blocking Queue Data Structure with Condition Variables

In the previous chapter, we looked at how to build a queue that blocks callers when they try to take from an empty queue. There were some tricky cases that involved some amount of trading performance for correctness. We ended up with a solution that used a manual-reset event but that could regularly wake up more threads than there were elements. For instance, if we were in a case where many threads waited for items in the queue and yet the queue was constantly empty, we'd wake every thread anytime a single element arrived. This would cause problems, but at least ensured items would not get lost. Moreover, the implementation was not necessarily straightforward.

We can use condition variables to achieve the same level of correctness, but with much better performance. And the code is strikingly simple. We'll have a data structure, `BlockingQueueWithCondVar`, that is just comprised of three fields: a `CRITICAL_SECTION` to ensure data synchronization, a `CONDITION_VARIABLE` for threads to wait on when taking from a queue that is empty, and a `STL queue<T>` to hold the queue's contents.

```
#define _WIN32_WINNT 0x0600 // (New to Windows Vista)
#include <queue>
#include <windows.h>

template <class T>
class BlockingQueueWithCondVar
{
    CRITICAL_SECTION m_crst;
    CONDITION_VARIABLE m_nonEmptyVar;
    std::queue<T> * m_pQueue;

public:
    BlockingQueueWithCondVar()
    {
        InitializeCriticalSection(&m_crst);
        InitializeConditionVariable(&m_nonEmptyVar);
        m_pQueue = new std::queue<T>;
    }

    ~BlockingQueueWithCondVar()
    {
        delete m_pQueue;
        DeleteCriticalSection(&m_crst);
    }

    void Enqueue(T obj)
    {
        EnterCriticalSection(&m_crst);
        m_queue.push_front(obj);
        WakeConditionVariable(&m_nonEmptyVar);
        LeaveCriticalSection(&m_crst);
    }

    T Dequeue()
    {
        EnterCriticalSection(&m_crst);

        // Wait until the queue is non-empty.
        while (m_queue.empty())
            SleepConditionVariableCS(&m_nonEmptyVar, &m_crst, INFINITE);

        T obj = m_queue.pop_back();

        LeaveCriticalSection(&m_crst);
    }

    return obj;
};

};
```

This is fairly straightforward. We do some simple initialization inside of the constructor and de-allocation inside of the destructor, as you'd expect. When we enqueue a new element into the queue, we always wake a single waiter with `WakeConditionVariable`. The queue uses the wake-one variant because it issues a wake each time an element is enqueued. Because each waiter processes only a single element, it would be wasteful to wake any more than that. And the `Dequeue` function is similarly very simple: it just checks the queue for emptiness, in a loop, and waits on the condition variable whenever it finds that there are no elements to process. It will be subsequently awakened by a call to `Enqueue`, at which point it takes the element from the queue (inside of the critical region) and returns.

.NET Framework Monitors

The CLR also supports condition variables in a first-class way, and they are deeply integrated with the monitor mutual exclusion facilities described earlier. They are slightly less powerful than Windows Vista condition variables because each monitor contains only a single condition variable. While this doesn't cripple most scenarios, it can be a frustrating limitation at times.

Waiting and Pulsing

Using the `Monitor` class, any thread can wait on an object that has already been locked via one of the static `Wait` method's overloads.

```
public static bool Wait(object obj);
public static bool Wait(object obj, int millisecondsTimeout);
public static bool Wait(object obj, TimeSpan timeout);
```

Calling this method atomically enqueues the thread into the target monitor's wait list and releases the lock on the object. Before it returns, it will have reacquired the lock on the target monitor. Attempting to wait on an object for which the calling thread doesn't own a lock will result in a `SynchronizationLockException` being thrown from `Wait`.

As with all timeouts reviewed thus far, a value of `-1` (`Timeout.Infinite`) indicates that no timeout should be used—the default for the `Wait` overload

that only accepts an object argument. If the wait returns before the condition has arisen, the return value will be false, else it will be true. Note that the method must always reacquire the lock on obj before returning, which means it may have to wait, even if a timeout was used. The timeout supplied as an argument has no impact on this subsequent wait—that is, there is no way to specify a timeout.

A thread that enables the condition for which other threads may be waiting is responsible for invoking the appropriate wake method, either Pulse (wake-one) or PulseAll (wake-all).

```
public static void Pulse(object obj);
public static void PulseAll(object obj);
```

Unlike Windows condition variables, it is required that the lock be held on obj when calling Pulse or PulseAll. This means there is simply no way to avoid the problem with CLR monitors where a thread wakes up from the condition variable only to find that it must immediately wait to reacquire the lock on the object.

It is worth mentioning how condition variables are implemented on the CLR. Waiting on an object forces inflation of the object header (see the discussion earlier on how monitor locking is implemented if you don't know what this means). Inside the resulting sync block, there is a wait list that is maintained in FIFO order. Whenever a thread wishes to wait on a condition variable, it first enqueues a HANDLE to its own private per thread Windows event into this wait list; it then waits on this event. A wake-one dequeues the head and sets the event, while a wake-all walks the whole list and sets each event. Because each thread uses a single per thread event for this purpose, it isn't necessary to allocate multiple events to handle waiting on multiple condition variables throughout the life of a given thread.

A Motivating Example: A Blocking Queue Data Structure with Monitors

For completeness sake, here's an implementation of the blocking queue shown earlier that uses CLR monitors to achieve mutual exclusion and conditional waiting, rather than critical sections and Vista condition variables. Aside from the mechanisms used, the algorithm is identical.

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

class BlockingQueueWithCondVar<T>
{
    object m_syncLock = new object();
    Queue<T> m_queue = new Queue<T>();

    public void Enqueue(T obj)
    {
        lock (m_syncLock)
        {
            m_queue.Enqueue(obj);
            Monitor.Pulse(m_syncLock);
        }
    }

    public T Dequeue()
    {
        lock (m_syncLock)
        {
            // Wait until the queue is non-empty.
            while (m_queue.Count == 0)
                Monitor.Wait(m_syncLock);

            return m_queue.Dequeue();
        }
    }
}
```

Guarded Regions

Note that in all of the above examples, threads must be resilient to something called **spurious wake-ups**—code that uses condition variables should remain correct and lively even in cases where it is awoken prematurely, that is, before the condition being sought has been established. This is not because the implementation will actually do such things (although some implementations on other platforms like Java and Pthreads are known to do so), nor because code will wake threads intentionally when it's unnecessary, but rather due to the fact that there is no guarantee around when a thread that has been awakened will become scheduled. Condition variables are not fair. It's possible—and even likely—that another thread will acquire the associated lock and make the condition false again before

the awakened thread has a chance to reacquire the lock and return to the critical region. For a waiting thread, therefore, checking of the condition variable predicate should always occur inside of a loop, that is:

```
while (!P) { ... wait ... }
```

This pattern can be generalized into something called a **guarded region**. For example, imagine a fictitious API, `When`, to support this coding pattern with managed condition variables. It takes two delegates: one that represents the predicate that determines when the prerequisite condition has been met and the other that represents the work to be done inside of the critical region once the predicate evaluates to true.

```
public static class GuardedRegion
{
    public static T When<T>(
        this object obj, Func<bool> predicate, Func<T> body)
    {
        lock (obj)
        {
            while (!predicate())
                Monitor.Wait(obj);
            return body();
        }
    }
}
```

Using this very simple method, we could easily rewrite the `Dequeue` method from earlier more succinctly. Here's an example that uses C# lambdas for expressiveness.

```
public T Dequeue()
{
    return m_syncLock.When(
        () => m_queue.Count > 0,    // predicate
        () => m_queue.Dequeue());   // body of the critical region
}
```

Where Are We?

In this chapter, we looked at several useful synchronization mechanisms that raise the level of abstraction from the basic kernel objects we saw in the previous chapter. This included simple mutual exclusion locks, `CRITICAL_REGION`

in Win32 and Monitor's Enter, TryEnter, and Exit methods in .NET, reader/writer locks, SRWLock in Win32 and ReaderWriterLockSlim in .NET, and, finally, condition variable types used for control synchronization, CONDITION_VARIABLE in Win32 and Monitor's Wait, Pulse, and PulseAll methods in .NET. You can build some sophisticated stuff out of these.

Next we will turn to some more effective scheduling techniques using the Windows and CLR thread pools. A thread pool raises the level of abstraction over direct thread management, much like these primitives did over direct kernel object management. This higher level of abstraction will allow us to focus more on application and algorithmic concerns instead of scheduling ones.

FURTHER READING

- J. Duffy. Atomicity and Asynchronous Exceptions. Weblog article, <http://www.bluebytesoftware.com/blog/2005/03/19/AtomicityAndAsynchronousExceptions.aspx> (2005).
- J. Duffy. Windows Keyed Events, Critical Sections, and New Vista Synchronization Features. Weblog article, <http://www.bluebytesoftware.com/blog/2006/11/29/WindowsKeyedEventsCriticalSectionAndNewVistaSynchronizationFeatures.aspx> (2006).
- J. Duffy. CLR Monitors and Sync Blocks. Weblog article, <http://www.bluebytesoftware.com/blog/2007/06/24/CLRMonitorsAndSyncBlocks.aspx> (2007).
- C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, Vol. 17, No. 10 (1974).
- S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, Third Edition (Addison-Wesley, 2005).
- M. Pietrek and R. Osterlund. Threading: Break Free of Code Deadlocks in Critical Sections Under Windows. *MSDN Magazine* (2003).

7

Thread Pools

UNITS OF CONCURRENT work are often comparatively small, mostly independent, and often execute for a short period of time before producing results and going away. Creating a dedicated thread for each piece of work like this is a bad idea: there are sizeable runtime costs (both in time and space) paid for each thread that is created and destroyed. If we were to create a new thread for each task the system had to run, the cost of the actual computation itself would be dwarfed in no time. These impacts also include more time spent in the scheduler doing context switches once the number of threads exceeds the processor count, an impact to cache locality due to threads constantly having to move from one processor to another, and an increase in working set due to many threads accessing disjoint virtual memory pages actively at once.

If your goal is to attain some kind of performance benefit from using concurrency, then this approach will undoubtedly foil your plans: either by delivering worse performance than a single threaded version of your program that performs all tasks serially, or at the very least, dramatically reducing the observed benefits. Even if your application seems to scale for the time being with this scheme, it's unlikely that it would continue scaling as more tasks are added to the system. Even for long running concurrent tasks, or tasks that are not performance motivated, introducing too many threads into a process can add sizeable pressure on many precious system resources: the thread scheduler, the pagefile (needed by the virtual memory system to

back the thread stacks), kernel object count, nonpageable kernel memory, and so on.

Windows and the CLR both provide thread pool components that seek to minimize these costs and globally optimize a program's thread usage. They tackle one slice of the broader resource management problem head on—managing threads. There are still threads being used by the pool, but the costs associated with creating and deleting them is amortized over many work items run during the lifetime of the entire process, while simultaneously striking a careful and general purpose balance between fairness and throughput.

Thread Pools 101

The underlying idea is simple. Some number of threads are managed automatically by each thread pool. The number of threads is based on a combination of configuration and dynamic information about the runtime machine's capacity and load. Programs queue work items that should run concurrently and the thread pool makes sure the work gets done. To support this, the pool manages a few things: a work queue, a set of threads that dequeue and execute items from that queue, and the decisions about how to grow and shrink the set of threads and how to assign work to threads. In some sense, the thread pool is a cooperative scheduler that can throttle the amount of active work going on at once to avoid overhead due to preemptively scheduling work items that exceeds the number of processors available.

Most people are better off using a thread pool and forgetting most of what was explained in Chapter 3, Threads. Many of the difficult issues around thread lifetime and management are handled for you by the pool, and there are fewer things to get wrong. If you don't use a thread pool, you have to manage the global work throttling problem, which tends to be complicated. This is particularly true if your code is composed in the same process with other third party components that also use concurrency. Using a common thread pool helps to ensure thread resources are balanced appropriately.

Only if the thread pool path has proven to be ineffective should explicit threading even be explored. There are of course a few exceptions to this

rule of thumb, such as if you need to employ a high priority dedicated daemon thread to perform some special, important, and regularly occurring activity, and so on, but these cases are certainly exceptions rather than the rule. Whenever you find yourself creating a thread, ask: “Is there a way I could do this by using the thread pool instead?” You’ll be much happier in the end.

Three Ways: Windows Vista, Windows Legacy, and CLR

Since I’ve hyped up the thread pool quite a bit now, it’s probably time to look at some specific details. Both Windows and the CLR offer different variants of the thread pool idea that are entirely different components and provide different APIs. These disparate pool components are unaware of each other and, hence, can “fight” with one another for resources in the same process. The practical impact of this design isn’t terrible and only matters if you’re doing managed-native interop. The impact is that you could end up with twice the optimal number of threads.

Windows has offered a native thread pool since Windows 2000. Windows Vista comes with an entirely new architecture and implementation (where much of the logic has been moved into user-mode) and offers a newly refactored set of APIs, several new capabilities, and superior performance. Though the Vista pool is the preferred choice for any new native code, you will have to decide whether using the new Vista thread pool is worth sacrificing support for legacy OS platforms. If you need to run on Windows Server 2003 and/or Windows XP, for example, you’ll need to use the legacy thread pool APIs. These still exist in Windows Server 2008 and Vista for backwards compatibility. The old thread pool APIs on Vista have been reimplemented on top of the new ones, so even if you code to the legacy APIs you’ll see improved performance when moving to Windows Vista.

If you’re writing in managed code, you should use the CLR’s thread pool instead. The APIs are similar to the legacy native APIs. In fact, I encourage all readers, whether they are programming in native or managed code, to read this entire chapter. The CLR’s thread pool was a fork of the old Win32 thread pool, so many of the legacy problems that the Vista pool solves are currently present in managed code. While it’s certainly possible to P/Invoke to access the new Vista thread pool from managed

code, there are some problematic cases you would have to worry about. The native thread pool, for example, will not interoperate with the CLR's garbage collector (GC); the GC needs to block threads during a collection, which the thread pool will respond to by introducing additional threads to run work. This can lead to some interesting problems. There are bound to be other issues that you'd encounter by going down this path, so I would strongly advise against it.

I will also mention that a lot of people favor writing custom thread pools. (You will find one later in this chapter.) The reasons are numerous. The platform thread pools are black boxes to most people, and, when it comes to scheduling work, black boxes can be intimidating. You'd like to know precisely how and when work will run, and what decisions went into determining those things. This chapter should help to eliminate the mystery. Once you understand how the decisions are made, however, you might legitimately disagree with the policies. There are some features to control these decisions, but not enough to satisfy every requirement. One last reason people roll their own is that the thread pool idea, at face value, is fairly simple to understand, and writing one is a good way to get initiated to basic threading and synchronization concepts. I recommend that you recognize this as what it is: a learning exercise and not an attempt to build product quality code that you will ship.

If you decide, after much analysis, that you must write your own thread pool, just know that it can be extremely costly. It typically starts off looking very simple and, over time, grows in complexity as various corner cases are discovered. Reading this chapter should convince you of this. And you may introduce some odd interactions between yours and the other thread pools in the system along the way. Since many platform components implicitly use the existing pools, you're apt to end up in a resource battle with those other platform components.

In Chapter 12, Parallel Containers, we will examine some more advanced queuing mechanisms for thread pool style work management. Namely, we'll take a look at a highly efficient work stealing queue that does even better than the platform's thread pools for most cases. While this is an interesting topic from an I-have-to-know-everything-there-is-to-know-about-concurrency standpoint, the platform thread pools are suitable for almost

everybody who needs to write real programs. So don't turn up your nose just yet without even reading the pages that follow. If you do end up creating your own thread pool, however, that section is a must read.

Common Features

Each of the three thread pools—the Windows Vista, legacy Win32, and CLR thread pool—offer very similar functionality. There are a handful of features that any one pool offers over another, and some dramatic differences in the thread management policies and APIs used to access the features, but we'll cover how you access four basic features with each of the particular pools. These features are: work callbacks, I/O callbacks, timer callbacks, and wait registration callbacks. Let's review each at a high level before moving on.

Work Callbacks

The simplest functionality offered is the ability to queue a work callback to execute asynchronously on a thread pool thread. A single work callback maps directly to the notion of a concurrent task. In the case of native code, this callback is represented by a function pointer, and in managed code, a delegate; both also accept an optional state argument. The callback code pointer plus the state argument form a closure. Each of the thread pool implementations maintains its own queue of work and a set of threads dedicated to executing work. Queuing a work item places the callback into a queue that these threads monitor. Eventually one of them will see it, dequeue the callback, invoke it, and then go back for more. This is the least specialized and most frequently used feature of the pools.

I/O Callbacks

Each of the three thread pools integrates with asynchronous I/O to simplify management of completion callbacks. A completion callback is an application specific activity that needs to run when some asynchronous I/O operation finishes. This might include marshaling the bytes read into a program data structure, updating some UI display, or initiating the next asynchronous I/O operation in a longer sequence of I/O work to be done, for example. This feature relies on asynchronous I/O in Windows, and specifically the completion ports capability.

There are many interesting facets to asynchronous I/O on Windows, of which I/O completion ports and the thread pool's support are just two. Accessing completion ports solely through the thread pool, while convenient, doesn't expose all of the power of programming them directly. More on asynchronous I/O and a full overview of completion ports can be found in Chapter 15, Input and Output. Because we are getting slightly ahead of ourselves for the purpose of discussing the thread pool's support, many of the asynchronous I/Oisms will be kept fairly terse.

Some I/O operations on Windows—such as `ReadFile` or `WriteFile`—can be run asynchronously. This means that the program thread that makes the call can continue doing useful work concurrently while the I/O operation executes (because the API may return before the I/O has actually completed) versus the thread blocking for the I/O to complete (as would normally be the case for synchronous I/O). When the I/O finishes, the OS fires an interrupt that allows the program to respond to the I/O completion. Asynchronous I/O works closely with the device itself to operate in a truly asynchronous manner, typically leading to less blocking and improved scalability.

A few other methods of I/O completion are available on Windows, such as having the thread that spawned the I/O periodically poll for completion or wait on a `HANDLE` that is set by the asynchronous I/O interrupt handler. Another completion mechanism is the I/O completion port, which is what the thread pools use internally for their asynchronous I/O support.

The 10 second I/O completion port elevator pitch is as follows. One or more threads can wait for something called an I/O completion packet to be posted to a completion port. Individual file `HANDLEs` may be bound to the port, in which case anytime an asynchronous I/O operation for such a file `HANDLE` completes, a packet is automatically posted to the port by the OS. It's also possible to post packets to a completion port by hand. Whenever a packet is posted to the port, it is made available to one of the I/O threads, either by unblocking a waiting thread (if any) or by letting the thread that is already running ask for the next packet. The I/O completion port attempts to keep the number of threads that are actively processing I/O completion packets as close to a certain “concurrency level” as possible; this is, by default, set to the number of processors on the machine. Because completion ports are integrated with many facets of the

kernel, they are given intimate knowledge of events such as blocking in order to attain this goal.

Why does the thread pool need to be involved in this? Having an I/O completion port isn't enough. You need to also manage the threads that are waiting for packets, including deciding how and when to create or destroy them, and you also need to devise your own callback mechanism, since completion ports only hand back raw data packets. This is where the thread pool saves the day: it manages its own internal completion port and the threads bound to that port. This allows you take advantage of the thread pool's clever thread management heuristics, alleviates you from coming up with a custom callback scheme, and also, keeping with the theme of process-wide resource management, composes nicely with the other forms of work that can be scheduled to run on the thread pool.

Timers

It's common for a program to want to schedule work to occur at a certain point in the future, possibly on a recurring basis. Say we wanted to download some stock ticker information from a Web service once every minute. One way of implementing this would be to dedicate an entire thread to perform the download every minute: it would download the information, issue a `Sleep(60000)`, download some more information, and so on. This approach requires managing a separate thread just for this task. As we accumulate more and more services with similar needs, the design of giving each its own dedicated thread just doesn't scale. Moreover, timers can be much finer grained than 1 second, and the risk of multiple threads waking at once, leading to a wave of context switches, increases as more of these timer-like threads are created.

A better approach is to use Windows kernel timer objects. We reviewed those in the previous chapter. And we saw that, as with any other kernel object, you can wait on one with any of the wait APIs, including waiting for one of many such timers to expire (using a `WAIT_ANY` style wait), handle the timer event, readjust the expiration time, and then reissue the wait. But you would need to manage all of these timers yourself, which can be tricky, and for such a common task, you'd want the platform to offer some help.

And it does. The thread pool provides a way to schedule timer based callbacks. You specify the timing intervals, including the first occurrence

and the subsequent recurrence rate, and the thread pool takes care of the rest. This makes the task of managing outstanding timers, recurrences, and deciding which thread to run the callbacks quite simple. While a true kernel timer is used internally, there is only one, and the thread pool does the math to calculate its expiration time based on the next-to-expire timer's due time. The pool lazily allocates a thread to wait on this timer object and manages individually registered callbacks.

Registered Waits

Each pool gives you a way to register a callback that is to be invoked once a specific kernel object becomes signaled. In native code, this means specifying an object `HANDLE`, and in managed code this takes the form of specifying a `WaitHandle` object. Each of the pools allows you to assign a timeout during registration to limit the wait: the callback will still run in the case of a timeout, but the callback will be passed a flag so that it can respond differently.

Using this feature makes waiting for a large number of objects much more efficient. The thread pool places all registered objects into groups of `MAXIMUM_WAIT_OBJECTS - 1` (i.e., 63), assigns one dedicated wait thread per group, and has this thread wait for any of the registered objects to become signaled via a wait-any style wait. (One slot is used for a thread pool internal event, hence groupings of 63 instead of 64.) When one object becomes signaled, the wait thread wakes up, schedules the callback to run in the pool's work queue, possibly removes the awakened object from the wait set, and then goes back to waiting. As waits become satisfied and the number of active objects that a particular thread must wait for drops to zero, the thread exits. This is a bit like I/O completion ports and helps to build more scalable algorithms in a continuation-passing style.

Threads are anything but cheap on Windows. This point has been made enough times already. Imagine you need to wait for any of 1,024 objects to become signaled. The naïve approach of having a single thread per object results in 1,024 blocked threads. Not only is this bad from the standpoint of resource consumption, it's also extraordinarily dangerous. Imagine what might occur if every one of those objects became signaled at once or in close proximity to one another. Each thread would become runnable immediately. Various factors could make this situation even worse. Imagine if the objects were events and enjoyed priority boosts;

you'd have a massive wave of context switching and your program would likely suffer very severe performance degradation. Now compare this to using the registered waits feature of the thread pool. You would only need 17 threads (1,024/63) to perform the waits. And because the response to waking up is to queue a callback to the thread pool's work queue, you enjoy all of the scheduling benefits, including keeping the number of runnable threads in the process within a reasonable limit. The pool works as a throttle.

Even if your code uses a wait-any style wait to consolidate wait threads, you may run into the `MAXIMUM_WAIT_OBJECTS` limitation yourself. Using the thread pool's registered wait feature is a great way to scale beyond this barrier.

ASP.NET has a feature in the .NET Framework 2.0 called asynchronous pages that is covered in the next chapter. It allows you to offload an entire Web request to be resumed once an event is signaled. The implementation for asynchronous pages relies on this very feature.

With all of that said, registering wait callbacks can be difficult to use. It requires that you encapsulate the whole continuation of your work into a callback at the time you would like to block. This can be challenging, depending on how much knowledge you have about the rest of the call stack at the time you decide to wait and how much work must be done after the callback completes.

Windows Thread Pools

Now it's time to get into the details. First we'll go through the Windows thread pools and then the CLR thread pool. Because the Vista APIs have effectively superseded the old ones (hence my calling them "the legacy APIs" throughout this chapter), let's focus on those first. Many people must continue using or maintaining old code bases and/or must continue running on down-level OSs, so we'll review the legacy APIs immediately afterward.

Windows Vista Thread Pool

The Vista thread pool supports the aforementioned capabilities. It does all of this in a centralized fashion so all of these capabilities are efficiently

handled in the same process without competing for and negatively impacting each other's use of system resources.

Internally the Vista thread pool manages several threads. A subset of those threads is used to invoke callbacks, in FIFO order from a single callback queue, regardless of whether those callbacks originate from a direct call to the work item APIs or the thread pool internals (I/O completions, timer expirations, or registered waits). A single thread handles timer waits and expirations, and there is a single thread created for each group of 63 wait registrations that perform the actual waiting and dispatching of callbacks. When these need to run some callback, it is just queued to run on the other set of callback threads. As of Windows Vista, you can actually have multiple pools running in the same process, in which case each such pool has its own set of all of these threads managed independently of each other.

There is an important distinction between the Vista and legacy thread pools that will become apparent when we compare the APIs further. With the old thread pool, any callbacks that had to perform asynchronous I/O needed to get queued to a separate set of threads. That's because the pool reserved the right to retire ordinary callback threads while outstanding asynchronous I/O and APCs were running asynchronously with that thread, effectively canceling them. All of the threads in the Vista thread pool remain alive until asynchronous I/O operations and APCs have completed, so you need not worry about choosing one or the other.

Work Items

The most basic function that the thread pool performs is enabling you to queue a callback for execution, represented in native code by a function pointer and `LPVOID` pair. Submitting work to execute on a thread pool thread is fairly straightforward. The simplest way to do so is with the `TrySubmitThreadpoolCallback` API.

```
BOOL WINAPI TrySubmitThreadpoolCallback(
    PTP_SIMPLE_CALLBACK pfns,
    PVOID pv,
    PTP_CALLBACK_ENVIRON pcbe
);
```

The `pfns` argument is a pointer to a callback function that will be invoked on a thread running in the thread pool, and the `pv` argument is an optional state argument, passed as the callback's `Context` argument.

```
VOID CALLBACK SimpleCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context
);
```

The `callback_environment` argument, `pcbe`, allows you to control where, specifically, the work gets executed. For now we will always pass `NULL` and ignore callback environments completely, though they are quite useful and we will return to them later. The thread pool supplies the `Instance` argument to the callback, which is just a pointer to an internally managed thread pool data structure; this structure can be used as an input argument to various other APIs that manage state associated with the callback (as we'll see later).

After `TrySubmitThreadpoolWork` returns `TRUE`, the work has been enqueued into the work queue. The callback threads monitor this queue for new work, running inside a loop that continuously dequeues and executes items as quickly as possible. After our work item has been enqueued, any of the thread pool threads are apt to dequeue and execute the work. Which particular one happens to run the work and the precise timing of its execution are determined by a combination of the queue contents and what threads are doing at that particular point in time.

The `TrySubmitThreadpoolCallback` function can fail—hence the `Try` part of its name—in which case the function returns `FALSE` and `GetLastError` can be used to retrieve failure details. This is usually caused by insufficient memory to allocate the necessary internal data structures. This should rarely happen except for low resource situations. Nevertheless, it is possible and, thus, needs to be considered and handled.

Note that because all of the APIs in this section are new to Windows Vista, you will need to define `_WIN32_WINNT` to be `0x0600` before importing `Windows.h` to access them.

An Alternative Way to Submit Work. There is an alternative way to submit work items to the pool. It's a multi-step process instead of a single API

call, but gives you two additional capabilities: you can submit the same work item object multiple times, and you can easily wait for the submitted work to finish. The latter is a very useful feature, so you'll probably find yourself using this alternative approach quite often. The first step is to call the `CreateThreadpoolWork` API.

```
PTP_WORK WINAPI CreateThreadpoolWork(
    PTP_WORK_CALLBACK pfnwk,
    PVOID pv,
    PTP_CALLBACK_ENVIRON pcbe
);
```

You supply a function pointer representing the work to be done concurrently, a `PVOID` state argument, and, as with `TrySubmitThreadpoolWork`, an environment (for which we will pass `NULL` for now). It gives back a pointer to a newly allocated `TP_WORK` structure, which is then submitted for execution with the `SubmitThreadpoolWork` function.

```
VOID WINAPI SubmitThreadpoolWork(PTP_WORK pwk);
```

Notice the `pfnwk` callback type is `PTP_WORK_CALLBACK` rather than `PTP_SIMPLE_CALLBACK`, as was taken by `TrySubmitThreadpoolCallback`. The only difference between them is that you can now access the `TP_WORK` object from inside the callback, whereas the `TP_WORK` object was entirely hidden with the previous scheme.

```
VOID CALLBACK WorkCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PTP_WORK Work
);
```

`CreateThreadpoolWork` will return `NULL` if it wasn't able to allocate the `TP_WORK` data structure. Check `GetLastError` for failure details.

Somewhat cleverly, `SubmitThreadpoolWork` will not fail; this is because the internal data structures used to queue work rely on storage that has already been allocated by reusing memory in the `TP_WORK` structure to link submissions together. When I say it cannot fail, that's not entirely true: the API doesn't validate the `pwk` argument, so if you pass garbage to it, you're likely to see an AV or memory corruption.

If you submit the same TP_WORK for execution multiple times, each one will execute, possibly concurrently, using the same callback and context information supplied to `CreateThreadpoolWork`. You can't associate any unique data with the submission itself, which, in my opinion, would have been quite useful, though it probably would have made it more difficult to achieve the no-failure-possible feature of `SubmitThreadpoolWork`.

Since creating the TP_WORK object means that `CreateThreadpoolWork` allocates memory, this object must be freed once it is no longer in use. If you fail to free it, the TP_WORK's memory will be leaked. We'll see later how cleanup groups can be used as an alternative mechanism to clean up a whole set of such thread pool objects at once without needing to keep track of every one that was allocated (a little GC-like). For now, however, you will have to do this on an individual basis with the `CloseThreadpoolWork` API.

```
VOID WINAPI CloseThreadpoolWork(PTP_WORK pwk);
```

If there are outstanding submitted callbacks for the TP_WORK object at the time that `CloseThreadpoolWork` is called, the thread pool will note the request for deletion and defer the actual freeing operation until all associated callbacks finish. This is possible because internally the thread pool uses reference counting to track which threads are using the object, ensuring that memory is never freed prematurely. Thus, it's actually safe to close the object immediately after calling `SubmitThreadpoolWork` one or more times, or within the callback itself, alleviating a whole set of coordination issues that would have otherwise arisen.

With the `TrySubmitThreadpoolCallback` mechanism for creating work, you didn't need to worry about freeing any memory. It's not that there aren't any TP_WORK objects involved—there are—it's just that the thread pool internally handles allocating and freeing them at the appropriate times.

Waiting for Work to Finish. After you've queued up some work, it's quite common that you will need to block the thread waiting until all of the work has finished. We'll see many common patterns in Chapter 13, Data and Task Parallelism; for example, *fork/join* concurrency often involves a single master thread that spawns some number of children and then waits for them

to complete. The Vista thread pool makes this extremely simple with the `WaitForThreadpoolWorkCallbacks` API.

```
VOID WINAPI WaitForThreadpoolWorkCallbacks(
    PTP_WORK pwk,
    BOOL fCancelPendingCallbacks
);
```

Pass to this API a pointer to the `TP_WORK` object you'd like to wait for, and it will block the calling thread until all scheduled work associated with `pwk` completes (i.e., all calls to `SubmitThreadpoolWork`, in case there are multiple). This function doesn't validate its arguments and can fail or corrupt state if you pass an invalid `PTP_WORK` as `pwk`. This API blocks the calling thread using a non-alertable, non-message pumping wait.

If you pass `TRUE` for `fCancelPendingCallbacks`, any `pwk` work that is still in the thread pool's callback queue (i.e., hasn't begun executing yet) will be canceled and removed from the queue, subject to timing and the inherent race conditions involved. If all work is canceled successfully, the API may not need to wait before returning. Any work that is already executing cannot be canceled using this mechanism. Please refer to Chapter 13 for a more general discussion of cancellation.

If there is outstanding work in the thread pool's queue and all other threads in the system exit, the process will exit. This can lead to dropped work. In fact, if work is actively executing on thread pool threads while process exit is initiated, each of them is terminated right in its tracks without unwinding the stack (via `TerminateThread`). To prevent this, you need to synchronize process shutdown with the outstanding callbacks that are required to execute. One way of doing this is to use `WaitForThreadpoolWorkCallbacks` during your program's shutdown coordination code. If you do this, you must be very careful: you cannot pass a timeout to the API and holding up shutdown indefinitely is a recipe for problems.

If the callback running on a thread pool thread causes an exception that goes unhandled, the process will terminate via the ordinary unhandled exception logic described in Chapter 3, Threads. There is one special case in which the Vista thread pool catches an exception: stack overflow. If code running on a thread pool thread triggers a stack overflow, the thread pool

catches it, resets the guard page, and keeps the thread alive. And then it goes right back to the queue to find new work. Arguments can be made in both directions, but I believe that it's too bad the pool engages in this practice: it's potentially quite dangerous and can cause some problems down the road in the program's execution. Swallowing a stack overflow could be masking deeper problems such as state corruption that will only be made worse by trying to continue running. Crashing the process is a more conservative approach, and it's generally much easier to find and fix the cause of a crash than to find and fix random state corruption that becomes apparent at some undetermined pointer after the problem occurred. Moreover, resetting the guard page and continuing to reuse the thread for additional callbacks may lead to even stranger complications, since various thread local state may persist, including critical sections that are still owned by the thread, possibly leading to future work items seeing broken state invariants. Nevertheless, that's the way that it works.

A Simple Example Tying it All Together. Here is a really simple code example that demonstrates the common pattern of using `CreateThreadpoolWork`, `SubmitThreadpoolWork`, `WaitForThreadpoolWorkCallbacks`, and `CloseThreadpoolWork` to schedule work and then wait for it to complete. Clearly the code could become even simpler with `TrySubmitThreadpoolCallback`. But if we did that, we would have to devise our own mechanism for the primary thread to wait for the work to complete.

```
#include <stdio.h>
#define _WIN32_WINNT 0x0600
#include <windows.h>

volatile LONG s_dwCounter = 0;

VOID CALLBACK WorkCallback(
    PTP_CALLBACK_INSTANCE Instance, PVOID Context, PTP_WORK Work)
{
    printf("- Callback #%ld\t(ctx %s)\t(tid %u)\n",
        InterlockedIncrement(&s_dwCounter),
        reinterpret_cast<char *>(Context),
        GetCurrentThreadId());
}
```

```

int main(int argc, wchar_t * argv[])
{
    char str[] = "Hello, TP";

    PTP_WORK pwk = CreateThreadpoolWork(&WorkCallback, str, NULL);
    if (!pwk)
    {
        // Handle failure. GetLastErrorMessage has details.
    }

    // Submit 10 copies of this work to run concurrently.
    printf("- Submitting work...\n");
    for (int i = 0; i < 10; i++)
        SubmitThreadpoolWork(pwk);

    // Do something interesting for a while...

    // And then later wait for the work to finish.
    printf("- Waiting for work...\n");
    WaitForThreadpoolWorkCallbacks(pwk, FALSE);
    printf("- Work is finished.\n");

    CloseThreadpoolWork(pwk);

    return 0;
}

```

Each piece of work in this case prints the result of incrementing a shared counter `s_dwCounter`, the Context—which, in this case, is just a string held in `main`'s stack (this is safe, by the way, but only because we wait in `main` until all of the scheduled callbacks are finished running)—and the current thread pool thread's unique ID. Depending on whether you're on a single or multiprocessor machine and the thread pool's thread creation decisions, you may see numbers printed out of order and/or more than one thread ID.

Timers

Now let's see how to go about creating timers. As with `TP_WORK` objects for work callbacks, the first step to scheduling a thread pool timer for execution is to allocate a new `TP_TIMER` object with the `CreateThreadpoolTimer` function.

```

PTP_TIMER WINAPI CreateThreadpoolTimer(
    PTP_TIMER_CALLBACK pfnti,
    PVOID pv,
    PTP_CALLBACK_ENVIRON pcbe
);

```

In fact, aside from the difference in callback type (PTP_TIMER_CALLBACK instead of PTP_WORK_CALLBACK), the signature of CreateThreadpoolTimer is the same as CreateThreadpoolWork. And the only difference between the callback signatures is that the timer based one takes a PTP_TIMER rather than a PTP_WORK as its last argument.

```
VOID CALLBACK TimerCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PTP_TIMER Timer
);
```

The callback will be called by the thread pool whenever the timer expires, passing the original pv value from CreateThreadpoolTimer as the Context argument. At this point, we've only allocated a new TP_TIMER object: it hasn't actually been given any sort of expiration time or recurrence information, so it's not active yet. In fact, it isn't much of a timer just yet. To schedule it, we must call the SetThreadpoolTimer function.

```
VOID CALLBACK SetThreadpoolTimer(
    PTP_TIMER pti,
    PFILETIME pftDueTime,
    DWORD msPeriod,
    DWORD msWindowLength
);
```

It should be obvious what PTP_TIMER is: a pointer to the TP_TIMER object we just allocated. What follows are three bits of time information that determine how and when timer callbacks are triggered.

- **PFILETIME pftDueTime:** The time at which the timer will expire next. This can be specified as an absolute time, for example, midnight on 5/6/2027, or as a relative time, for example, 30 minutes and 23 seconds from the time at which SetThreadpoolTimer was invoked. Please refer back to Chapter 5, Windows Kernel Synchronization, where we reviewed in the context of waitable timers how to specify both relative and absolute times with a FILETIME structure.
- **DWORD msPeriod:** The number of milliseconds added to the current time to determine the next expiration time in a recurrence, performed automatically by the thread pool each time the timer expires.

This enables you to create recurring events. So, for example, if we created a timer with a due time of 5/6/2027 1:30 P.M. and a period of $(1000 * 60 * 60 * 24)$, the timer would expire on 5/6/2027 1:30 P.M., and then 5/7/2027 1:30 P.M., and so on, each time approximately 24 hours from the previous expiration. This parameter is optional: passing 0 indicates that this timer is a **one-shot timer** and that after the expiration at `pftDueTime` the timer won't fire anymore. Otherwise, this is a **recurring timer**.

- `DWORD msWindowLength`: An optional amount of delay, in milliseconds, which is acceptable between the timer expiration time and the actual callback execution time. Pass 0 if you do not care. If the thread pool gets behind running callbacks due to system load, for example, or a number of timers are set to expire very close in proximity to one another, then specifying a non-0 window length allows the thread pool to dispatch all of those expirations with overlapping expiration times (taking into account the window) all at once.

You can call `SetThreadpoolTimer` on the same timer object multiple times. This has the effect of changing the existing timer's schedule. No matter the current state, the next time the timer will fire is governed by the new `pftDueTime`. If the timer is already a recurring timer, then subsequent recurrences will be based on the new `msPeriod`, including turning the recurring timer into a one-shot timer if `msPeriod` is specified as 0. If the timer is a one-shot timer and has already fired, it will be rescheduled based on the new times.

Closing and Stopping Timers. Just as with `TP_WORK` objects, the `TP_TIMER` objects returned from `CreateThreadpoolTimer` must be deleted when you are finished with them. This is done with the `CloseThreadpoolTimer` function.

```
VOID WINAPI CloseThreadpoolTimer(PTP_TIMER pti);
```

We've seen that you can create a one-shot timer or a recurring timer. If you choose to create a recurring timer, it will keep firing indefinitely until you explicitly stop it. There are two ways to stop an already registered timer from firing. One is to make a call to `SetThreadpoolTimer` with a NULL value

for the `pftDueTime` argument. (Or, alternatively, specify a real `pftDueTime` but pass `0` for the `msPeriod`, in which case it will fire only once more.) Alternatively, you can just close the timer with a call to `CloseThreadpoolTimer`, which also stops the timer from expiring in the future. In both cases, there may be callbacks queued to execute, and stopping the timer doesn't prevent those from executing: it only prevents future callbacks for the particular timer from being generated.

A recurring timer's next expiration date is set at the time the timer actually expires, not when the timer's callback finishes working. Imagine, for instance, that you have a timer that expires every 10 milliseconds and whose callback takes 20 milliseconds to run; there will be a never-ending backlog of timer callbacks to execute in this scheme. If you want the timer's expiration time to be set based on when the timer callback finishes—which for this example is a bit like setting the timer's recurrence to 30 milliseconds—then you must queue your timer as a one-shot timer (i.e., `0` for `msPeriod`) and then make a call to `SetThreadpoolTimer` at the end of the callback routine to keep it going.

Timer Internals. Timers are implemented with a single process-wide thread, created the first time a timer is registered in the process. There is a single kernel waitable timer object. This thread sits in a loop, calculates how long it should wait based on the next-to-expire timer, sets the kernel timer object's expiration time, and then waits. When it wakes up, it queues the timer callback to run on one of the work callback threads and updates that particular timer's expiration time (for recurring timers) or removes the timer from its wait list (for one-shot timers) and then goes back to waiting.

If you think about this scheme for a moment, you will realize why the `msWindowLength` argument to `CreateThreadpoolTimer` can make a difference for performance. If many timers expire close together, but not quite at the same time, then the pool will have to continuously sleep and wake back up for very small periods of time, creating substantial context switching overhead. Permitting the pool to lump expirations together can improve the performance of timer dispatch dramatically.

Waiting for Timer Callbacks to Complete. As with work item callbacks, you can wait for all outstanding timer callbacks that have

been queued due to a particular TP_TIMER object to complete with `WaitForThreadpoolTimerCallbacks`.

```
VOID WINAPI WaitForThreadpoolTimerCallbacks(
    PTP_TIMER pti,
    BOOL fCancelPendingCallbacks
);
```

Specifying TRUE for `fCancelPendingCallbacks` will ensure that existing callbacks that are in the queue do not fire. And as with the other waiting mechanisms reviewed for the other callback timers, it does nothing for already executing callbacks. If you are using `WaitForThreadpoolTimerCallbacks` to synchronize the release of resources that those callbacks may require (such as dynamically loaded DLLs or kernel objects), then you should ensure the timer is disabled before waiting for existing callbacks to complete. If you do this in the reverse order, additional expiration callbacks may get created after the wait returns.

I/O Completion Ports

There are a few asynchronous I/O API specific steps you must take before scheduling an I/O callback to the thread pool. If you want to issue an asynchronous `WriteFile`, for example, you must first ensure that your call to `CreateFile` includes the `FILE_FLAG_OVERLAPPED` flag in the `dwFlagsAndAttributes` argument. You then must allocate an `OVERLAPPED` structure and pass its address into the call to `WriteFile`. All asynchronous I/O on Windows works in this same basic way. Winsock, for example, permits you to pass a `WSA_FLAG_OVERLAPPED` flag to `WSASocket` and provide a pointer to a `WSAOVERLAPPED` structure (which extends `OVERLAPPED` with some socket specific fields) to specify asynchronous socket operations like `WSASend` and `WSARecv`. The Windows thread pool only accommodates asynchronous file I/O, so you'll have to wait until Chapter 15, Input and Output, to learn more about asynchronous file and network I/O, and I/O completion ports more generally.

When you ask the thread pool to run some callback when I/O completes, you first specify the `HANDLE`, representing the object opened for overlapped I/O, and whose I/O completions should be handled via the particular callback. (In the case of asynchronous sockets, you must cast the `SOCKET` to a `HANDLE` when passing it.) This is done with the `CreateThreadpoolIo` function

which (unsurprisingly) looks quite a bit like the other Create APIs we've reviewed.

```
PTP_IO WINAPI CreateThreadpoolIo(
    HANDLE f1,
    PTP_WIN32_IO_CALLBACK pfnio,
    PVOID pv,
    PTP_CALLBACK_ENVIRON pcbe
);
```

The `f1` argument is the `HANDLE` opened for asynchronous I/O and `pfnio` is a pointer to the callback routine called in response to completions on it. The `pv` argument is an opaque value that is passed along to the callback. This function can fail, for example if allocating the `TP_IO` object cannot succeed due to insufficient memory, in which case the return value is `NULL`.

The `PTP_WIN32_IO_CALLBACK` callback function pointer refers to a function with the following signature:

```
VOID CALLBACK IoCompletionCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PVOID Overlapped,
    ULONG IoResult,
    ULONG_PTR NumberOfBytesTransferred,
    PTP_IO Io
);
```

The thread pool supplies the `Instance` pointer, which can be passed to various other routines that we'll see later. `Io` is just a pointer to the original `TP_IO` object queued to the pool, and `Context` is the `pv` argument to `CreateThreadpoolIo`. The rest is very I/O specific: `Overlapped` is the pointer to an `OVERLAPPED` structure specified during a call to the asynchronous API (say, `WriteFile`), `IoResult` contains the result of the I/O operation (`NO_ERROR` if it was successful), and `NumberOfBytesTransferred` specifies how many bytes were transferred during the operation (read or written), as the name implies. Notice that we didn't have to actually pass a pointer to the `OVERLAPPED` structure we're using when we made the call to `CreateThreadpoolIo`. This is all taken care of internally by the asynchronous I/O mechanisms themselves, and, in fact, the `OVERLAPPED` used for any given `HANDLE` can change from one operation to the next because you can change which `OVERLAPPED` object you use and/or perform many asynchronous operations simultaneously.

Callbacks will not fire until you make a call to the `StartThreadpoolIo` routine, passing a pointer to your newly created `TP_IO` object. In fact, if you begin any asynchronous operations on the specific `HANDLE` in between the call to `CreateThreadpoolIo` and `StartThreadpoolIo`, the `OVERLAPPED` structure may become corrupted, so don't do that.

```
VOID WINAPI StartThreadpoolIo(PTP_IO pio);
```

After this call, it's safe to start asynchronous I/O operations on the `HANDLE`. Whenever an I/O operation completes, the thread pool will run your `pfnio` callback just as it would any other work item queued to the thread pool. From within this callback, it is safe to begin additional asynchronous I/O without any special measures taken.

Once you're done issuing I/O operations against a particular file or socket, you should free its associated `TP_IO` object with a call to `CloseThreadpoolIo`.

```
VOID WINAPI CloseThreadpoolIo(PTP_IO pio);
```

Finally, much like the `WaitForThreadpoolWorkCallbacks` function, you can wait for all callbacks associated with a particular `TP_IO` object to finish using the `WaitForThreadpoolIoCallbacks` routine.

```
VOID WINAPI WaitForThreadpoolIoCallbacks(
    PTP_IO pio,
    BOOL fCancelPendingCallbacks
);
```

Just as with waiting for ordinary worker callbacks, you may optionally cancel any of them that have been queued to execute but have not yet executed by passing `TRUE` for `fCancelPendingCallbacks`. Canceling callbacks does nothing with actively executing callbacks, nor does it prevent subsequent asynchronous I/O completions from creating new ones.

Registered Waits

To register a wait notification for the Vista thread pool, you must first create a wait object with the `CreateThreadpoolWait` API. At this point, you're probably very familiar with this pattern.

```
PTP_WAIT WINAPI CreateThreadpoolWait(
    PTP_WAIT_CALLBACK pfnwa,
    PVOID pv,
    PTP_CALLBACK_ENVIRON pcbe
);
```

The `pfnwa` argument to `CreateThreadpoolWait` is a pointer to a wait callback, which is typededefed as a pointer to a function with the following signature.

```
VOID CALLBACK WaitCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PTP_WAIT Wait,
    TP_WAIT_RESULT WaitResult
);
```

The `pv` argument to `CreateThreadpoolWait` specifies the opaque context pointer that will be passed to your callback as the `Context` argument when the wait condition is satisfied. It will never dereference this memory—it's yours. Creation of the wait object allocates memory and returns a pointer to it, which also means the function can fail and return `NULL`.

After creating a wait object, no waits have been registered with the thread pool yet. To do that, you must tell the pool about the `TP_WAIT` object and the `HANDLE` for the object for which you'd like to wait to become signaled. This is done with `SetThreadpoolWait`.

```
VOID WINAPI SetThreadpoolWait(
    PTP_WAIT pwa,
    HANDLE h,
    PFILETIME pftTimeout
);
```

Once you call this function, the thread pool will move the newly registered `HANDLE` on to one of its wait threads. If all current threads are waiting on 63 objects already, then a new thread will be spun up. After this happens, as soon as `h` becomes signaled, the callback associated with `pwa` will be queued to run on one of the pool's callback threads. All of the usual kernel object wait rules apply: that is, auto-reset events being reset when the wait is satisfied, only one thread being awakened, and so on. You may call `SetThreadpoolWait` on the same `TP_WAIT` object as many times as you please for any number of unique `HANDLE`s.

You can also supply a timeout when registering a callback. Passing `NULL` for `pftTimeout` means that no timeout is required. Timeouts here use the same `FILETIME` scheme as described for timers: a negative value indicates that the timeout is relative to the current time, while any other value

represents the absolute time at which the wait will expire. When a timeout occurs, your callback will still execute and the `WaitResult` argument to the `WaitCallback` routine will be `WAIT_TIMEOUT` rather than the usual `WAIT_OBJECT_0`. If one of the objects being waited on is a mutex that was abandoned, the `WaitResult` will be `WAIT_ABANDONED_0`. (Registering a wait for a mutex is an extraordinarily bad idea due to thread affinity, as we'll see in more detail shortly.)

If you call `SetThreadpoolWait` multiple times with the same `TP_WAIT` object, the last call will override previous calls. If the new value of `h` is `NULL`, no waits will be associated with the `TP_WAIT` object after the call to `SetThreadpoolWait` is complete. If `NULL` is specified, or if a new `HANDLE` value is provided, the thread pool internally notifies the thread waiting on the previously specified `HANDLE` and it is removed from its wait set.

Once a callback has occurred for a particular `HANDLE`, that object is removed from the thread's wait set. If you'd like to register another callback to occur when the kernel object becomes signaled again, you can make a call to `SetThreadpoolWait` in your callback.

```
HANDLE myWaitObject = ...;

PTP_WAIT myWait = CreateThreadpoolWait(&MyWaitCallback, ..., ...);
SetThreadpoolWait(myWait, myWaitObject, ...);

// Elsewhere ...
VOID CALLBACK MyWaitCallback(
    PTP_CALLBACK_INSTANCE Instance, PVOID Context,
    PTP_WAIT Wait, TP_WAIT_RESULT WaitResult)
{
    // Immediately re-register another wait.
    SetThreadpoolWait(Wait, myWaitObject, ...);

    // Handle the event ...
}
```

Specifying a mutex as a registration's object is usually a bad idea. Mutexes have thread affinity, meaning that the wait thread that performs a wait will be considered the owner of the mutex. But in this case, all the wait thread does is turn around and queue the callback to run on a thread pool callback thread. The thread that will run the callback doesn't own the mutex at all and therefore cannot release it. There is no way to work around

this with the Vista thread pool. We'll see later that the legacy APIs offer a way to deal with this.

Finally, once you are done with a wait, you must de-allocate its associated memory and resources. This is done with `CloseThreadpoolWait`.

```
VOID WINAPI CloseThreadpoolWait(PTP_WAIT pwa);
```

If there are outstanding callbacks executing for this wait object, they will be permitted to finish before the `TP_WAIT` memory is freed. If there are no callbacks running, but a thread is waiting on a registered `HANDLE` associated with this `TP_WAIT` object, the thread will be notified and it will wake up and remove the `HANDLE` from its wait set.

You can use the `WaitForThreadpoolWaitCallbacks` function to wait for any callbacks that are in-flight to finish executing.

```
VOID WINAPI WaitForThreadpoolWaitCallbacks(
    PTP_WAIT pwa,
    BOOL fCancelPendingCallbacks
);
```

If `fCancelPendingCallbacks` is TRUE, then any callbacks that have not yet begun executing will be canceled. This does not wait for the wait associated with the `TP_WAIT` object to be satisfied or for it to timeout, it merely ensures any existing callbacks are completed. For the same reason, you must be careful with timers and synchronizing the release of resources that a callback will use. You must also be careful with wait registrations because they may be satisfied immediately after your wait returns.

The ordinary `CreateThreadpoolWait`, `SetThreadpoolWait`, and `CloseThreadpoolWait` sequence can be illustrated by this code sample. We allocate a set of events, register waits for them all, and sit in a loop signaling them for a little while. Error checking is omitted for brevity. We also don't synchronize with the completion of wait registrations and callbacks—we'll discuss why in just a moment.

```
#include <stdio.h>
#define _WIN32_WINNT 0x0600
#include <windows.h>

const int g_cEvents = 8;
HANDLE g_hEvent[g_cEvents];
```

```

void InitFileTimeWithMs(PFILETIME pft, DWORD dwMilliseconds)
{
    LARGE_INTEGER cv;
    cv.QuadPart = -((LONG64)dwMilliseconds * 1000 * 10);
    pft->dwLowDateTime = cv.LowPart;
    pft->dwHighDateTime = cv.HighPart;
}

VOID CALLBACK WaitCallback(
    PTP_CALLBACK_INSTANCE Instance, PVOID Context,
    PTP_WAIT Wait, TP_WAIT_RESULT WaitResult)
{
    UINT_PTR i = reinterpret_cast<UINT_PTR>(Context);

    // Print some interesting info.
    printf("Wait: result = %u, event=%#p (tid = %u)\n",
        WaitResult,
        reinterpret_cast<UINT_PTR>(Context),
        GetCurrentThreadId());
}

int main(int argc, wchar_t * argv[])
{
    // Initialize auto-reset events.
    for (int i = 0; i < g_cEvents; i++)
        g_hEvent[i] = CreateEvent(NULL, FALSE, FALSE, NULL);

    FILETIME ft;
    InitFileTimeWithMs(&ft, 500);

    // Create and register 100 waits per event.
    const int g_cWaits = g_cEvents * 100;
    PTP_WAIT waits[g_cWaits];
    for (int i = 0; i < g_cWaits; i++)
    {
        UINT_PTR event = (UINT_PTR)i % g_cEvents;
        waits[i] = CreateThreadpoolWait(
            &WaitCallback, reinterpret_cast<PVOID>(event), NULL);
        SetThreadpoolWait(waits[i], g_hEvent[event], &ft);
    }

    // Go through and set the events a bunch of times.
    for (int i = 0; i < 50; i++)
        for (int j = 0; j < g_cEvents; j++)
            SetEvent(g_hEvent[j]);

    // Close everything (w/out waiting for callbacks).
    for (int i = 0; i < g_cWaits; i++)
        CloseThreadpoolWait(waits[i]);
}

```

```
    for (int i = 0; i < g_cEvents; i++)
        CloseHandle(g_hEvent[i]);

    return 0;
}
```

Tricky Synchronization with Callback Completion

Synchronizing with callback completion for I/O, timer, and wait registration completion is harder than it might appear at first glance. Moreover, we mentioned earlier that it's sometimes a good idea to reregister such a registration recursively from within its callback. This is particularly true of timers and wait registrations. (This is especially true of the latter given that it's the only way to create a registration that continues to persist after an object has been signaled once.) All of this creates a synchronization pitfall.

If you have threads that wait for callbacks to finish, close the object, and then move on thinking that no additional callbacks will finish, you will get burned. Take wait registrations as an example. Imagine one thread makes a call to `WaitForThreadpoolWaitCallbacks` and then `CloseThreadpoolWait`; afterwards it might go on to free a DLL or de-allocate a resource that the wait's callback uses. The naïve, and incorrect, approach might be:

```
PTP_WAIT myWait = CreateThreadpoolWait(...);
SetThreadpoolWait(myWait, realHandle, ...);

// ...

WaitForThreadpoolWaitCallbacks(myWait, FALSE);
CloseThreadpoolWait(myWait);
// free the resources now...
```

This is inviting disaster. Even though we waited for all callbacks to complete, additional callbacks could be queued after the call to `WaitForThreadpoolWaitCallbacks` but before the call to `CloseThreadpoolWait` (which, recall, removes the registration). In this case, we may move on to freeing resources concurrently with our callback as it executes. This kind of tricky race condition would undoubtedly be very difficult to find and fix.

The solution is to use a three-step process. In the case of wait registrations, that entails: (1) cancel the waits, (2) wait for callbacks to finish, and finally (3) close the wait object. (This works similarly for timers.)

Keeping with the original example above, that might look a bit like the following.

```
PTP_WAIT myWait = CreateThreadpoolWait(...);
SetThreadpoolWait(myWait, realHandle, ...);

// ...

SetThreadpoolWait(myWait, NULL, NULL); // Step 1: cancel the waits.
WaitForThreadpoolWaitCallbacks(myWait, FALSE); // Step 2: wait.
CloseThreadpoolWait(myWait); // Step 3: close the wait object.
// free the resources now...
```

Using cleanup groups also helps with this situation: closing a cleanup group does all of this in its implementation so that when it returns we can be sure that no subsequent callbacks will execute. That brings us to our next topic: thread pool environments.

Thread Pool Environments

Environments have been mentioned in passing a number of times, as several of the APIs described earlier allow you to pass in a pointer to one. Up to this point, we've always been passing **NULL**. But allocating and supplying a pointer to a true thread pool environment allows you to control various policies surrounding the execution of callbacks and to operate on a logical grouping of work rather than individual callbacks. Specifically, you can do the following.

- Isolate a group of callbacks from all other callbacks in the process.
- Perform cleanup work when all work associated with an environment completes. This includes an ability to have the thread pool call some arbitrary application specific cleanup callback in addition to automatically freeing the various thread pool data structures that were allocated for that environment.
- Wait for and/or cancel all outstanding (and not currently executing) work associated with a particular environment. This allows you to synchronize unloading a DLL or cleaning up particular resources when all thread pool work, which might use it, finishes. This covers ordinary work callbacks as well as I/O, timer, and wait registration callbacks, in addition to the associated registrations.

The feature described by the first bullet is possible because you can create separate pool objects, and the second and third both depend on a separate thing called a **cleanup group**. Before doing any of this, however, you need to first initialize an environment object with the `InitializeThreadpoolEnvironment` function. Unlike the creation APIs we've seen earlier, this function doesn't dynamically allocate the object—you pass a pointer to a memory location and it will initialize its contents. The environment must be destroyed later with `DestroyThreadpoolEnvironment`.

```
VOID InitializeThreadpoolEnvironment(PTP_CALLBACK_ENVIRON pcbe);  
VOID DestroyThreadpoolEnvironment(PTP_CALLBACK_ENVIRON pcbe);
```

Each takes a pointer to a `TP_CALLBACK_ENVIRON` block of memory and initializes or destroys the target memory's contents, respectively.

Creating Isolated, Dedicated Pools. Each process has one default Vista thread pool inside of it. Any work created with a `NULL` argument for the callback environment, as shown earlier, will go into this default pool's process-wide shared queue and will be serviced by a process-wide shared set of threads. This sharing applies within all processes, including those that host many in-process components (such as `svchost.exe`). The fact that this intimate level of sharing happens can cause problems for some components, particularly because some may queue work at an uneven rate. For example, one "chatty" component that queues many small work items can starve another component that queues work less frequently and in coarser chunks. Because the queue is serviced in FIFO order, this isn't always an issue; but the mere possibility that unpredictable wait times may occur is enough to concern many developers.

As of Vista, you can now create multiple pools inside the same process. Each pool has its own work queue and manages its own set of worker threads. This allows you to isolate components from one another so that the normal Windows preemptive scheduling can create some sort of fairness and can deal with possible starvation, albeit at the cost of having more threads in the system and possibly incurring more context switches. The thread pool thread creation and retirement policies do not change at all when you have multiple pools in the same process; in other words, they are unaware of each other, and each will be greedy and try

to use as many processors as possible. This can certainly cause performance anomalies, but the benefits from being able to isolate components from one another sometimes outweigh this risk.

To create a new pool, call the `CreateThreadpool` function.

```
PTP_POOL WINAPI CreateThreadpool(PVOID reserved);
```

After creating the pool, you will need to associate it with a callback environment.

```
VOID SetThreadpoolCallbackPool(
    PTP_CALLBACK_ENVIRON pcbe,
    PTP_POOL ptpp
);
```

After making this call, all subsequent work items that are scheduled for execution through the specified callback environment `pcbe` will execute in the new pool.

As with the other thread pool objects we've looked at so far, you also need to free the object when it's no longer in use. This is done with the `CloseThreadpool` function.

```
VOID WINAPI CloseThreadpool(PTP_POOL ptpp);
```

If there is work actively executing in the target thread pool, freeing will take place after all of the work completes. If there are work items in the pool that have not yet been scheduled for execution, they are canceled and will never execute.

Once you have a separate thread pool object, you can also set separate minimum and maximum thread counts on it. We'll describe the ordinary default thread creation and deletion policies later, but the minimum is the smallest number of active threads the thread pool will keep on hand, and the maximum is the most it will create to service work. The default minimum is 0 and the default maximum is 500. (The value of 500 was chosen for legacy compatibility with the pre-Vista thread pool infrastructure. For machines with more than 500 processors, this is a poor default, but at the time of this writing, such machines are not yet commonplace.) You can change these for a custom thread pool with the `SetThreadpoolThreadMinimum` and `SetThreadpoolThreadMaximum` functions.

```
BOOL WINAPI SetThreadpoolThreadMinimum(PTP_POOL ptpp, DWORD cthrdMic);
VOID WINAPI SetThreadpoolThreadMaximum(PTP_POOL ptpp, DWORD cthrdMost);
```

The `SetThreadpoolThreadMinimum` function can fail, in which case it returns FALSE, because it actually attempts to allocate enough threads to satisfy the minimum. Once it has returned successfully, there is at least the minimum number of threads specified running in the thread pool.

Note that it is not possible to alter the default thread pool's minimum and maximum count; instead, you must specify a pointer to a custom `TP_POOL` object. Prior to Vista, you could change the process-wide default pool's maximum (as we see later). The reason this capability has been removed is because it depends on races: the last component to call the API would win. This can cause conflicts between components in the same process that are unaware of each other but want different maximum or minimum values.

Cleanup Groups. Whenever a thread pool object is returned from one of the APIs we've reviewed above, it must later be cleaned up with the respective close function. This point has probably already been driven home simply. However, the thread pool offers a feature called cleanup groups, which allows you to cleanup all such objects that have been associated with a particular environment with one API call. This takes advantage of the fact that all of these objects are reference counted internally. Cleanup groups also allow you to specify a callback that will get invoked when either the group is being freed or work in the queue is canceled, providing an opportunity for you to free any arbitrary state that is used by callbacks within the group.

The first step to using a cleanup group is to call `CreateThreadpoolCleanupGroup`.

```
PTP_CLEANUP_GROUP WINAPI CreateThreadpoolCleanupGroup();
```

This allocates a new `TP_CLEANUP_GROUP` structure and returns a pointer to it. If allocation of the data structure fails, `NULL` is returned, and, as usual, `GetLastError` can be used to retrieve details. The group is not used at all until you associate it with an environment.

```
VOID SetThreadpoolCallbackCleanupGroup(
    PTP_CALLBACK_ENVIRON pcbe,
    PTP_CLEANUP_GROUP ptpcg,
    PTP_CLEANUP_GROUP_CANCEL_CALLBACK pfng
);
```

The callback pfng is optional and is a function pointer of type.

```
VOID CALLBACK CleanupGroupCancelCallback(
    PVOID ObjectContext,
    PVOID CleanupContext
);
```

If specified, the pfng callback will be invoked once a call to `CloseThreadpoolCleanupGroupMembers` has been made (more on that momentarily). This provides a hook for any sort of custom application specific cleanup logic, for example freeing memory used by all callbacks within a particular group. For those familiar with garbage collection based systems, this functionality is a bit like a finalizer for the whole cleanup group.

To actually initiate the cleanup, which includes waiting for all (and possibly canceling any outstanding) callbacks and running the pfng callback (if specified), you can make a call to the `CloseThreadpoolCleanupGroupMembers` function.

```
VOID WINAPI CloseThreadpoolCleanupGroupMembers(
    PTP_CLEANUP_GROUP ptpcg,
    BOOL fCancelPendingCallbacks,
    PVOID pvCleanupContext
);
```

This will return once all of ptpcg's callbacks are either completed or canceled. If `fCancelPendingCallbacks` is FALSE, the function must wait for any pending callbacks to get scheduled and to finish running. Otherwise, if it's TRUE, callbacks that haven't been scheduled yet will be removed from the queue and will never execute. The `pvCleanupContext` pointer is some application specific opaque value that is passed to the `CleanupGroupCancelCallback` as its `CleanupContext` argument.

This API is similar to the `WaitForThreadpoolWorkCallbacks` and related APIs we looked at above, but is more convenient for a number of reasons. To start with, you needn't track all of the individual thread pool objects by hand, which you would have had to do with the individual wait functions. Additionally, this synchronizes with timer expirations and wait registrations so you can be assured all outstanding callbacks have completed and that no additional callbacks will be created for these objects in the future.

Perhaps the most common need for `CloseThreadpoolCleanupGroupMembers` is to synchronize DLL unloading. If you have written a service

that uses the thread pool and a subsequent shutdown causes an important DLL to be unloaded, you must be careful that work hasn't been queued to the thread pool that will subsequently try to use that DLL. Having the service use a cleanup group and close that before unloading the DLL is a simple way of dealing with this coordination, whereas without it you'd have to do it all by hand. Similarly if you have memory or OS resources that are shared among callbacks, you need to ensure additional callbacks do not attempt to run after or during the release of those resources.

Once all of the members have been cleaned up, you can go ahead and close the group, which de-allocates the memory and resources associated with it. This is done with the `CloseThreadpoolCleanupGroup` routine.

```
VOID WINAPI CloseThreadpoolCleanupGroup(PTP_CLEANUP_GROUP ptpcg);
```

Finally, the `DisassociateCurrentThreadFromCallback` function allows you to explicitly unblock any threads waiting for callbacks with any of the wait APIs for a particular object, assuming the current callback is the last one for the specific object. While this unblocks threads waiting with APIs like `WaitForThreadpoolWorkCallbacks`, it does not unblock those waiting for the cleanup group members to complete, which allows the callback to continue using DLLs that such waiters will subsequently unload.

```
VOID WINAPI DisassociateCurrentThreadFromCallback(
    PTP_CALLBACK_INSTANCE pci
);
```

Thread Pool Thread Creation and Deletion

The Vista thread pool—like most thread pools you'll find—tries to keep its pool of running threads as close to the number of processors on the machine as possible. This allows it to fully utilize, without oversubscribing, the available hardware. But such a simple policy of having as many (or few) threads as there are processors is not good enough. Threads are apt to block occasionally, in which case the thread pool often needs to introduce more threads than there are processors, enabling additional work to be done while the waiting occurs. The Vista thread pool does precisely this. While the details about to be discussed are subject to change from release to release, an overview of them will at least give you an idea of the variables considered by the pool.

All Vista pools begin life with no threads, including the process-wide default thread pool. As work is queued, additional threads are introduced as quickly as needed to execute work items until the goal of having the same number of threads as processors is reached. Once this goal has been reached, subsequent thread creation is throttled. I/O completion ports are used to communicate work to these threads and to block them. Namely, if one of the thread pool threads has been blocked for longer than 10 milliseconds, causing the active threads to drop below the processor count, and the queue is nonempty, a new thread will be created automatically to execute the work. The decision about when to introduce new threads is made anytime new work is enqueued, in addition to various other points throughout the thread pool's implementation.

Throttling at 10 milliseconds instead of instantaneously introducing more threads as soon as a blocked thread is witnessed helps to avoid creating too many threads when work blocks for very short periods of time. This kind of short blocking happens frequently in many systems, due to things like page faulting and momentary waits for contended resources, like locks.

Threads are destroyed automatically after they have been idle for 10 seconds without having any work to perform, no matter whether this brings the thread count below the number of processors or not.

Obviously the thread count won't drop below the pool's minimum, if one has been specified with `SetThreadpoolThreadMinimum`. Similarly, the thread count won't exceed the maximum, if specified by a call to `SetThreadpoolThreadMaximum` (or the default of 500).

As we'll see in Chapter 15, Input and Output, each I/O completion port has a **concurrency level** representing the desired number of actively running threads processing completion packets from the port. When worker threads aren't executing callbacks, they are waiting on the I/O completion port. Windows will do its best to ensure the number of runnable threads processing work from the port stays as close to the concurrency level as possible, done in part by integration with the OS blocking primitives. Each pool's concurrency level is set to the number of processors on the machine. So even if the pool introduces more threads than processors (because of the conditions noted above), that doesn't mean all of them will continue running. For example, imagine there are P threads, where P is the number of

processors, and the thread pool creates another because one of those threads was blocked for 10 milliseconds; immediately after this, the thread unblocks; now we have $P + 1$ running threads; the next thread to go back to the completion port, assuming none of them subsequently block again, will not be given any work to do because the port knows that the desired concurrency level has already been reached.

In low resource conditions, the thread pool may not be able to create enough worker threads to perform all of the work in the queue. The pool will keep trying to introduce threads after such failures, with a delay of 10 seconds in between each attempt, until it succeeds.

Thread pool threads are created with the default stack reserve/commit information from the PE file. There is no way to override this. If you need threads with very large stacks, you will have to resort to manual thread management using `CreateThread`, and so forth, or by changing the PE file's default stack sizes, as discussed in Chapters 3 and 4.

The thread pool's heuristics are very effective for most cases. In some circumstances, however, it may be necessary for work on the pool to take an extraordinarily long time to complete. In these cases, you run the risk of starving other work that is waiting to be serviced in the pool, even though the callback may not necessarily block or do something to trigger the pool to create more threads. (As an aside, the thread pool is not well suited for this. You should try, to the best of your ability, to marshal any long running work such as this to a dedicated thread instead of tying up one of the thread pools.) Long running callbacks should notify the thread pool via the `CallbackMayRunLong` function. This tells the thread pool to allocate a new thread in to process other work. When the work item completes, the thread pool is told that it can safely destroy this extra thread. You can also notify the thread pool that an entire group of work associated with a particular environment is expected to run long with the `SetThreadpoolCallbackRunLong` API.

```
BOOL WINAPI CallbackMayRunLong(PTP_CALLBACK_INSTANCE pci);
VOID SetThreadpoolCallbackRunsLong(PTP_CALLBACK_ENVIRON pcbe);
```

The `CallbackMayRunLong` function returns TRUE if the thread pool was able to either free up another thread to process work or create an entirely new

thread, and FALSE otherwise. A return value of FALSE doesn't necessarily mean the thread pool won't subsequently introduce work based on its ordinary heuristics. This API should be viewed as a hint, and, thus, the return value isn't tremendously valuable. `SetThreadpoolCallbackRunsLong` provides no indication of whether it could free up a thread or not.

Callback Completion Tasks

There are a whole bunch of completion tasks that can be associated with a thread pool callback. All of them are similar in that they will execute after the callback is finished but before returning the thread back to the pool. These simplify various synchronization sensitive, but fairly common, activities upon callback completion:

```
VOID WINAPI LeaveCriticalSectionWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci,
    PCRITICAL_SECTION pcs
);
VOID WINAPI FreeLibraryWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci,
    HMODULE mod
);
VOID WINAPI ReleaseMutexWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci,
    HANDLE mut
);
VOID WINAPI ReleaseSemaphoreWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci,
    HANDLE sem,
    DWORD crel
);
VOID WINAPI SetEventWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci,
    HANDLE evt
);
```

Each function takes a pointer to a `TP_CALLBACK_INSTANCE`, which is supplied by the thread pool as the first argument to the callback itself. So if you're going to use any of them, you'll be making the call from inside the callback code. `LeaveCriticalSectionWhenCallbackReturns` takes a pointer to a `CRITICAL_SECTION` data structure and ensures the section is released when the callback finishes. `ReleaseMutexWhenCallbackReturns`,

`ReleaseSemaphoreWhenCallbackReturns`, and `SetEventWhenCallbackReturns` each take a HANDLE to a mutex, semaphore, or event kernel object, respectively, and ensure the object is signaled when the callback completes. `ReleaseSemaphoreWhenCallbackReturns` also takes a count, `crel`, which indicates how many times to release the semaphore. `FreeLibraryWhenCallbackReturns` simply calls the `FreeLibrary` function to unload a DLL from memory. These callback completion routines are only issued if the callback completes without throwing an unhandled exception; this is generally fine since the process will exit anyway, but if you are relying on state during process shutdown, this could be an issue that you encounter. For these cases, it's better to write your own explicit `__try/__finally` blocks in the callback.

Each callback can only remember one unique value for each of the cleanup APIs. If you try to make multiple calls to any of them, the thread pool will raise an `ERROR_INVALID_PARAMETER` exception. For example, if you want to release two critical sections when your callback finishes, you cannot do so by calling `LeaveCriticalSectionWhenCallbackReturns` once for each critical section. You'll need to do it the old fashioned way, at least for all but one of them.

Though the order of execution for these callbacks is not documented, empirical data suggests that it is done in the following order.

1. The critical section is released, if applicable.
2. The mutex is released, if applicable.
3. The semaphore is signaled, if applicable.
4. The event is set, if applicable.
5. The DLL is freed, if applicable.

While being undocumented means that the order of execution is subject to change, for application compatibility reasons it's doubtful that it will. Nevertheless, you shouldn't take a dependency on this fact. The reason I bring this up is that it could help you debug a tricky synchronization timing issue. Note also that if any of these steps fail, the thread pool thread will stay alive, but, depending on which step fails, subsequent callbacks may not execute: if signaling the semaphore fails, for instance, then the event will not be set.

Remember: You Don't Own the Threads

When your code runs inside a callback from a thread pool thread, you must not leave any thread local state polluting the thread when it is returned to the pool. Such state could adversely affect future work that subsequently gets scheduled on the same thread. Once a thread has been polluted in this way, it's only a matter of time before a conflict occurs: it's only a matter of severity and it's bound to be very nondeterministic, meaning it will be very difficult to track down. Reproducing the failure will involve tracing the history of work that once ran on a specific thread, possibly going back very far in time.

A very simple example of pollution is changing a thread's priority. If you call `SetThreadPriority` on a thread pool thread to, say, bump the priority to higher than normal, then future work will also run at that higher priority. Another example is calling `CoInitialize` on a thread pool thread to join an STA. All subsequent work will run under the STA, and, depending on whether you are working with any COM components in the thread pool callbacks, strange anomalies may arise. Moreover, depending on whether any other components already joined an apartment, the call may or may not succeed. Yet another example is the simple act of placing data into TLS and leaving it there. If future callbacks try to access this slot, they will find the data that was left behind and likely get confused.

Generally speaking, the Vista thread pool does not check for and revert any sort of thread pollution. It does, however, check for one specific case because of the thread of security vulnerabilities: if a thread is returned to the pool with security impersonation left on it, the thread pool will revert the impersonation before executing any additional work on that thread. As with the stack overflow policy mentioned earlier, this is a dubious policy. If impersonation was left on, it's likely that state of the kinds mentioned might have been left behind too.

Persistent Threads. The legacy thread pool has an option to queue work to a "persistent thread." This guarantees that the thread on which a particular work item runs will not exit as long as the thread pool continues running work. This is there to accommodate functions such as `RegNotifyChangeKeyValue`, which requires that the thread on which the function is called remains

alive. While the new Vista thread pool doesn't support persistent threads, you achieve the same effect by creating a separate pool object and using `SetThreadpoolThreadMinimum` and `SetThreadpoolThreadMaximum` to set the minimum and maximum thread counts to equal values. This ensures that no threads in that particular pool will ever exit.

Doing this interferes with the pool's ability to manage resources, so it should only be used to work around application compatibility problems. Even then you should probably consider using the legacy APIs. The legacy APIs are supported on Vista: internally, the thread pool manages a separate pool object that only has a single thread bound to it.

Debugging

There are a set of useful debugger commands available through the `!tp` extension in Windbg. Here is a dump of its usage from the tool itself.

```
Usage: !tp pool    <address> <flags> -- dump a thread pool
        obj     <address> <flags> -- dump a work, io, timer, or wait
        tqueue <address> <flags> -- dump the active timer queue
        waiter [address]      -- dump a thread pool waiter
        worker [address]      -- dump a thread pool worker

Flag definitions:
0x1   -- dump tersely (single-line output)
0x2   -- dump members
0x4   -- dump pool work queue
```

For pool, waiter, and worker, an address of zero will dump all objects. For waiter and worker, omitting the address will dump the current thread.

We won't drill too deeply into the output from these commands because they expose many implementation details about which most people won't care and that would be overkill to review. One of the more useful capabilities, however, is to dump the work queue with `!tp pool ... 0x6`, allowing you to see a count of pending callbacks, cleanup group information, and other objects that you can chase with the `!tp obj` command.

Legacy Win32 Thread Pool

We'll spend considerably less time discussing the legacy Win32 thread pool. We bring it up for two reasons: people are apt to be writing or maintaining

code that uses the old thread pool for years to come (not everybody can take a dependency on a brand new OS right away, nor can they rewrite all of that existing code), and for historical insight into the platform's origin.

The old thread pool has been reimplemented in Vista in terms of the new one, and so as we review the old APIs, we'll relate them back to the new ones.

Work Items

To queue a work item with the legacy thread pool, you use `QueueUserWorkItem`.

```
BOOL WINAPI QueueUserWorkItem(
    LPTHREAD_START_ROUTINE Function,
    PVOID Context,
    ULONG Flags
);
```

The `Function` is a pointer to the callback routine, which happens to use the same function pointer type as `CreateThread` (though the return value from the callback is ignored); `Context` is an opaque `PVOID` passed to the `Function` when invoked; and the `Flags` allow you to control a few aspects of where and how the callback runs. These flags include three mutually exclusive options.

- `WT_EXECUTEDEFAULT` (`0x0`): This is the default (i.e., if you pass `0`) that causes the work to get queued to an ordinary worker thread. All waiting on this thread is done with an I/O completion port, which means that waits are nonalertable and, thus, no APCs are able to run. Additionally, these threads do not check for outstanding I/O before exiting. If you exit a thread before the asynchronous I/O, it initiated has completed, the I/O request will be canceled; if you begin asynchronous I/O on such a thread, you will be disappointed.
- `WT_EXECUTEINIOTHREAD` (`0x1`): This flag ensures that the thread on which the callback runs will not exit before asynchronous I/O requests or APCs that were begun on it have completed. This ensures that it's safe to initiate asynchronous I/O operations from the thread pool. The queuing of this work is done with an APC. That

means that if any work running on an I/O thread performs an alertable wait, it may result in dispatching a work item that has been queued to an I/O thread. This can cause reentrancy problems, so you must take care to ensure that thread-wide state is consistent whenever an alertable wait is issued on such a thread. The Vista thread pool now treats all callback threads as I/O threads, in the sense that it won't exit before all initiated asynchronous I/O has finished.

- **WT_EXECUTEINPERSISTENTTHREAD (0x80)**: As mentioned earlier, a small number of Win32 APIs requires that a thread stay around "forever" after the API has been called on that particular thread. `RegNotifyChangeKeyValue` is one such routine. Specifying this flag ensures that the callback runs on a thread that won't go away and therefore enables you to use such APIs. This is implemented pre-Vista by running the work on the default timer queue's thread. As we will see, running code on this thread is dangerous because it can delay timer expirations. So if you need to use this option, first reconsider it and then proceed with great care. On Vista, at least, this causes work to run on a hidden dedicated single-threaded pool.

There are two other flags that are orthogonal.

- **WT_EXECUTELONGFUNCTION (0x10)**: This, much like the Windows Vista thread pool's `CallbackMayRunLong` API, instructs the pool that the work about to run may take a long time. The thread pool responds by dedicating more threads than it would have otherwise thrown at the pool. This translates to one additional thread for each work item queued with this flag.
- **WT_TRANSFER_IMPERSONATION (0x100)**: This flag, which is new to Windows XP SP2 (client) and Windows Server 2003 (server), causes the `QueueUserWorkItem` routine to capture the calling thread's impersonation token and to propagate it to the thread pool thread for the duration of the callback. Normally, when this flag isn't set, the process identity token is used instead and the impersonation token from the queuing thread is ignored.

After calling this function, the work has been queued to a work queue and will execute as soon as threads are available. QueueUserWorkItem can fail because it must allocate memory, in which case it returns FALSE, and GetLastError will return details about the failure.

Timers

The legacy thread pool's timer facilities allow you to group many timers together into something called a **timer queue**. A timer queue is a logical grouping of related timers that can be managed and deleted at once and provides some level of isolation between timers so that one group can be serviced and can expire without affecting another. The thread pool associates a single timer thread with each timer queue that has been created. There is also a single default timer queue that your program can use if you don't want to group them together. Individual timers are associated with a particular timer queue, which is what specifies the callback and expiration information including whether the timer is a one-shot or recurring timer.

Before creating individual timers, we can create a timer queue.

```
HANDLE CreateTimerQueue();
```

This function returns a HANDLE to the newly created queue, or NULL if creation of the queue failed. The next step to creating a timer is to associate one or more individual timers with a queue using the CreateTimerQueueTimer function.

```
BOOL WINAPI CreateTimerQueueTimer(
    PHANDLE phNewTimer,
    HANDLE TimerQueue,
    WAITORTIMERCALLBACK Callback,
    PVOID Parameter,
    DWORD DueTime,
    DWORD Period,
    ULONG Flags
);
```

The TimerQueue argument is just the HANDLE that was previously returned from CreateTimerQueue. Passing NULL for this argument uses the process-wide default timer queue, if you don't have a need to create and

specify your own. `Callback` is the function to call whenever the timer expires and `Parameter` is an opaque `PVOID` that gets passed to the callback. `WAITORTIMERCALLBACK` is a pointer to a function of the following signature.

```
VOID CALLBACK WaitOrTimerCallback(
    PVOID lpParameter,
    BOOLEAN TimerOrWaitFired
);
```

The `lpParameter` argument will be whatever was passed as `Parameter` to the `CreateTimerQueueTimer` routine, and `TimerOrWaitFired` will always be `TRUE` to indicate that the callback was caused by a timer expiring.

One thing you'll notice is that the specification of expiration times for timers is easier with the legacy APIs than with Vista's thread pool. The `DueTime` argument represents the relative time of the timer's first expiration, in milliseconds, from the current time. `Period` is for recurring timers. Specifying a value of `0` indicates a one-shot timer; any non-`0` value creates a recurring timer that will continue to fire every so many milliseconds until it has been explicitly stopped or deleted.

The API returns `FALSE` to indicate failure, and the `phNewTimer` output argument is a pointer to a `HANDLE` that receives the newly created timer's `HANDLE`. This is needed to work with the timer subsequently, including deleting it.

The `Flags` argument for `CreateTimerQueueTimer` accepts a superset of the values `QueueUserWorkItem` accepts. Everything said above for `WT_EXECUTEDEFAULT`, `WT_EXECUTEINIOTHREAD`, and so on, applies also for timer callbacks. One additional value is possible: `WT_EXECUTEINTIMERTHREAD (0x20)`, and, to be truthful, you should do your best to avoid it completely. Specifying this flag indicates that the timer's callbacks should be run on the actual thread that waits for timers to expire and, usually, handles queuing work to execute as normal callbacks in the thread pool callback threads. Running callbacks on this thread can delay other expiring timers. Moreover, because timers result in APCs being queued to the timer thread, any code that blocks using an alertable wait can cause other timer code to be dispatched, which (for other callbacks that use `WT_EXECUTEINTIMERTHREAD`) can cause difficult reentrancy problems. The often cited motivation for using this feature is to eliminate the

overhead required to transfer the work to a callback thread; it can offer better performance, but there are a multitude of worries that follow.

One thing you can do with the HANDLE returned by `CreateTimerQueue-Timer` is to alter an existing timer's recurrence after it's been created. This won't work for one-shot timers that have already expired (the call is ignored—note the difference compared to Vista), though you can change their initial firing date, provided it hasn't already passed.

```
BOOL WINAPI ChangeTimerQueueTimer(
    HANDLE TimerQueue,
    HANDLE Timer,
    ULONG DueTime,
    ULONG Period
);
```

This changes the target timer's `Due Time` and `Period` as though these values had been specified initially when the timer was created. The `TimerQueue` argument must be the same HANDLE that was specified when you created `Timer`. You can use this API to turn a recurring timer into a one-shot timer (that is, the next time it expires will be its last) by specifying a 0 for the `Period` argument.

When you're done with a timer, it must be deleted with the `Delete-TimerQueueTimer` function. This de-allocates the resources associated with it and is necessary even for one-shot timers. It also has the effect of stopping a recurring timer from firing subsequently:

```
BOOL WINAPI DeleteTimerQueueTimer(
    HANDLE TimerQueue,
    HANDLE Timer,
    HANDLE CompletionEvent
);
```

The first two arguments are simple; they specify the queue and timer that is to be deleted. The `CompletionEvent` argument is more complicated. The simplest thing to do is to pass `NULL` as `CompletionEvent`. The `Delete-TimerQueueTimer` routine will stop the timer from firing again in the future, but you will not know when all callbacks associated with the timer have finished. If you need to unload a DLL that the timer callback uses or to do any state manipulation that would interfere with the timer's ability to complete, you would need to build in additional synchronization to ensure you

don't proceed until all callbacks have finished. This would be quite difficult to do, particularly since you wouldn't know which callbacks were still sitting in the thread pool's callback queue.

That's the purpose of `CompletionEvent`. If you pass `INVALID_HANDLE_VALUE`, the call to `DeleteTimerQueueTimer` will not return until all of the callbacks have finished running for the target timer. This is quite handy and helps to deal with the aforementioned problems. Similarly, you can pass a real kernel object `HANDLE` (usually to an event object), in which case it will be signaled by the thread pool once all callbacks have finished for the target timer. You shouldn't be waiting for the timer to finish running from within a timer callback because the callback would be waiting for itself to finish.

If you create your own timer queues, you must delete those too. To do this, use either the `DeleteTimerQueue` or `DeleteTimerQueueEx` function.

```
BOOL WINAPI DeleteTimerQueue(HANDLE TimerQueue);
BOOL WINAPI DeleteTimerQueueEx(
    HANDLE TimerQueue,
    HANDLE CompletionEvent
);
```

The `CompletionEvent` argument for `DeleteTimerQueueEx` is interpreted the same way as `DeleteTimerQueueTimer`: that is, `INVALID_HANDLE_VALUE` requests that the thread be blocked until all callbacks in the queue have finished, a real object `HANDLE` asks for it to be signaled when all have finished, and `NULL` means return right away without waiting. `DeleteTimerQueue` is the same as calling `DeleteTimerQueueEx` with a `NULL` value for `CompletionEvent`.

I/O Completion Ports

As with the Vista pool, you can use the legacy APIs to specify that a callback runs on the thread pool whenever an asynchronous I/O operation completes on a particular `HANDLE` or `SOCKET`. This is done with the `BindIoCompletionCallback` routine.

```
BOOL WINAPI BindIoCompletionCallback(
    HANDLE FileHandle,
    LPOVERLAPPED_COMPLETION_ROUTINE Function,
    ULONG Flags
);
```

This works in the same basic way the Vista API does. `FileHandle` must represent a file, named pipe, or socket handle opened for overlapped I/O, `Function` is a callback routine that responds to the completion event, and `Flags` is just a reserved argument and must be the value `0`. The callback is a pointer to a function with the following signature.

```
VOID CALLBACK FileIOCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped
);
```

Note that it is possible to issue additional asynchronous I/O operations from the callback. In this case, however, you must be careful; you cannot simply issue the asynchronous I/O request. Recall the discussion earlier about `WT_EXECUTEDEFAULT` and `WT_EXECUTEINIOTHREAD` and that the default threads may exit before the I/O completes. To work around this, you can marshal the call to create the asynchronous I/O work to an I/O thread using the `QueueUserWorkItem` function, passing the `WT_EXECUTEINIOTHREAD` flag. This extra step is a little cumbersome—it would be nice if `Flags` accepted `WT_EXECUTEINIOTHREAD` rather than being reserved—but is required to ensure I/O completions do not get silently dropped.

Registered Waits

The Win32 function `RegisterWaitForSingleObject` registers a callback to be invoked by the thread pool once the specified `HANDLE` is signaled, just like the Vista APIs `CreateThreadpoolWait` and related APIs already described. This API was added in Windows 2000, and requires `_WIN32_WINNT` to be defined at `0x0500` or higher.

```
BOOL WINAPI RegisterWaitForSingleObject(
    PHANDLE phNewWaitObject,
    HANDLE hObject,
    WAITORTIMERCALLBACK Callback,
    PVOID Context,
    ULONG dwMilliseconds,
    ULONG dwFlags
);
```

The `hObject` argument specifies the kernel object on which the wait registration will wait. Before returning, the function will store a wait handle

into `phNewWaitObject`, which can be subsequently used to deregister the wait. This is not an ordinary object `HANDLE`; you cannot close it, wait on it, or do anything that you'd normally do with a `HANDLE`. `Callback` is a pointer to the function to invoke once the object becomes signaled, and `Context` is an opaque value that gets passed to this callback. We've already seen `WAITORTIMERCALLBACK` when we reviewed timers—it's typedefed as a pointer to a function with the following signature.

```
VOID CALLBACK WaitOrTimerCallback(
    PVOID lpParameter,
    BOOLEAN TimerOrWaitFired
);
```

As you might guess, the `Context` passed to `RegisterWaitForSingleObject` is passed as `lpParameter` to the callback.

You can specify a timeout with the `dwMilliseconds` argument. As with most other wait APIs, a value of `INFINITE` (i.e., `-1`) means no timeout, a value of `0` indicates the state of the object should be tested without blocking, and anything else places an upper limit on the number of milliseconds before the callback will time out. If a callback times out, the thread pool will pass `FALSE` for the callback's `TimerOrWaitFired` argument, otherwise it is `TRUE`.

Because `RegisterWaitForSingleObject` must allocate memory, it can fail. If it does, it will return `FALSE`, and further details can be extracted by calling `GetLastError`.

The `dwFlags` parameter for `RegisterWaitForSingleObject` controls a vast number of things. In fact, it is a superset of those options supported by `QueueUserWorkItem`'s `Flags` argument, and all of the same caveats apply.

There are two flags that are specific to wait registrations. The first is `WT_EXECUTEONLYONCE` (`0x8`). Perhaps the biggest difference in behavior between the new Vista pool and the legacy pool is that the legacy thread pool continually reregisters waits after callbacks finish. We saw already that the Vista pool does not do this (though we saw how to simulate it). This continuous reregistration happens until the registration is manually unregistered through a call to either `UnregisterWait` or `UnregisterWaitEx` (which we'll look at soon), even if the callback is invoked due to a timeout. To change this behavior, you may specify the `WT_EXECUTEONLYONCE` flag in `dwFlags` during registration. This guarantees that only one callback will

ever be queued per registration. This is useful particularly for objects that remain signaled, such as manual-reset events. If you register a wait that is set to execute multiple times (the default) on such an object, callbacks will be queued indefinitely up as fast as the thread pool can queue them once the object becomes signaled. The resulting situation is highly problematic and can lead to infinite queuing.

The second wait specific flag, `WT_EXECUTEINWAITTHREAD` (`0x4`), specifies that the callback should run on the thread used for waiting instead of being transferred to a worker thread via a callback. This is equivalent to `WT_EXECUTEINTIMERTHREAD` and has all of the same disadvantages that we already reviewed. The callback can interfere with the pool's ability to dispatch wait callbacks in a timely fashion.

The `WT_EXECUTEINWAITTHREAD` option can be used as a workaround for the mutex issue noted earlier. Because the thread that runs your callback is the same one that waited on the mutex, your callback is able to release the mutex. The mutex situation is worse on the legacy APIs if this flag isn't set. If `WT_EXECUTEONLYONCE` is not set, the wait thread will go back and try to wait on the mutex as soon as the callback is dispatched. Since mutex acquisitions are recursive, this wait will be satisfied immediately, leading to a similar problem to the manual-reset event situation mentioned previously.

Each registration must eventually be unregistered with either `UnregisterWait` or `UnregisterWaitEx`. Unregistering a wait ensures no subsequent callbacks are generated for the registration, and then it de-allocates all of the resources associated with it.

```
BOOL WINAPI UnregisterWait(HANDLE WaitHandle);  
BOOL WINAPI UnregisterWaitEx(HANDLE WaitHandle, HANDLE CompletionEvent);
```

While unregistering a wait ensures no future callbacks will be created, there could be one or more that have already been queued to the thread pool's work queue and/or actively running on thread pool threads. If there is at least one callback associated with the specified `WaitHandle` that is still active, the function returns `FALSE` and `GetLastError` returns `ERROR_IO_PENDING`. The wait in this case has been unregistered, but you must be careful; you mustn't release any resources that the callbacks may need to use (such as unloading dynamically loaded DLLs).

`UnregisterWaitEx` allows you to be notified when all callbacks have finished, which provides a way to cope with this issue. The simplest way of doing this is to pass `INVALID_HANDLE_VALUE` as `CompletionEvent`, in which case the call to `UnregisterWaitEx` blocks until all callbacks have finished. Alternatively, you can supply a `HANDLE` to a kernel object (such as an event) for the `CompletionEvent` argument, and the thread pool will signal the object once all associated callbacks have completed. This allows you to control the way in which the thread waits, including possibly pumping messages.

Thread Pool Thread Management

Because the old thread pool APIs are built right on top of the new Vista ones, everything discussed in the previous section now applies to the legacy APIs too (when run on Vista). The new Vista thread management policies are vastly improved over the old ones—the old APIs throttled the creation of new threads dramatically—so we won’t go into many details about how the previous scheme worked.

The old thread pool capped the maximum number of threads at 512 by default, whereas the new one caps them at 500. With the legacy pool, you used to be able to change this maximum with a macro from `Winnt.dll`, `WT_SET_MAX_THREADPOOL_THREADS`, that takes two arguments: `Flags`, which is just a variable containing flags that will be passed to `QueueUserWorkItem` (see earlier), and `Limit`, which represents the new maximum count. This macro encodes `Limit` into the contents of the `Flags` in a special way so that `QueueUserWorkItem` sees it and can respond. The way that `Limit` is encoded means that you cannot set the limit higher than about 65,535, which happens to be quite a few more threads than you’d ever need anyway.

For example, this call sets the pool’s limit to 1,000 threads.

```
ULONG someFlags = ...;
WT_SET_MAX_THREADPOOL_THREADS(someFlags, 1000);
QueueUserWorkItem(&MyWorkCallback, NULL, someFlags);
```

It turns out that this tactic won’t work on Vista. This setting will be ignored. There is no way to change the default pool’s maximum—you’ll need to create a separate pool and use the `SetThreadpoolThreadMaximum` routine.

This could create some surprising application compatibility problems when moving programs that use the old thread pool to Vista, so beware.

CLR Thread Pool

The CLR provides an entirely different set of APIs, though they have very similar capabilities to the native Windows thread pools. The basics are the same: you can queue up a chunk of work that will be run by the thread pool, use the pool to run some work when asynchronous I/O completes, execute work on a recurring or timed basis using timers, and/or schedule some work to run when a kernel object becomes signaled using registered waits. The interface is much more akin to the legacy native thread pool APIs than the new Vista ones.

The CLR thread pool internally manages two process-wide pools of threads and consequently two ways of tracking work. One pool of threads uses a custom work queue and is meant to execute work item callbacks, timer expiration callbacks, and wait registration callbacks. The other pool of threads uses an I/O completion port and executes only I/O completion callbacks. Being process-wide, these are shared among all CLR AppDomains inside the process. The thread pool manages servicing all AppDomains in the process as fairly as it can manage.

When a managed process starts, there are no threads dedicated to the worker pool (by default). Upon the first work item being queued to the pool, the CLR will spin up a new thread to execute the work. When that thread is done executing the work item, it returns to the pool, waits for a new work item to be queued, executes it, and so on. As new threads are needed, they are created, and as existing threads are no longer needed, they are destroyed. The same basic architecture is also true of the I/O pool. The process is more complicated than this, but at a high level, that's what happens. We'll look deeper into the specific heuristics used after we see how to use the thread pool.

Work Items

There is a `ThreadPool` static class in the `System.Threading` namespace. The `QueueUserWorkItem` and `UnsafeQueueUserWorkItem` static methods are the

popular ones, and both schedule work to execute concurrently on a thread pool worker thread.

```
public static class ThreadPool {
    public static bool QueueUserWorkItem(WaitCallback callBack);
    public static bool QueueUserWorkItem(
        WaitCallback callBack,
        object state
    );

    [SecurityPermission(SecurityAction.LinkDemand, Flags=
        SecurityPermissionFlag.ControlPolicy |
        SecurityPermissionFlag.ControlEvidence)]
    public static bool UnsafeQueueUserWorkItem(
        WaitCallback callBack,
        object state
    );
    ...
}
```

Each method takes a delegate of type `WaitCallback` and, optionally, an extra `state` argument, typed as `object`, which is passed through to the callback and accessible via its sole argument. Though these methods are typed as returning a `bool`, this was a mistake in the original API design: they always communicate failures by throwing an exception. `WaitCallback` is just a simple delegate type:

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

Most programs should use `QueueUserWorkItem` instead of `UnsafeQueueUserWorkItem`. The only difference between them is whether an `ExecutionContext`, which includes various security information (such as the `SecurityContext` and `CompressedStack`), is captured at the time of the call (on the queuing thread) and then used when invoking the `callBack` on the thread pool. As the names imply, `QueueUserWorkItem` captures and restores the context, while `UnsafeQueueUserWorkItem` does not.

Because `QueueUserWorkItem` is available to partially trusted code, it will always capture and flow the context. This also includes impersonation information established for the thread in managed code. The context is then restored on the thread pool thread just prior to invoking the delegate and cleared afterwards. This ensures that a partially trusted program or piece of code cannot elevate its privileges simply by queuing work to the thread

pool. `UnsafeQueueUserWorkItem` gets around this, but as shown previously, using it requires satisfying a link demand for `ControlPolicy` and `ControlEvidence` permissions. If your assembly could end up running work that originates from a partially trusted caller on the thread pool, you most want to use the `QueueUserWorkItem` method to avoid the possibility of elevation of privilege security vulnerabilities.

The reason why there's even a question about which to use—that is, why not always err on the side of security and flow the context?—is because `QueueUserWorkItem` costs more due to the extra context capture and restoration steps. The overhead imposed means `QueueUserWorkItem` is somewhere in the neighborhood of 15 to 30 percent more than a call to `UnsafeQueueUserWorkItem` in terms of micro-benchmarked execution time. (Prior to 2.0, the overhead was actually over 100 percent.) For fine-grained work items run by code that never executes in anything but a full trust environment, this overhead may be noticeable enough that you want to use the unsafe method instead. But, conversely, this is noise for many cases because the call's absolute cost is fairly small.

Note that the `CurrentCulture`, `CurrentUICulture`, or `CurrentPrincipal` state does not flow from the queuing thread to the thread pool. If you wish to flow this state, you have to do it manually by hand. Unlike the Windows impersonation identity token, these properties were always intended for application specific purposes.

The queued delegate ends up executing on any arbitrary thread pool thread, solely determined by which thread gets to it first. This means you should not take dependencies on any thread specific state persisting between executions of different callbacks because the thread chosen to execute your callbacks is apt to change. Sometimes, by chance, the same thread might be chosen, which has the effect of masking a problem.

If a thread pool work item throws an exception that goes unhandled, the CLR will use the ordinary unhandled exception policy process to decide what to do. In cases that don't involve an external host such as SQL Server or ASP.NET, the process will crash (provided the exception is not of type `ThreadAbortException` or `AppDomainUnloadedException`, which are swallowed). Prior to the CLR 2.0, the thread pool would silently

swallow and ignore all unhandled exceptions. The change in behavior was instituted to ensure that important failures don't go unnoticed, helping managed code developers build and test for superior robustness and reliability. There is a configuration flag to control this; it was explained in Chapter 3, Threads.

Unlike the Vista thread pool, there isn't any easy out-of-the-box way to wait for the completion of a work item or set of work items that were queued to the thread pool. This is unfortunate because it's a rather common requirement. The simplest approach is to allocate an event that is set at the end of the work and then have the calling thread wait on it.

```
using (ManualResetEvent finishedEvent = new ManualResetEvent(false))
{
    ThreadPool.QueueUserWorkItem(delegate
    {
        // Do the work here.
        finishedEvent.Set();
    });

    // Continue working concurrently with the thread pool work...
    // And then wait for it to finish:
    finishedEvent.WaitOne();
}
```

While simple, this isn't the most efficient approach. It's often the case that the thread pool work will finish before the calling thread gets around to checking, in which case it'd be nice to not allocate the event at all. And if we want to wait for many callbacks to finish executing, things become more complicated. Your first approach might be to allocate an event for each work item, but this is extraordinarily inefficient. A better approach is to have the last completed callback signal the event. That might look something like this.

```
int remainingCallbacks = n;
using (ManualResetEvent finishedEvent = new ManualResetEvent(false))
{
    for (int i = 0; i < n; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate
        {
            // Do the work here.
        });
    }
}
```

```
        if (Interlocked.Decrement(ref remainingCallbacks) == 0)
    {
        // The last callback sets the event.
        finishedEvent.Set();
    }
});

}

// Continue working concurrently with the thread pool work...
// And then wait for it to finish:
finishedEvent.WaitOne();
}
```

A managed process can exit with work items still sitting in the thread pool's queue, and even with items actively running on one or more thread pool threads. This is because each thread pool thread is marked as being a background thread. This surprises some people. If you have important work that must execute before the process exits—such as saving some user changes to data—you should consider using a separate scheduling mechanism. This might involve explicitly managing threads or looking at an alternative scheduling mechanism for these circumstances. Changing the thread pool thread's `IsBackground` property once your work is scheduled might seem like one possible solution, but it won't prevent the process from exiting before the work is seen and run by a thread in the pool.

I/O Completion Ports

As already mentioned, the CLR thread pool maintains a single process-wide I/O completion port. All the existing asynchronous I/O APIs in the .NET Framework rely on the thread pool's I/O completion port support to "do the right thing." For example, when you use `FileStream`'s `BeginRead` or `BeginWrite` methods, they will automatically coordinate with the thread pool to ensure that, when the I/O completes, the provided callback runs on an I/O thread in the thread pool. It's quite rare that anybody ever needs to work with the I/O APIs on the `ThreadPool` class itself.

If you read the previous section on how the native thread pool interacts with asynchronous I/O, the following will be familiar. And, once again, I will be a little terse when it comes to details about I/O completion

ports because they are covered in greater detail in Chapter 15, Input and Output.

Once you have an object opened that is capable of asynchronous I/O (e.g., a file opened with `CreateFile` with the `FILE_FLAG_OVERLAPPED` flag), all that is required for asynchronous I/O completions to fire on the thread pool is to call the `BindHandle` method.

```
public static class ThreadPool {
    [SecurityPermission(SecurityAction.Demand, Flags=
        SecurityPermissionFlag.UnmanagedCode)]
    public static bool BindHandle(IntPtr osHandle);
    public static bool BindHandle(SafeHandle osHandle);

    ...
}
```

The `IntPtr` overload is deprecated because `SafeHandle` is the preferred way of managing OS handles in the .NET Framework as of 2.0. In any case, I lied a little bit. Binding the handle to the thread pool isn't sufficient. The thread pool's I/O threads are expecting a certain format in the `OVERLAPPED` data structures used during asynchronous I/O so that it can find the callback information. If you don't conform to this, bad things will happen. So, you'll need to use the .NET Framework's overlapped APIs.

We'll omit as much discussion of the I/O specific parts of the overlapped APIs as we can. They are covered much more comprehensively in Chapter 15, Input and Output. There's only a small set of APIs that we need to discuss now, and they all exist on the `System.Threading.Overlapped` class.

```
public class Overlapped {
    public unsafe NativeOverlapped * Pack(
        IOCompletionCallback iocb
    );
    public unsafe NativeOverlapped * Pack(
        IOCompletionCallback iocb,
        object userData
    );
    [SecurityPermission(SecurityAction.LinkDemand, Flags=
        SecurityPermissionFlag.ControlPolicy|
        SecurityPermissionFlag.ControlEvidence)]
    public unsafe NativeOverlapped * UnsafePack(
        IOCompletionCallback iocb
    );
}
```

```
[SecurityPermission(SecurityAction.LinkDemand, Flags=
                    SecurityPermissionFlag.ControlPolicy|
                    SecurityPermissionFlag.ControlEvidence)]
public unsafe NativeOverlapped * UnsafePack(
    IOCompletionCallback iocb,
    object userData
);
...
}
```

You can construct a new `Overlapped` object with its no-argument constructor. There are other constructors that accept arguments that map to the native `OVERLAPPED` structure (which we've already established will be ignored for now). When we call either the `Pack` or `UnsafePack` method, we specify an `IOCompletionCallback` that will run when I/O completes. This is a simple delegate type.

```
public unsafe delegate void IOCompletionCallback(
    uint errorCode,
    uint numBytes,
    NativeOverlapped * pOVERLAP
);
```

The difference between `Pack` and `UnsafePack` is that the former captures the context and restores it before running the I/O callback and the latter doesn't. This is analogous to the difference between `QueueUserWorkItem` and `UnsafeQueueUserWorkItem`.

The `userData` object supplied to `Pack` is either an array or array of arrays that will be used as the buffers during asynchronous I/O operation. The runtime will pin these to ensure that they don't move while the asynchronous I/O is occurring and will unpin them when the I/O finishes. The runtime also handles synchronizing with `AppDomain` unloads to guarantee that, even if the `AppDomain` in which the I/O was initiated is unloaded before the I/O completes, the buffers remain pinned for as long as needed to avoid GC heap corruption.

Provided that the `NativeOverlapped *` returned by the pack API is used when initiating asynchronous I/O and that this I/O is against a file handle that's been bound to the thread pool with `BindHandle`, the `iocb` callback supplied will run on an I/O thread in the thread pool when said I/O completes.

You can marshal the `NativeOverlapped *` back into an `Overlapped` object with the static `Unpack` method and can release its resources with the static `Free` method. Internally there is a cache of `NativeOverlapped` objects, so when you allocate and free them, the implementation is returning objects from and to a pool of reusable structures.

Finally, there is an `UnsafeQueueNativeOverlapped` API on `ThreadPool` that provides an alternative way to run code in the thread pool for non-asynchronous I/O callbacks. This schedules an arbitrary callback that has been packed into a `NativeOverlapped *` to run on one of the thread pool's I/O threads without requiring that actual asynchronous I/O be involved. In other words, you completely control queuing the work. The implementation of this API turns around and posts a completion packet to the I/O completion port.

```
public static class ThreadPool {
    [SecurityPermission(SecurityAction.LinkDemand, Flags=
        SecurityPermissionFlag.ControlPolicy |
        SecurityPermissionFlag.ControlEvidence)]
    public static unsafe bool UnsafeQueueNativeOverlapped(
        NativeOverlapped * overlapped
    );
    ...
}
```

This API can be slightly more efficient than `QueueUserWorkItem` in some circumstances. Often the overhead of creating and managing `NativeOverlapped *` objects not only makes programming more complex, but also degrades performance due to pinning. Only if you do not need to allocate many overlapped objects—as would be the case if all of your calls to queue work used the same callback delegate—will you possibly see substantial performance improvements by allocating a single `NativeOverlapped *` and using `UnsafeQueueNativeOverlapped` instead of `QueueUserWorkItem`. This is the approach that the Windows Communication Foundation uses to queue work.

Timers

There is a `Timer` class in the `System.Threading` namespace that makes use of the CLR thread pool just as the Win32 timer interfaces use the native

thread pool. Using this class is straightforward. To create and schedule a new timer, construct one. By the time the constructor returns, the newly allocated `Timer` will have been registered with the pool.

```
[HostProtection(SecurityAction.LinkDemand,
    Synchronization=true, ExternalThreading=true)]
public class Timer : MarshalByRefObject, IDisposable {
    public Timer(TimerCallback callback);
    public Timer(
        TimerCallback callback,
        object state,
        int dueTime,
        int period
    );
    public Timer(
        TimerCallback callback,
        object state,
        long dueTime,
        long period
    );
    public Timer(
        TimerCallback callback,
        object state,
        TimeSpan dueTime,
        TimeSpan period
    );
    public Timer(
        TimerCallback callback,
        object state,
        uint dueTime,
        uint period
    );
    ...
}
```

All the overloads take a `TimerCallback`. This is a delegate that will be called on the thread pool each time the timer expires.

```
public delegate void TimerCallback(Object state);
```

The constructors also accept a `state` argument that is passed straight through to the callback and two pieces of time information: `dueTime`, which is the first time that the timer will expire; and `period`, which is the expiration recurrence after that first expiration. Both are specified in terms of milliseconds (unless you use the `TimeSpan` overload, in which case you can

specify hours, minutes, seconds, and so forth). If the period is 0, then the resulting timer is a one-shot timer and will not fire more than once. After creating the `Timer` object, it will have already been scheduled and will begin firing immediately based on the `dueTime`.

Timers always capture the current execution context and restore it on the callback thread, much like `QueueUserWorkItem`. There is no unsafe version that bypasses this.

There are several kinds of timers available in the .NET Framework. Another one lives in the `System.Timers` namespace of `System.dll`, and it follows the .NET component model: this allows you to drag and drop an instance onto a designer pane easily and also specify an `ISynchronizeInvoke` object to ensure that the timer works properly inside of a GUI application. Each presentation technology in the .NET Framework also offers its own special timer. Windows Forms, for example, provides the `System.Windows.Forms.Timer` class, and the Windows Presentation Foundation has a `System.Windows.Threading.DispatcherTimer` class. These are subtle variants on the timer theme, but tailor their APIs to the presentation framework in question.

You can change the timing information after the timer has been created using one of the `Change` methods. In fact, if you create a timer using the one constructor overload that doesn't take a `dueTime` or `period`, you must call `Change` on it before it will fire. Again, there are four overloads, one each for `Int32`, `Int64`, `TimeSpan`, and `UInt32`-specified times.

```
public class Timer : MarshalByRefObject, IDisposable {
    public bool Change(Int32 dueTime, Int32 period);
    public bool Change(Int64 dueTime, Int64 period);
    public bool Change(TimeSpan dueTime, TimeSpan period);
    public bool Change(UInt32 dueTime, UInt32 period);

    ...
}
```

After this call, the timer will fire again at the specified `dueTime` and recur with the specified `period` after that. Note that although `Change` is typed as returning a `bool`, it will actually never return anything but true. If there is a problem changing the timer—such as the target object already having been deleted—an exception will be thrown.

You can use `Change` to temporarily or permanently stop a timer from firing. If you pass `-1` as the `dueTime`, the timer will be put into a state such that no callbacks occur. This does not physically delete the timer object, so if you don't follow that with a call to `Dispose`, you will have a resource leak on your hands.

```
public class Timer : MarshalByRefObject, IDisposable {  
    public void Dispose();  
    public void Dispose(WaitHandle notifyObject);  
  
    ...  
}
```

The simple `Dispose` overload deletes the timer resources, including stopping the timer from firing in the future. This synchronizes with the timer implementation to ensure that concurrency issues are addressed. It is possible that after `Dispose` returns, there are timer callbacks that are either actively executing or sitting in the thread pool's work queue waiting to execute. That's what the second `Dispose` overload is for: if you pass a non-null `notifyObject` to it, the pool will signal it when all callbacks for the timer have completed. This can be any `WaitHandle`, such as a `ManualResetEvent`, for instance.

To simplify things, you can instead request that `Dispose` return only when all callbacks have completed by passing a `WaitHandle` with a `Handle` value of the default, `WaitHandle.InvalidHandle`. This is usually what you want to do and it avoids having to allocate a true event object, which is more costly. Since the `WaitHandle` class is abstract, you need to use a little hack, which is to create your own subclass.

```
class InvalidWaitHandle : WaitHandle { }  
Timer t = new Timer(...);  
...  
t.Dispose(new InvalidWaitHandle());
```

With this scheme, `Dispose` will only return once all of the timer's callbacks have finished running. You want to avoid waiting for the timer callbacks to complete from within a timer callback itself because that would lead to a deadlock.

Registered Waits

The CLR thread pool's wait registration feature was modeled almost directly from the legacy Win32 thread pool's similar support. Just as with

the native pools, there is a single wait thread created for every 63 objects registered. This thread manages waiting on objects and queuing the callbacks to run on one of the thread pool's worker threads when an object is signaled.

To create a new registration, use the `RegisterWaitForSingleObject` or `UnsafeRegisterWaitForSingleObject` method on `ThreadPool`.

```
public static class ThreadPool {
    public static RegisteredWaitHandle RegisterWaitForSingleObject(
        WaitHandle waitObject,
        WaitOrTimerCallback callBack,
        object state,
        int millisecondsTimeOutInterval,
        bool executeOnlyOnce
    );
    [SecurityPermission(SecurityAction.LinkDemand, Flags=
        SecurityPermissionFlag.ControlPolicy|
        SecurityPermissionFlag.ControlEvidence)]
    public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(
        WaitHandle waitObject,
        WaitOrTimerCallback callBack,
        object state,
        int millisecondsTimeOutInterval,
        bool executeOnlyOnce
    );
    ...
}
```

Each method offers four overloads, and all of them require you to pass a timeout. The three others haven't been shown because they are basically the same. They allow you to pass a `uint`, `long`, or `TimeSpan` for the timeout argument instead of an `int`.

The difference between `RegisterWaitForSingleObject` and `UnsafeRegisterWaitForSingleObject` is much like the difference between `QueueUserWorkItem` and `UnsafeQueueUserWorkItem`: the unsafe version does not capture and propagate the execution context and associated security state.

The `waitObject` argument is the kernel object whose signaling will cause the callback to be scheduled, `callBack` is the code to queue to the thread pool in response to either the object being signaled or the timeout expiring, and `state` is an opaque object that is just passed along to the callback. `WaitOrTimerCallback` is a delegate type defined as.

```
public delegate void WaitOrTimerCallback(object state, bool timedOut);
```

The milliseconds based timeout indicates when the wait should time out. If you don't wish to specify a timeout, `Timeout.Infinite` (-1) can be supplied. If a timeout occurs, the `timedOut` argument passed to the callback will be `true`; otherwise, it is `false`. If the `executeOnlyOnce` argument during registration is `true`, the callback will fire once before the registration is automatically disabled.

As was mentioned earlier, if you are registering a wait for an object that stays in the signaled state (e.g., a manual-reset event), then you must specify `executeOnlyOnce` if you'd like to avoid the thread pool continuously queuing a never ending number of callbacks as quickly as it can. And just as was mentioned for both the Vista and legacy thread pool APIs, registering a wait for a `Mutex` is a bad idea. As with Vista, there's no way in the .NET Framework to get the wait registration callback to run on the same thread that owns the mutex, meaning it can never be released after a registered wait is satisfied.

You'll notice these methods return an instance of `RegisteredWaitHandle`; this object can be used to stop a wait and/or clean up the registration's associated resources. If you fail to call `Unregister` on it at some point, a callback will be run anytime the object gets signaled for the rest of the process's lifetime.

```
public class RegisteredWaitHandle : MarshalByRefObject
{
    public bool Unregister(WaitHandle waitObject);
}
```

If you forget to call this for a registration for which `executeOnlyOnce` is `true`, a finalizer protecting the underlying resources will eventually take care of cleaning up the resources for you. If `executeOnlyOnce` is `false`, the resources will continue to be used, and wait callbacks will continue to be generated whenever the target object becomes signaled, until the process exits.

No additional callbacks will be queued after this call returns, but it is possible that some callbacks will be actively executing or in the queue waiting to execute. It is sometimes necessary to synchronize with the completion of the existing callbacks so that resources they use can be cleaned up without

worrying about races. That's the purpose of the `waitObject` argument. If a non-null `waitObject` is supplied, the CLR thread pool will signal it once the wait callbacks have completed. This is quite a bit like the timer's `Dispose` method described earlier, and the same `InvalidWaitHandle` trick shown earlier works here too.

```
class InvalidWaitHandle : WaitHandle { }
RegisteredWaitHandle rwh = ThreadPool.RegisterWaitForSingleObject(...);
...
rwl.Unregister(new InvalidWaitHandle());
```

Unregistering and waiting for callbacks to complete from within a wait callback itself will cause a deadlock.

Remember (Again): You Don't Own the Threads

It was already noted above in the context of the Windows thread pool that polluting a thread pool thread with some thread local state and then returning it to the pool is a bad practice. This is as true with managed code as it is with native code. The CLR's thread pool does, however, have a few safeguards in place that the native pools don't have. You should not rely on these, but they are worth mentioning.

Like Windows, the CLR will first and foremost reset any security impersonation information that may have been left behind. It also resets any culture that has been left behind, thread priority, the thread name (i.e., changes made with the `Thread.Name` property) and ensures that the thread is still marked as a background thread (i.e., `Thread.IsBackground` is `true`) so that it won't hold up process exit. The fact that these are reset automatically does not suggest that you should intentionally rely on them in any way. Many things are left as-is when a thread returns to the pool, however: TLS modifications, for example, are retained on the threads, because the performance cost of clearing TLS slots when each work item completes would be too high.

Thread Pool Thread Management

Let's quickly take a look at how the CLR thread pool decides when to create and destroy threads in the thread pool, and how you might impact this process.

Details of Thread Injection and Retirement Algorithm

As with the Windows thread pool, the CLR's pool abstracts the management of threads through the use of some sophisticated heuristics. The specific heuristics employed are different, however. These heuristics determine the optimal number of threads by looking at the machine architecture, rate of incoming work, and the current CPU utilization across the entire machine. Often referred to as the **thread injection and retirement algorithm**, this logic decides when to create new threads to process work and when to destroy threads due to lengthy periods of idle queue activity or because the machine is fully utilized. This is great because without it you'd need to figure it out yourself (and test it on various machine configurations, of course).

Even better is that most people can remain unaware of the specific algorithms behind injection and retirement. Depending on internal implementation details such as this is a bad idea anyway. But understanding them can help you to understand the performance and scalability characteristics of your program, and it is interesting for those who are thinking about alternative ways to schedule work.

Recall that the CLR thread pool actually manages two sets of threads: one of them handles general work items (`QueueUserWorkItem`, timer expiration callbacks, and wait registration callbacks); and the other handles any I/O completions (due either to `BindHandle` or `UnsafeNativeQueueNativeOverlapped`). Despite this, the thread management for both is nearly identical. The main difference is in how work is queued to the threads: in the worker thread case, there is a custom pool and associated work queue, while in the I/O thread case, everything happens through I/O completion ports. Additionally, I/O completion ports throttle the number of running threads.

When work is queued to the pool, the thread pool will create threads on the calling thread until the optimal number of threads has been reached. That optimal number is the processor count of the current machine. Once this target has been reached, the CLR will throttle the creation of threads. The CLR's heuristics are more complicated than the native pool heuristics (and one could argue not as effective), so we will avoid going into detail on the specific algorithms. To summarize:

- As soon as the target count has been reached, new thread creation is throttled at a maximum rate of one thread per 500 milliseconds.

Under no circumstances will the thread pool exceed this creation rate once the number of threads outnumbers the number of processors or minimum thread count, whichever is larger.

- A daemon thread runs in the background, periodically looking for starvation and possibly injecting new threads to service work. This decision is made based on complex logic that considers the depth of the work queue and the CPU utilization of the machine. Generally if the utilization is too low, it generates more threads; if the utilization is very high, it removes threads.
- If there are two or more idle threads with no work in the thread pool, the thread pool will instruct the excess threads to quit (subject to the minimum). This helps to ensure there aren't too many threads with no work to do. The remainder will eventually be taken care of by the daemon thread.
- It is possible to set the minimum and maximum number of threads in the pool, as we will see soon, which ensures the pool never shrinks below or grows above the specific values, respectively.

This thread injection and retirement logic is similar for I/O threads. It is more effective, however, because I/O completion ports automatically throttle the number of runnable threads based on when threads block in the kernel.

As a developer, you have little to no control over any of this. What you can control is the minimum and maximum number of threads in the pool. Usually the defaults are fine, but let's take a look at this feature anyway.

Minimum and Maximum Threads

Because there are separate pools of threads for worker and I/O threads, there are four values: minimum and maximum worker threads, and minimum and maximum I/O threads. The default minimum values for both are 0 threads. That means the process begins life with no threads dedicated to the pool and that during periods of idle time the pool can shrink back down to nothing. The default maximum values are set to a certain constant number multiplied by the number of processors at runtime: for worker threads the value is 25 per processor for the CLR 2.0 and 250 per processor as of 2.0 SP1, while for I/O threads the value is always 1,000.

Due to the automatic throttling of runnable threads, it's not too bad to have a large number of I/O threads waiting. Windows will ensure only the optimal number of them execute work. Contrast this with worker threads, where all of them fetch and execute work until they are explicitly told to shut down. You might also be curious about the fairly sizeable change in worker thread maximum from 2.0 to 2.0 SP1 (25 to 250 per processor). There's a good reason for it: we'll return to this in a few paragraphs' time.

CLR hosts often override these defaults automatically. In fact, the ASP.NET 2.0 "autoconfigure" process sets the minimums to 50 per processor and maximums to 100 per processor (the old values, and the ones still listed in the `machine.config` template, are 1 per processor for the minimums and 20 per processor for the maximums). Just as you can change the values yourself, most hosts also let you override the defaults through host specific configuration. The `processModel` element in the `machine.config` file lets you instruct ASP.NET to use different minimum and maximum values, for example.

```
<configuration> ...
  <system.web> ...
    <processModel
      maxWorkerThreads="..."
      minWorkerThreads="..."
      maxIoThreads="..."
      minIoThreads="...">
    />
  </system.web> ...
</configuration>
```

The host specific configurations apply only to programs running in the respective host. Setting the `machine.config` settings in the shown way only works for ASP.NET, that is, not all programs running on the machine that use the thread pool, for example.

You can also change these values programmatically. The `ThreadPool` class offers the static methods `GetMaxThreads` and `GetMinThreads` so that you can read the current settings, and `SetMaxThreads` and `SetMinThreads` to modify them. The minimum thread count APIs were added in the .NET Framework 1.1, while the maximum thread count APIs were added in the

.NET Framework 2.0. There is also a `GetAvailableThreads` API that returns the number of threads that are currently not busy executing work.

```
public static class ThreadPool {
    public static void GetAvailableThreads(
        out int workerThreads,
        out int completionPortThreads
    );
    public static void GetMaxThreads(
        out int workerThreads,
        out int completionPortThreads
    );
    public static void GetMinThreads(
        out int workerThreads,
        out int completionPortThreads
    );
    public static bool SetMaxThreads(
        int workerThreads,
        int completionPortThreads
    );
    public static bool SetMinThreads(
        int workerThreads,
        int completionPortThreads
    );
}
...
}
```

Notice that I previously said the pool's default is 250 "per processor." The per processor part is changed internally. So if you have a 4 processor machine and ask for the maximum worker thread count, it will return the number 1,000. Similarly, you must do any such math before providing a new value via the `SetMaxThreads` API.

For many programs, the defaults will suffice. During performance testing and analysis, it's common to experiment with different values based on the workload specific rate of blocking. In theory, having one thread per processor will yield the best possible performance (due to less context switching and cache thrashing). But in practice, threads routinely block. When a thread blocks, the thread pool needs to have another one to process other work or else an entire processor could be wasted. Having too few threads can, therefore, cause low processor utilization. If a thread blocks and there is work in the queue, you'd like the thread pool to quickly respond by

throwing another thread at the queue. On the other hand, having too many threads can cause high context switch overhead and a large number of cache misses. If threads are always compute bound, it's wasteful to have more threads than the number of processors. And there's a delicate balance because when a thread blocks, who can say for how long it will remain blocked? Introducing a new thread right away might be overkill. The thread pool weighs many factors when creating threads, and the only way to influence this behavior is by changing the minimum and maximum settings.

Aside from just performance motivations, there are also two common issues that usually motivate a change of the default values. With the new default of 250 worker threads per processor, one of them has mostly gone by the wayside.

Deadlocks Caused by a Low Maximum. The first common problem is using up the maximum number of threads. As described earlier, the thread pool stops creating new threads once its current count reaches the maximum. It is possible to deadlock your program if the maximum is too low, which is why the CLR 2.0 SP1 increased the default number of worker threads from 25 to 250 per processor. More often than not, this deadlock represents an architectural flaw, particularly if it happens deterministically, particularly if it occurs with the maximum set to 250.

To illustrate, consider this example

1. Thread t0 queues a work item w0 to the thread pool.
2. w0 queues 32 new work items w1..w32 to the thread pool.
3. w0 waits for w1..w32 to complete, by blocking the thread pool thread.

Depending on what w1..w32 do when they get assigned to a thread pool thread, and the number of maximum threads, this program might deadlock. If the maximum was set to 25, then all 32 work items cannot be running concurrently. But maybe that's OK: the first 24 would run; then, as some of them finish, the remaining ones would execute. But what if the thirty-second work item needs to set a flag that all of the other threads read before completing? This program will never finish. It's not difficult

to identify this problem after it's happened, but it isn't completely obvious before that. Here's a code snippet of this very situation.

```
using System;
using System.Threading;

class Program
{
    public static void Main()
    {
        ManualResetEvent outerEvent = new ManualResetEvent(false);
        ThreadPool.QueueUserWorkItem(delegate
        {
            ManualResetEvent innerEvent = new ManualResetEvent(false);

            // Queue 32 new work items:
            for (int i = 0; i < 32; i++)
            {
                ThreadPool.QueueUserWorkItem(delegate(object state)
                {
                    int idx = (int)state;
                    // Do some work...
                    Console.WriteLine("w{0} running ...", idx);

                    if (i == 31)
                    {
                        // Last one sets the event.
                        innerEvent.Set();
                    }
                    else
                    {
                        // All others wait.
                        innerEvent.WaitOne();
                    }
                }, i);
            }

            // Wait for them to finish:
            innerEvent.WaitOne();
            outerEvent.Set();
        });
    }

    Console.WriteLine("Main thread: waiting for w0 to finish");
    outerEvent.WaitOne();
}
}
```

This is really terrible code. If you run it, you'll see what happens. Because all work items wait for the last one to set the event, the thirty-second work item has to be scheduled in order to unblock all of those threads. But for the thirty-second work item to run, the thread pool would have to create 33 threads. Depending on the maximum number of threads, this program may never finish. (You'll also note how slowly new threads are introduced due to the throttling of one thread per 500 milliseconds after exceeding the processor count. That's the second common problem with the thread pool, which we'll return to soon.)

As I noted earlier, this represents a serious design flaw in your program. You should avoid as much interdependency between work items as is possible, and you should strive to avoid blocking thread pool threads. While a worthy goal, it isn't always completely possible to achieve. Many components use the thread pool internally, so it's often hard to predict how much slack in the number of thread pool threads you will need to avoid this situation. That's the main reason the CLR upped the default maximum number of worker threads so high. It's not that the CLR team expects most programs to use this many threads, but rather it avoids unexpected deadlocks in stressful cases.

ASP.NET 2.0 actually offers a configuration setting to deal with this situation. In the `machine.config`, you will find the `httpRuntime` element with the `minFreeThreads` attribute.

```
<configuration>
  <system.web>
    <httpRuntime minFreeThreads="..." />
  </system.web>
</configuration>
```

Setting this ensures that a certain number of thread pool threads are not used to execute Web page requests so that they are free to run asynchronous work. Why would you want to do this? Well, it's fairly common for Web pages to use asynchronous actions: to do some I/O, like communicate with another Web server or read files off the disk. This often uses the thread pool. And the Web page itself is being run off the thread pool. If it weren't for the `minFreeThreads` setting, you would be continuously running into the same problem noted above if any of those page requests queued work to the thread pool. As with the general case above, relying too heavily on `minFreeThreads`

probably indicates an architectural problem in your Web site. ASP.NET 2.0 offers a feature called asynchronous pages that can help avoid the problem altogether, as reviewed in the next chapter.

Delays Caused by a Low Minimum. Another common problem with the thread pool is an artifact of the way threads are created. As noted, the thread pool throttles its creation of new threads at a rate of 1 thread per 500 milliseconds once the thread count has exceeded the number of processors on the machine. For irregular workloads that sometimes need more threads than processors (e.g., for work that blocks), this can present some problems. Imagine this case.

1. A 4-processor Web server has been rebooted and the process just spun up.
2. Sixteen new Web requests arrive almost simultaneously.
3. The CLR thread pool quickly responds by creating the first 4 threads as the new work gets queued up without delay because there is no throttling when the number remains below the number of processors.
4. For whatever reason, each of those 4 actively executing requests block.
5. After 500 milliseconds, the CLR thread pool notices the requests are blocked and responds by creating a single thread to service the fifth request. It creates just 1 thread, mind you, not 4.
6. After another 500 milliseconds, assuming the other 5 threads are still blocked, the thread pool introduces another thread to service additional work.
7. And so on.

Depending on the length of blocking, this could be pretty bad. Blocking for longer than 500 milliseconds is a lifetime, but it can happen. And I've just thrown out an extreme case to make the point. Less extreme cases can suffer from the effects of this throttling too.

Ignoring the fact that this application has seemingly been poorly architected—asynchronous pages should likely be used, as noted earlier—the users of this Web application probably aren't going to be very happy.

Assuming the first 15 requests block for a lengthy period of time, the user who submitted the sixteenth request might have to wait 6 seconds for their request to get serviced (each of the 12 threads after the first 4 takes 0.5 seconds to be created). If the server in this example has a constant load and the workload is regular (i.e., most Web page requests have the same blocking frequency), the pool will eventually become primed with the optimal number of threads, and we should see a reduction in these kinds of delays. But many programs exhibit volatile loads, especially servers. It's common for many applications to have heavy usage during certain hours of the day and be nearly vacant during other hours. Usually it's best if your program can react quickly to these sudden changes in load, otherwise your users will be treated to frustrating and unpredictable delays. The throttling used here represents a fundamental inability in the CLR thread pool's ability to deal with such volatile loads.

Believe it or not, this is such a common source of problems that several Microsoft Support Knowledgebase articles have been generated. And this is the reason for the fairly large discrepancy in ASP.NET 2.0's default minimum number of threads and the unhosted CLR's default (50 per processor versus 0, respectively), and is certainly a reason for you to consider changing the default minimum values yourself. Note that having too large a minimum causes a lot of problems too, so you shouldn't take this step without careful consideration (and only if you've observed a true problem). Each thread consumes stack space, which will get swapped out frequently if the minimum is very high, increasing the number of page faults, which means more I/O (and lower CPU utilization). Having too many threads fighting for the queue will cause context switching overhead and cache effects, as noted already. If you decide you must change it, there really isn't any magic number: you should experiment, measure, refine, measure, and so on.

Debugging

There is a !threadpool SOS extension command in Visual Studio and Windbg. Running it prints out some very basic information, including the last CPU utilization sample that the pool's daemon thread observed, the number of active timers, and the total, running, idle, minimum, and maximum thread counts for the worker and I/O thread pools. Unlike the native thread pool debugging

support, there is no easy way to inspect the contents of the pool's queues. Nevertheless, this basic information is enough to give you an idea if the pool has become deadlocked, among other things.

A Case Study: Layering Priorities and Isolation on Top of the Thread Pool

Two commonly asked for features that the CLR thread pool does not support are prioritization of work items (i.e., asking that the thread pool prefer to run one task over another) and isolation of queues between different AppDomains and /or components inside of a process. Since the CLR doesn't provide these features out-of-the-box (no priorities and it always shares the same pool across all AppDomains in the process), let's briefly explore what it takes to build these on top of the existing pool. It's not difficult.

While one approach is to build an entirely new thread pool, you then have to worry about many of the issues the CLR pool already takes care of: load balancing between AppDomains, thread creation and deletion, and so on. The approach we will explore is much simpler, and can be summarized as follows.

- When somebody queues a work item to our custom thread pool, which we'll call the `ExtendedThreadPool`, we will queue the callback in our own custom work queue and call the CLR thread pool's `QueueUserWorkItem` function. The key difference here is that we'll pass our own callback function to the CLR thread pool, which dispatches work based on priority and isolation between pools.
- There is one per AppDomain `ExtendedThreadPool` object, but users of our pool can also create their own `ExtendedThreadPool` objects. The implementation ensures fair processing of all queues in the AppDomain by round robinning between all of them inside the custom callback.
- We support three priorities—low, normal, and high—passed as an enumeration argument to our queuing function. Each `ExtendedThreadPool` object contains three work queues, one for each priority. (A priority queue data structure would have been better, but to cut down on the code we have to show we'll process individual queues in priority order.)

Listing 7.1 contains the code for our custom pool.

LISTING 7.1: A custom thread pool with isolation and priorities

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

// We support three priorities: Low, Normal, High.
public enum WorkItemPriority
{
    Low = 0,
    Normal = 1,
    High = 2
}

public class ExtendedThreadPool
{
    // One global list of weak refs to registered pools.
    private static List<WeakReference> s_registeredPools =
        new List<WeakReference>();

    // The default pool object.
    private static ExtendedThreadPool s_defaultPool =
        new ExtendedThreadPool();

    // The next pool we will service.
    private static int s_currentPool = 0;

    // Each pool is just comprised of a queue of work items.
    private Queue<WorkItem>[] m_workItems;

    public ExtendedThreadPool()
    {
        // Initialize our work queues.
        m_workItems = new Queue<WorkItem>[
            ((int)WorkItemPriority.High) + 1];
        for (int i = 0; i < m_workItems.Length; i++)
            m_workItems[i] = new Queue<WorkItem>();

        // And register the pool globally.
        lock (s_registeredPools)
        {
            s_registeredPools.Add(new WeakReference(this));
        }
    }

    // Get the one default per-AppDomain pool.
    public ExtendedThreadPool Default {
```

```
    get { return s_defaultPool; }
}

// Convenience methods that use the default pool.
public static void DefaultQueueUserWorkItem(
    WaitCallback callback, object state)
{
    DefaultQueueUserWorkItem(
        callback, WorkItemPriority.Normal, state);
}

public static void DefaultQueueUserWorkItem(
    WaitCallback callback, WorkItemPriority priority, object state)
{
    s_defaultPool.QueueUserWorkItem(callback, priority, state);
}

// Queue a work item for the target pool.
public void QueueUserWorkItem(WaitCallback callback, object state)
{
    QueueUserWorkItem(callback, WorkItemPriority.Normal, state);
}

public void QueueUserWorkItem(
    WaitCallback callback, WorkItemPriority priority, object state)
{
    Queue<WorkItem> q = m_workItems[(int)priority];
    lock (q)
    {
        q.Enqueue(new WorkItem(callback, state, this));
    }
    ThreadPool.UnsafeQueueUserWorkItem(s_dispatchCallback, null);
}

private static WaitCallback s_dispatchCallback = DispatchWorkItem;
private static void DispatchWorkItem(object obj)
{
    WorkItem? work = null;
    do {
        // We just round robin between the pools.
        int poolId = Interlocked.Increment(ref s_currentPool);
        WeakReference poolRef;
        lock (s_registeredPools)
        {
            poolRef = s_registeredPools[
                poolId % s_registeredPools.Count];
        }

        ExtendedThreadPool pool =
            (ExtendedThreadPool)poolRef.Target;
```

```

        if (poolRef.IsAlive) {
            // Grab the next item out of the queue and dispatch it.
            for (int i = (int)WorkItemPriority.High;
                i >= (int)WorkItemPriority.Low;
                i--)
            {
                Queue<WorkItem> q = pool.m_workItems[i];
                lock (q)
                {
                    if (q.Count > 0) {
                        work = q.Dequeue();
                        break;
                    }
                }
            }
        }

        // Keep looping until we find work.  Because
        // DispatchWorkItem will ALWAYS execute once (and only
        // once) per registration, we don't have to worry about
        // infinite loops.
    }
    while (work == null);

    // Now just run the callback.
    work.Value.m_callback(work.Value.m_state);
}

struct WorkItem
{
    internal WaitCallback m_callback;
    internal object m_state;
    internal ExtendedThreadPool m_pool; // To keep our pool alive.

    internal WorkItem(WaitCallback callback, object state,
                      ExtendedThreadPool pool)
    {
        m_callback = callback;
        m_state = state;
        m_pool = pool;
    }
}

```

A notable limitation with this example is that it doesn't properly capture and use `ExecutionContexts` when running work items. In that sense, is more similar to `UnsafeQueueUserWorkItem` than `QueueUserWorkItem`.

One point is worth clarifying since it is apt to create confusion. Because we register each pool with a global list, we use `WeakReference` objects to

refer to the pools. If we didn't, we'd have a leak on our hands: our global list would keep every pool ever created alive, even if all other references went way. Notice that we do store a strong reference from each `WorkItem` queued to a pool, however. This ensures every work item queued to a pool will run before the pool object is collected, which means that users of the pool don't have to worry about trying to synchronize with outstanding callbacks.

Performance When Using the Thread Pools

Both the native and CLR thread pool implementations have enjoyed numerous performance improvements over the years. For sake of discussion, there are two basic metrics we consider.

1. The raw throughput of queuing work items.
2. The throughput of executing work items from the queue.

The first is important because many parallel algorithms of the kind we look at in the Algorithms Section of this book make frequent calls to queue new work items. Substantial overhead here stretches the sequential amount of work done by any given thread, particularly as many such algorithms must queue more than one work item. The second is also important because the overhead imposed on each work item can make concurrency look less attractive, particularly for very fine-grained work items. Both limit the possible parallel speedups that can be realized and are affected by adding more processors: as more processors are added, there may be more contention for enqueueing new work items (metric 1) in addition to dequeuing work items for execution (metric 2). We will take a quick look at scalability after examining these micro-benchmark style metrics.

In the native code arena, the move to Vista brings with it vastly better performance all around. This is primarily due to the thread pool's code living in user-mode rather than kernel-mode, incurring fewer kernel transitions. Even programs still using the legacy APIs but running on Windows Vista will benefit from this new architecture, because the old APIs are just reimplemented in terms of the new ones.

The CLR's thread pool has also had some large performance improvements over the years. Considering the first metric, from 1.1 to 2.0 the performance distance between `QueueUserWorkItem` and `UnsafeQueueUserWorkItem` was shortened dramatically. It used to be the case that `QueueUserWorkItem` was more than twice the cost of `UnsafeQueueUserWorkItem`, but in 2.0 this was reduced to about 15 to 30 percent more costly, on average. That margin is certainly not 0 percent, but it's much better. This comparison is a little unfair because `QueueUserWorkItem` in 2.0 actually costs less than `UnsafeQueueUserWorkItem` did in 1.1, so programs that use `QueueUserWorkItem` saw a dramatic increase in performance when moving to 2.0 without any other changes.

In terms of the second metric, the CLR thread pool has been completely re-architected in the .NET Framework 2.0 SP1. There are now fewer transitions into and out of the runtime for both general work item callbacks in addition to I/O completion callbacks. Work dispatch for the managed thread pool was already very lean, but for some scenarios this change will lead to many improvements in work dispatch throughput. This is particularly true of I/O completion callbacks and will be much more noticeable for very short callbacks.

Here are two graphs comparing the relative throughput of the various thread pools: Windows Vista, the legacy pool in Windows XP SP2, and the safe and unsafe APIs on the CLR 1.1, 2.0, and 2.0 SP1. The numbers have been normalized so that the pool with the best performance will show as 100 percent and all others have been compared against that and will have a smaller percentage. As noted earlier, we consider throughput in the single threaded sense and do not analyze the scalability of the algorithms as more and more processors get involved.

Figure 7.1 shows the throughput of simply queuing work items to the pool.

As we can see, the Vista thread pool far outperforms the other pools in this regard. The CLR 1.1 had the worst performance and has gotten better and better with each subsequent release. The story is different in the callback throughput department, shown in Figure 7.2.

Let me note that this graph may be deceiving at first. This measures thread pool imposed overheads for callbacks that do absolutely no work at

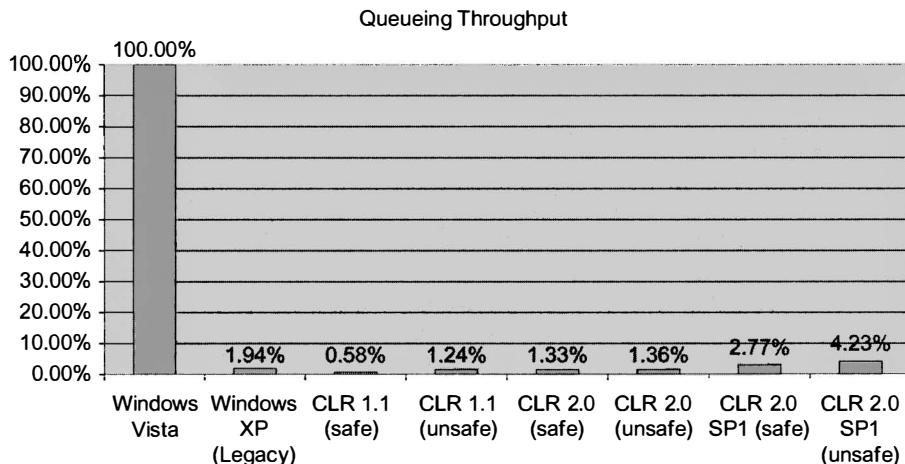


FIGURE 7.1: Throughput of queuing work items to the pool

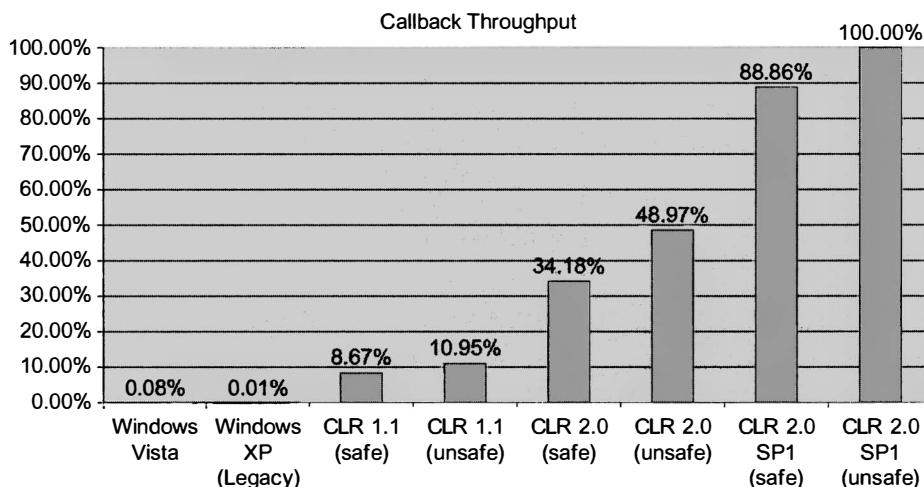


FIGURE 7.2: Throughput of callback execution inside the pool

all on a single CPU system. As the size of the work that the callback performs increases, the impact that these overheads make on the overall throughput decreases quite a bit. And because it's on a single CPU system, it doesn't measure synchronization interaction at all either.

In this case, we can see that the CLR's thread pool has made successfully larger improvements over the years and does better than both the Vista and XP thread pools in raw callback dispatch throughput. The

Windows XP thread pool has, by far, the worst performance of the bunch. Though the difference between Vista and XP appears small in this graph, in reality, the XP thread pool only provides 12 percent of the callback throughput of Vista.

We will conclude by looking at some scaling numbers. We compare the execution time of running N tasks each comprising of C cycles on a single thread versus queuing each of the N tasks to run on the P thread pool threads, where P is the number of processors on the machine. Each of the threads will receive N/P tasks and, for each one, run C cycles' worth of simulated work. In all measurements, we show the CLR 2.0 SP1 and Windows Vista thread pools side-by-side, and, in all cases, prime the pools to ensure we don't measure the cost of lazily allocating the threads.

In summary, the single threaded case will execute in roughly $O(NC)$ time, while the thread pool case will execute in $O(Q + (CNS)/P)$, where Q is the overhead that results from using the pool (we measure the calls to `ThreadPool.QueueUserWorkItem` in our accounting, which means Q is actually some factor of N) and S is the overhead that results on the thread pool for each item dequeued. Sadly, this isn't a constant factor: it depends heavily on contention to dispatch work items from the shared queue. This depends on the size of individual tasks.

In the Figure 7.3, the y -axis represents C , and the abscissa represents the "parallel speedup," a term we will become more familiar with in subsequent chapters. This is the time to execute on 1 thread divided by the time to execute on many threads. The numbers were gathered on a 4-core, 2-CPU machine, that is, an 8-way, so we would like to see these values approach 8. We plot 5 different values for N : 8, 100, 1,000, 10,000, and 100,000. Before moving on, please note that these numbers are a snapshot in time on one very specific machine. Try not to read too much into them, particularly comparing the absolute numbers between the managed and the Vista thread pools. Focus on the larger picture.

It is interesting to note the case in which N is 8. We see that the "break even" point occurs when C is around 12,500 for the CLR and 25,000 for Windows Vista: in other words, this is when the speedup exceeds 1.0, and, therefore, the parallel version beats the sequential version in terms of execution time. In the other cases, the degradation at the low end of

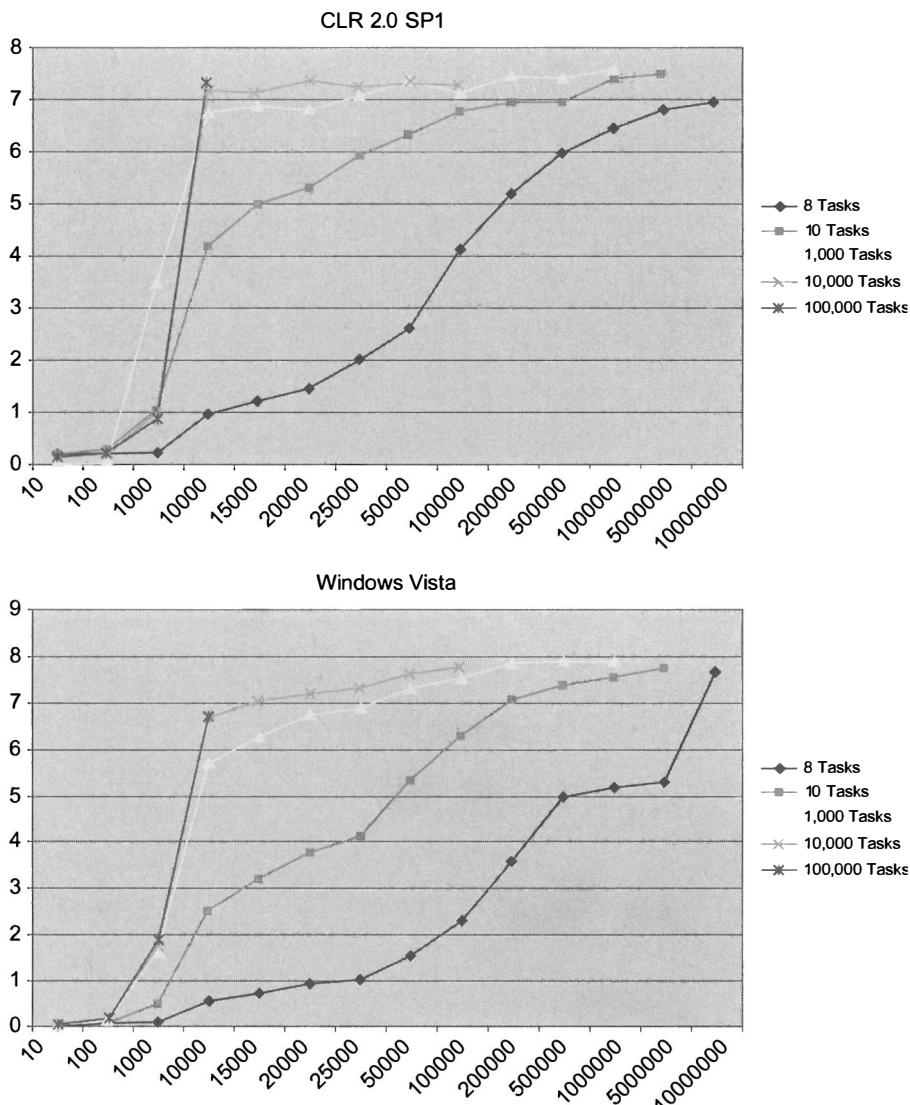


FIGURE 7.3: Parallel speedup with simple work decomposition

the graph is caused by more contention to dispatch work: high values of N with small values of C means the thread pool will have to revisit the shared queue often. In fact, the amount of synchronization is some factor of N .

One useful technique to avoid the synchronization and constant overheads associated with dispatching each new work item is to logically chunk

work together algorithmically rather than relying on the dynamic partitioning of the thread pool. In this example, we could statically partition the number of tasks so that each thread receives the same number of disjoint work items, that is, N/P . In other words, in pseudo-code, rather than doing the following.

```
for (int i = 0; i < N; i++)
{
    ThreadPool.QueueUserWorkItem(delegate(object obj)
    {
        int j = (int)obj;
        ... do work for the 'j'th iteration ...
    }, i);
}
```

We would instead perform a partitioning step up front, and only queue P callbacks.

```
int P = Environment.ProcessorCount;
int stride = (N + P - 1) / P;
for (int i = 0; i < P; i++)
{
    ThreadPool.QueueUserWorkItem(delegate(object obj)
    {
        for (int j = ((int)obj) * stride, c = j + stride;
            j < c && j < N;
            j++)
        {
            ... do work for the 'J'th iteration ...
        }
    }, i);
}
```

Using this technique has the advantage of substantially reducing the burden on the thread pool in terms of dequeuing and running callbacks. We queue up P callbacks, versus N , and see some fairly dramatic improvements as Figure 7.4 illustrates (with equivalent plottings for N and C as the previous graph).

One could argue that this is an unfair comparison. The reason this one looks much better is because we've effectively flattened many smaller work items into fewer larger work items, which is going to scale better. But that's also the point. Sometimes simple solutions can yield particularly large

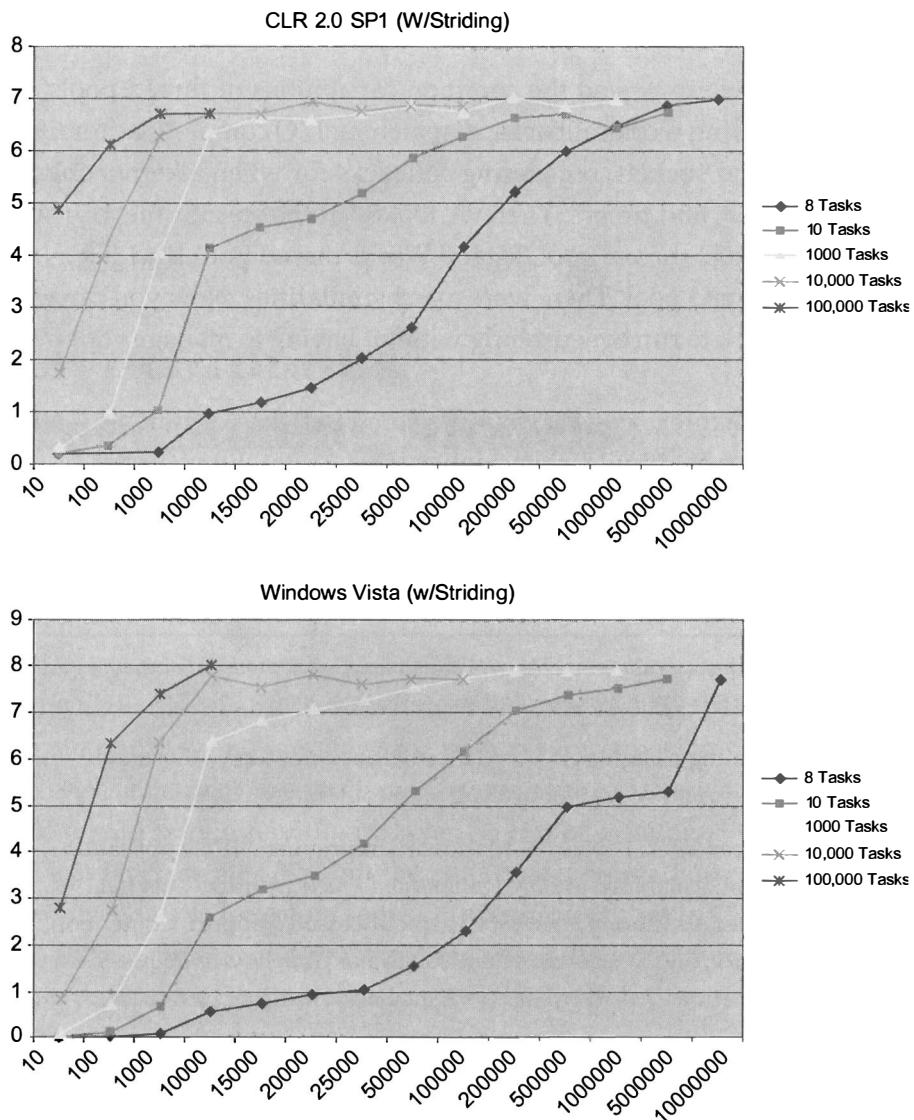


FIGURE 7.4: Parallel speedup with striding based work decomposition

gains. There are also some downsides to this kind of static decomposition: if one of the threads blocks, for instance, then other work items cannot make progress (because you've fixed the decomposition). We'll return to this topic in Chapter 13, Data and Task Parallelism.

Where Are We?

In this chapter, we reviewed the common capabilities of thread pools on Windows—queuing work callbacks, dispatching I/O completions for files, named pipes, and sockets, registering callbacks for when a kernel objects becomes signaled, and timers. Then we looked at the specific mechanisms for the Vista Win32 thread pool, legacy Win32 thread pool, and the .NET Framework’s thread pool. There were many similarities. Now you can easily queue up work to run concurrently without having to manage your own pools of threads.

In the next chapter, we will examine some patterns common to .NET Framework types that build even higher level abstractions on top of the thread pool idea.

FURTHER READING

- K. Cwalina, B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Addison-Wesley, 2006).
- J. Duffy. Implementing a High-perf IAsyncResult: Lock free Lazy Allocation. Weblog article, <http://www.bluebytesoftware.com/blog/> (2006).
- J. D. Meier, S. Vasireddy, A. Babbar, A. Mackman. Improving .NET Application Performance and Scalability. *MSDN Patterns and Practices*, <http://msdn2.microsoft.com/en-us/library/ms998583.aspx>. Microsoft Support. Contention, Poor Performance, and Deadlocks when You Make Web Service Requests from ASP.NET applications. *Microsoft Support Knowledgebase*, KB 821268 (2004).
- Microsoft Support. FIX: Slow Performance on Startup when You Process a High Volume of Messages Through the SOAP Adapter in BizTalk Server 2006 or in BizTalk Server. *Microsoft Support Knowledgebase*, KB 886966 (2004).
- J. Richter. 2007. Implementing the CLR Asynchronous Programming Model. *MSDN Magazine* (2007).

8

Asynchronous Programming Models

In the last chapter, we saw how to efficiently use threads through the higher level abstraction of thread pools. The .NET Framework goes one step further and has standard patterns for exposing the capability to run *asynchronously*. The implementations of this pattern typically use the CLR thread pool internally or layer on top of existing asynchronous OS services (such as file I/O), but the patterns accommodate common coordination needs. We'll explore some OS specific facilities in Chapter 15, Input and Output, but a wonderful attribute about them is that most are exposed using these same common patterns in .NET.

The two most prevalent patterns follow.

- The **asynchronous programming model** (APM) is the most common model and has been around since the inception of the .NET Framework. It is the recommended pattern for most libraries that offer asynchronous versions of certain methods. It is typified by its paired methods, named `BeginFoo` and `EndFoo`, for some synchronous API named `Foo`, and its reliance on the `System.IAsyncResult` interface. It supports a rich set of capabilities, including several different modes of reacting to asynchronous completion.

- The second pattern is called the **event-based asynchronous pattern**, a.k.a. **asynchronous pattern for components** and is meant for UI oriented components that must integrate with progress reporting and cancellation. The distinguishing characteristic for APIs implementing this pattern is the `Async` suffix, in contrast with the `Begin/End` prefix for the APM. This pattern is typically more complicated to implement and also carries some semantic overhead (e.g., requiring transfer back to the GUI thread). It can be simpler from a usage standpoint, however, because the only completion mechanism is event based (unlike the APM, which offers multiple mechanisms); additionally, Visual Studio provides a seamless development experience and makes it easy to hook up event handlers. A related feature, `BackgroundWorker`, implements this pattern and is available for general purpose asynchronous programming (see Chapter 16, Graphical User Interfaces).

If you are creating a new API and trying to choose which pattern to implement, a good rule of thumb is that the APM is best when your target audience is other library developers, whereas the event-based model should be used if your primary target audience is application developers.

In the .NET Framework 3.5, a slight variant is provided that is specific to asynchronous sockets programming. Because it is not a pervasive and commonly used pattern, discussion is deferred to Chapter 15, Input and Output, when we get to the specific asynchronous capabilities of sockets on Windows. In the meantime, let's look at the two common patterns.

Asynchronous Programming Model (APM)

The APM is implemented by several .NET Framework classes to provide a consistent pattern for programming asynchronous operations. The existence of the APM means that in a lot of cases, as a user of concurrency, it's not even necessary for you to think about queuing work separately to the thread pool; it just happens in the implementation of some .NET Framework API that you call in your program. And, as a library developer, providing APM versions of your compute- or I/O-bound operations helps the

users of your APIs similarly take advantage of concurrency with a simple, familiar interface.

Each APM enabled operation offers two special methods. If we have an ordinary synchronous method `Foo`, then implementing the APM version entails two new methods `BeginFoo` and `EndFoo`. The transformation from `Foo` to the APM methods is simple.

- `BeginFoo` accepts the same input arguments as `Foo` with two additional arguments appended, `AsyncCallback callback` and `object state`, and it returns an `IAsyncResult` object. This object offers some convenient operations that allow you to poll or wait for completion. Later we'll look at a standard implementation of `IAsyncResult` that can be reused.
- `EndFoo` accepts the `IAsyncResult` object and has the same return type as `Foo` does. Any exceptions that occur during the asynchronous invocation of `Foo` are caught and then rethrown when `EndFoo` is called. But its primary purpose is to fetch the value returned by the asynchronous call.

The `AsyncCallback` type is just a delegate from the `System` namespace:

```
public delegate void AsyncCallback(IAsyncResult ar);
```

The `callback` is invoked by the APM provider once `Foo` has finished running, making it easy to run some logic that consumes the results. There are other ways to rendezvous with the completion of an asynchronous operation; we'll see more on this later. The `state` is just an opaque object that is accessible inside your callback and/or completion logic. Both `callback` and `state` are always optional arguments, meaning `null` can be passed.

The purpose of `EndFoo` is three-fold. First and foremost, it is responsible for retrieving the value that was returned from `Foo`, so long as the return type `T` is non-void. Second, if an exception occurred during the execution of `Foo`, `EndFoo` will rethrow it so that your program can handle it as it would have if `Foo` had thrown it. Failing to call `EndFoo` means that you're potentially swallowing an exception in your program. And finally, `EndFoo` will clean up resources associated with the asynchronous operation, often

involving a kernel object meant to accommodate waiting. All correctly written implementations of the APM should ensure that, even if `EndFoo` is not called, resources are not leaked. Usually that means having a finalizer or relying on smart resource handles—such as `SafeHandles`—that are already protected.

The `IAsyncResult` interface, also from the `System` namespace, looks like the following.

```
public interface IAsyncResult
{
    object AsyncState { get; }
    WaitHandle AsyncWaitHandle { get; }
    bool CompletedSynchronously { get; }
    bool IsCompleted { get; }
}
```

The properties are straightforward and can be used for the noncallback kinds of completion. `AsyncState` captures what was passed as `state` to the `BeginFoo` method, `AsyncWaitHandle` is a kernel object (typically a manual-reset event) that is signaled once the operation completes, `CompletedSynchronously` indicates whether the operation ran synchronously or asynchronously, and `IsCompleted` gets set to true when the operation is done.

Let's take an abstract example of what an APM counterpart for a sequential API looks like. Given a sequential method `Foo`, the transformation is somewhat mechanical.

```
T Foo(U u, ..., V v);
```

The standard APM methods would be:

```
IAsyncResult BeginFoo(U u, ..., V v, AsyncCallback callback, object state);
T EndFoo(IAsyncResult asyncResult);
```

Looking past the syntax, let's talk about what these things do. `BeginFoo` is responsible for initiating `Foo` to run asynchronously, passing the arguments `U u, ..., V v`. This often means calling `QueueUserWorkItem` with a little wrapper over `Foo` so that success, failure, and completion can all be handled according to APM convention, that is:

```
IAsyncResult BeginFoo(U u, ..., V v, AsyncCallback callback, object state)
{
```

```
FooAsyncResult asyncResult = ...;

ThreadPool.QueueUserWorkItem(delegate
{
    try
    {
        // Store return value on asyncResult so we return on EndFoo.
        T retval = Foo(u, ..., v);
        asyncResult.SetReturnValue(retval);
    }
    catch (Exception e)
    {
        // Store exception on asyncResult so we rethrow on EndFoo.
        asyncResult.SetException(e);
    }
    finally
    {
        // Signal completion.
        asyncResult.SignalDone();
    }
});

return asyncResult;
}
```

This is meant to illustrate the flow of control. Notice that `BeginFoo` could return before, while, or after `Foo` finishes executing, depending on the way work is scheduled on the thread pool. The meat of the implementation is omitted: the `FooAsyncResult` class. We'll explore a sample implementation of `IAsyncResult` later. Also, we don't necessarily need to run `Foo` on the thread pool. In some specific circumstances, we could use Windows I/O completion ports for asynchronous I/O, for instance, so that no thread ever has to block.

Rendezvousing: Four Ways

After a thread kicks off asynchronous work, there is a decision to make: How will we rendezvous with the completion of that work so that the `EndFoo` method can be called, possible exceptions handled, and the return value processed in an appropriate way? This rendezvous may or may not involve the original thread. In fact, four basic rendezvous patterns are supported:

1. A thread can make a call to `EndFoo` directly. The APM provider is responsible for doing the right thing in this method: if already

completed, it will return or throw right away; otherwise, it will block waiting for completion. When the call returns or throws, the asynchronous operation is complete.

2. Any thread with access to the `IAsyncResult` can use the `AsyncWaitHandle` to block until the concurrent work has finished.
3. Any thread with access to the `IAsyncResult` (usually the thread that started the work) can “poll” for completion by checking the `IsCompleted` flag. When the asynchronous work has finished, the `IsCompleted` flag will be set to `true`, and it is then safe to call `EndFoo`.
4. Finally, a callback may be supplied to `BeginFoo`, which is called when `Foo` finishes. This typically executes on a thread pool thread, and inside the callback code you can make a call to `EndFoo` to retrieve the results.

You can also mix a combination of these things, though you have to be somewhat careful. You must ensure no two threads ever call `EndFoo` on the same `IAsyncResult`. While some APM providers may handle this situation, it is not a standard part of the pattern. Should you depend on one particular implementation handling this, you’re apt to encounter race conditions and compatibility problems down the road.

Now we’ll look at an example program that uses a synchronous method `Foo` and, specifically, how we can morph the program into using `BeginFoo` and each of these completion mechanisms instead. This is more of a case study walkthrough of the completion mechanisms and will be useful to illustrate practical concerns that will arise when you try to consume the APM from your own code. Here is the original synchronous program.

```
T f()
{
    S0;
    T t = g();
    S1;
    return t;
}

T g()
{
    V v = S2;
```

```

T t;
try
{
    t = Foo(v);
    S3(t);
}
catch (SomeException e)
{
    S4;
}
S5;
return t;
}

```

The markers S0 . . . S5 are meant to indicate some set of program statements that are immaterial to the example itself. What is important about them is the control flow and when they will execute. For simplification purposes, imagine that no references to t are found in any of the statements except for S3. That is, the call to Foo produces a value stored in t, which is returned from g to f, and then f returns it without inspecting the value.

Where are the opportunities for asynchronous execution here? The possibility of race conditions and shared resources aside, Foo can run concurrently with respect to at least S5 and S1 due to the lack of control dependence. It can run concurrently with S0 too, but because the call to Foo is dependent on the output of S2, we would need to restructure the code somehow, probably issuing S2 before S0.

We'll now work our way through the rendezvous techniques: from mechanism #1 to mechanism #4. You will find that #1 is generally the least different from the sequential code while #4 is generally the most different.

Mechanism #1: Calling EndFoo Directly

If we wanted S5 to be run concurrently with the call to Foo, S3, and S4, we could change the Foo call to a BeginFoo call and then shuffle the code around slightly.

```

T f() { ... remains the same ... }

T g()
{
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
}

```

```

T t;
try
{
    t = EndFoo(asyncResult);
    S3(t);
}
catch (SomeException e)
{
    S4;
}
return t;
}

```

Now we run S5 concurrently with Foo, and “join” with the work before returning the value. Astute readers will notice a subtle distinction between the original code and this new version. Whereas in the original example, if Foo threw an exception other than `SomeException`, we would never get to run any of the code in S5, in this rewritten version, S5 is run before we even check for exceptions. If there were some set of effects that S5 made that needed to be undone in the case of unhandled exceptions, we would have to add the code as an extra exception handler, somewhat transaction-like. We’re also making a ton of assumptions about ordering: that it’s actually safe to run S5 in parallel with Foo and so on.

There is still opportunity for additional concurrency that is going completely unrealized. Recall we said S1 can run concurrently with Foo too. But doing that requires breaking the clean split between f and g. This is unfortunate, but speaks to the fact that the APM can be **viral** in nature: that is, it can pervade your program if care is not taken. This rewrite of the above code now permits both S5 and S1 to run concurrently with respect to Foo, but it requires that we tightly couple f and g. In fact, I’ve just fused them into a single function.

```

T f()
{
    S0;
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
    S1;
    T t;
    try
    {
        t = EndFoo(asyncResult);
    }
}

```

```

        S3(t);
    }
    catch (SomeException e)
    {
        S4;
    }
    return t;
}

```

Notice that g is completely gone. Some of the other completion mechanisms make this more palatable, such as enabling g to pass f a completion routine for the callback method. But no matter what you do, the clean split between f and g must change. All of the caveats about ordering and undoing side effects mentioned for S5 also apply to S1 in this example too.

Mechanism #2: Calling AsyncWaitHandle's WaitOne Method

The only real advantage the AsyncWaitHandle rendezvous mechanism offers over calling EndFoo is that you have more control over how the thread waits. You can use timeout based waits or something like WaitHandle's WaitAll or WaitAny.

For instance, we might use a wait with a timeout in order to provide regular status updates to the user about the progress of the operation, say, every 100 milliseconds:

```

T f() { ... remains the same ... }

T g()
{
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
    while (!asyncResult.AsyncWaitHandle.WaitOne(100, false))
    {
        // Notify user of progress.
    }
    T t;
    try
    {
        t = EndFoo(asyncResult);
        S3(t);
    }
    catch (SomeException e)
    {
        S4;
    }
}

```

```
        return t;
    }
```

(Later in this book, in Chapter 16, Graphical User Interfaces, we'll examine a useful abstraction with the name of `BackgroundWorker`. This is a component that is specifically meant for maintaining responsive UIs with progress indicators, cancellation, and so on.)

Similarly, we could use a timeout to put an actual upper bound on the time we're willing to wait for `Foo`. Say we are willing to wait for only a maximum of 500 milliseconds for `Foo` to complete and, if this timeout expires, we will throw an exception of some sort:

```
T f() { ... remains the same ... }

T g()
{
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
    if (!asyncResult.AsyncWaitHandle.WaitOne(500, false))
    {
        throw new TimeoutException(...);
    }
    T t;
    try
    {
        t = EndFoo(asyncResult);
        S3(t);
    }
    catch (SomeException e)
    {
        S4;
    }
    return t;
}
```

This approach has one big problem. Even if we timed out, we really should handle calling `EndFoo` so that exceptions from the call to `Foo` are handled and the `IAsyncResult` resources can be cleaned up. It would be terrible if `Foo` threw a `TheMachineIsOnFireException` and the thread calling `f` and `g` caught and swallowed the `TimeoutException` thrown by `g`, without `EndFoo` ever having been called. One way of handling this is to queue

the exception handling part of the continuation on to the thread pool just before throwing the exception.

```
T f() { ... remains the same ... }

T g()
{
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
    T t;
    if (!asyncResult.AsyncWaitHandle.WaitOne(500))
    {
        ThreadPool.QueueUserWorkItem(delegate
        {
            try
            {
                EndFoo(asyncResult);
            }
            catch (SomeException e)
            {
                S4;
            }
        });
        throw new TimeoutException(...);
    }
    try
    {
        t = EndFoo(asyncResult);
        S3(t);
    }
    catch (SomeException e)
    {
        S4;
    }
    return t;
}
```

This approach makes some assumptions and isn't universally appealing. We're assuming that it's OK to run S4 at any arbitrary point in the future, including after the calls to f and g have returned. It also is not semantically equivalent to the sequential program. We're also blocking a thread pool thread. If the timeout may have happened because of a deadlock, we may completely tie up the thread pool. What we really want is a way to cancel the work after 500 milliseconds, and to go back to waiting on it (hoping that

cancellation is responsive). We will explore cancellation a bit more in Chapter 13, Data and Task Parallelism.

To take this example further, say we wanted to run two APM-capable operations, Foo and Bar concurrently, and wanted to handle them in whatever order they complete. This is another example where the `AsyncWaitHandle` offers an advantage because we can wait for either (or both) to complete with `WaitHandle`'s `WaitAny` and `WaitAll` methods. If this were the simple synchronous version of the code we wanted to modify to be asynchronous:

```
S0(Foo(...));
S1(Bar(...));
```

Then the APM version using `WaitAny` would go as follows.

```
IAsyncResult fooAsyncResult = BeginFoo(...);
IAsyncResult barAsyncResult = BeginBar(...);

WaitHandle[] handles = new WaitHandle[]
{
    fooAsyncResult.AsyncWaitHandle,
    barAsyncResult.AsyncWaitHandle
};

int awoken = WaitHandle.WaitAny(handles);
if (awoken == 0)
{
    S0(EndFoo(fooAsyncResult)); // Won't block.
    S1(EndBar(barAsyncResult)); // May block.
}
else
{
    S1(EndBar(barAsyncResult)); // Won't block.
    S0(EndFoo(fooAsyncResult)); // May block.
}
```

Of course things become more complicated if we need to handle the possibility of failure coming from `EndFoo` or `EndBar`. Would we block waiting for the other to finish inside of a `finally` block? This is a difficult question to answer, but without doing something like this we'd run the risk of losing exceptions. The topic of cancellation once again comes up.

Mechanism #3: Polling the IsCompleted Flag

The IAsyncResult object offers an IsCompleted flag, of type bool. When the asynchronous work has finished, this gets set to true. So your rendezvous logic can guard the call to EndFoo on this value, allowing you to avoid blocking and instead do other work while the asynchronous computation completes.

```
T f() { ... remains the same ... }

T g()
{
    V v = S2;
    IAsyncResult asyncResult = BeginFoo(v);
    S5;
    while (!asyncResult.IsCompleted)
    {
        S6;
    }
    T t;
    try
    {
        t = EndFoo(asyncResult);
        S3(t);
    }
    catch (SomeException e)
    {
        S4;
    }
    return t;
}
```

In this example, we introduced a new statement, S6, that does something useful while the concurrent operation is executing. This is a little like the waiting with timeout example shown before (where we provided status to the user) with one distinction: checking IsCompleted does not block the calling thread. You must use this tactic with care: if S6 is something computationally expensive, it may end up using CPU resources that could have otherwise been used to finish running Foo. It would also be bad if S6 were an empty statement, because it amounts to a completely inappropriately written spin wait.

Mechanism #4: Callbacks

The callback rendezvous technique can be more complicated to deal with than the others. It requires a style of programming referred to as **continuation passing style (CPS)**, where the continuation of whatever you would have done after `Foo` completed (in a synchronous program) has to be represented with callback delegate instead. It can be difficult to save enough information at the time of a `BeginFoo` call to be able to resume the entire logical continuation of work asynchronously at some point in the future. Moreover, the thread pool is meant only for short bursts of work, so you probably wouldn't want to save the whole logical continuation (i.e., the whole stack's worth), meaning this technique works best when the amount of work to do in response is fairly small (much like an event handler). The other mechanisms, by contrast, allow you to write your code similar to a synchronous program, with little regions carved out where the work happens asynchronously.

Attempting to use the callback rendezvous approach for this particular sample highlights these challenges. Several callers in the current stack may depend on the output of calling `Foo`, because it is returned from both `f` and `g`. We need to move the continuation statements `S3`, `S4`, `S5`, and `S1` in the callback, requiring a lot of code refactoring to turn `Foo` into `BeginFoo`. And that alone is insufficient: since the caller of `f` also needs the output of `Foo`, we would need to make the things that happen after `f` returns part of the continuation too, possibly requiring callers to supply their own callbacks as arguments. Depending on the amount of code on the callstack you own, this may be possible, but this can get very complex very quickly.

For purposes of discussion, and to illustrate when a callback might be useful, pretend `g` looks like the following.

```
void g()
{
    V v = S2;
    try
    {
        T t = Foo(v);
        S3(t);
    }
    catch (SomeException e)
    {
        S4;
```

```
    }
    S5;
}
```

Now it's simple and `f` doesn't enter into the equation (because it doesn't depend on the value returned by `g`). Now we can just ensure the body of `g` is captured correctly into a continuation.

```
void g()
{
    V v = S2;
    BeginFoo(v,
        delegate(IAsyncResult asyncResult)
    {
        try
        {
            T t = EndFoo(asyncResult);
            S3(t);
        }
        catch (SomeException e)
        {
            S4;
        }
    }, null);
    S5;
}
```

The call to `Foo` has been replaced with a call to `BeginFoo`, kicking off the asynchronous work, and the program continues. This achieves what we sought to achieve in the first mechanism shown, which is that `S1` in `f` is able to run concurrently with `Foo`, and this particular example doesn't require that we break the abstraction between `f` and `g` as we did earlier. In fact, `g` can now run concurrently with code that runs even after `f` returns. This requires some additional thought to avoid race conditions and concurrency bugs, however, particularly if `g` is accessing any global state.

Implementing IAsyncResult

Implementing the APM can be broken into three steps: (1) writing `BeginFoo`, (2) writing `EndFoo`, and (3) implementing the `IAsyncResult` class to tie it all together. We already saw a skeleton of (1) and (2) earlier, so let's focus on the admittedly more difficult task of (3).

There are several existing resources on implementing the APM, most notably the *.NET Framework's Design Guidelines* (see Further Reading). Let's

look briefly at how you would go about it. Anybody doing serious reusable library development should review the *Framework's Design Guidelines* for additional insights and consistency guidelines, both in the area of the APM and for a broader perspective.

Listing 8-1 demonstrates a basic `SimpleAsyncResult` class that can be reused for just about any APM implementation you will ever have to write.

LISTING 8.1: A reusable IAsyncResult implementation, SimpleAsyncResult<T>

```
using System;
using System.Threading;

public delegate T Func<T>();

public class SimpleAsyncResult<T> : IAsyncResult
{
    // All of the ordinary async result state.
    private volatile int m_isCompleted; // 0==not complete, 1==complete.
    private ManualResetEvent m_asyncWaitHandle;
    private readonly AsyncCallback m_callback;
    private readonly object m_asyncState;
    // To hold the results, exceptional or ordinary.
    private Exception m_exception;
    private T m_result;

    private SimpleAsyncResult(
        Func<T> work, AsyncCallback callback, object state)
    {
        m_callback = callback;
        m_asyncState = state;
        m_asyncWaitHandle = new ManualResetEvent(false);

        RunWorkAsynchronously(work);
    }

    public bool IsCompleted
    {
        get { return (m_isCompleted == 1); }
    }

    // We always queue work asynchronously, so we always return false.
    public bool CompletedSynchronously
    {
        get { return false; }
    }
}
```

```
public WaitHandle AsyncWaitHandle
{
    get { return m_asyncWaitHandle; }
}

public object AsyncState
{
    get { return m_asyncState; }
}

// Runs the thread on the thread pool, capturing exceptions,
// results, and signaling completion.
private void RunWorkAsynchronously(Func<T> work)
{
    ThreadPool.QueueUserWorkItem(delegate
    {
        try
        {
            m_result = work();
        }
        catch (Exception e)
        {
            m_exception = e;
        }
        finally
        {
            // Signal completion in the proper order:
            m_isCompleted = 1;
            m_asyncWaitHandle.Set();
            if (m_callback != null)
                m_callback(this);
        }
    });
}

// Helper function to end the result. Only safe to be called
// once by one thread, ever.
public T End()
{
    // Wait for the work to finish, if it hasn't already.
    if (!m_isCompleted)
    {
        m_asyncWaitHandle.WaitOne();
        m_asyncWaitHandle.Close();
    }

    // Propagate any exceptions or return the result.
    if (m_exception != null)
        throw m_exception;
}
```

```
        return m_result;
    }
}
```

So what are the interesting parts of this code? The constructor function accepts a `Func<T>` delegate representing the actual work to be done asynchronously. It then initializes our new `SimpleAsyncResult<T>` object and queues this work to run asynchronously with `RunWorkAsynchronously`. If we look inside that function, you'll see that we use the thread pool and call the delegate from within a `try` block. If work succeeds, we store the return value in the `m_result` field of the object; if it throws an exception, we store that in the `m_exception` field. We do not let the exception propagate past our `catch` block; doing so would cause an unhandled exception on the thread pool, triggering a process crash. After either of these situations occurs, we initiate the completion logic.

All APM implementations should perform the same completion steps in the same order:

1. Modify state so that `IsCompleted` will return `true`.
2. Set the `AsyncWaitHandle` so that any waiting threads will be awokened.
3. Invoke the callback supplied by the caller, if any.

It is important to ensure that 1 and 2 have been performed before 3, just in case the callback itself (or the `EndFoo` method) depends on these things having been set.

And of course there's the `End` method. This takes care of waiting for the asynchronous work to complete: the code checks `IsCompleted` first and will only call `WaitOne` on the `AsyncWaitHandle` if it returns `false`. Because calling `WaitOne` is fairly expensive even for an event that has already been set, this is slightly more efficient. After that, we check to see if an exception was thrown (`m_exception`); if so, we rethrow it; otherwise, we return the result yielded by the work delegate (`m_result`).

Note that rethrowing an exception such as this destroys the original stack trace. This is one of the areas where platform support for concurrency is lacking: if the exception goes unhandled, breaking into the debugger will bring you to the `throw m_exception` statement in `SimpleAsyncResult<T>`.

End instead of the statement at which the exception was thrown (asynchronously). In fact, the thread from which the exception was thrown will have been returned to the pool. This means any thread local state, including local variables on the thread's stack, will not be available.

We always return `false` for the `CompletedSynchronously` property. Returning `true` is a relatively obscure situation that doesn't happen much. It must return `true` if the thread being used to execute the callback is the same thread that was used to invoke the `BeginFoo` operation in the first place. Because our code always queues work to run in the thread pool, this isn't ever possible. Some APM implementations are clever enough to run the callback on the current thread if it doesn't make sense to run the code asynchronously. In these cases, your callback could end up using a lot of stack (unexpectedly) if it tries to continue to call `BeginFoo` over and over again from within the completion callbacks. The `FileStream` class's `BeginRead` and `BeginWrite` operations, for example, can result in this behavior because Windows asynchronous I/O may be able to finish the I/O operation so quickly that transferring the callback to another thread isn't necessary. We discuss this possibility more in Chapter 15, Input and Output. Most programs can remain unaware of `CompletedSynchronously`.

Once we have the `SimpleAsyncResult<T>` class, we can wrap it with standard `BeginFoo` and `EndFoo` APM methods. For example, Listing 8.2 demonstrates a simple APM variant of some synchronous `Work` method that calls `Thread.Sleep` and then returns a new random number:

LISTING 8.2: A simple APM implementation using `SimpleAsyncResult<T>`

```
public class SimpleAsyncOperation
{
    public int Work(int sleepyTime)
    {
        Thread.Sleep(sleepyTime);
        return new Random().Next();
    }

    public IAsyncResult BeginWork(
        int sleepyTime, AsyncCallback callback, object state)
    {
        return new SimpleAsyncResult<int>(
            delegate { return Work(sleepyTime); },
            callback,
            state
        );
    }
}
```

```
        );
    }

    public int EndWork(IAsyncResult asyncResult)
    {
        SimpleAsyncResult<int> simpleResult =
            asyncResult as SimpleAsyncResult<int>;

        if (simpleResult == null)
            throw new ArgumentException("Bad async result.");

        return simpleResult.End();
    }
}
```

A significantly more efficient approach to implementing the APM involves lazily allocating the `AsyncWaitHandle` object only when it is requested (i.e., a caller accesses `AsyncWaitHandle` directly or calls `EndFoo` before `IsCompleted` is `true`). Though there are many more complicated examples of how to do this, it is very straightforward with the help of some additional lazy initialization abstractions that we will explore later in Chapter 10, Memory Models and Lock Freedom.

Where the APM Is Used in the .NET Framework

The APM is used in many places in the platform in various ways. Here is a list of some of the most important APM-capable operations in the core assemblies that ship as part of the .NET Framework 3.0 (`mscorlib.dll`, `System.dll`, `System.Core.dll`, `System.Data.dll`, `System.Transactions.dll`):

- All delegate types, by convention, offer a `BeginInvoke` and `EndInvoke` method alongside the ordinary synchronous `Invoke` method. While this is a nice programming model feature, you should stay away from them wherever possible. The implementation uses remoting infrastructure that imposes a sizeable overhead to asynchronous invocation. Queuing work to the thread pool directly is often a better approach, though that means you have to coordinate the rendezvous logic yourself (or use the APM implementation we're about to examine).

- `System.IO.Stream` provides `BeginRead` and `BeginWrite` APM methods. A default implementation is provided on the `Stream` base type so that all of the subclasses in the .NET Framework get `BeginRead` and `BeginWrite` methods for free. `Stream` uses the asynchronous delegate functionality mentioned above. Most streams, notably `FileStream`, override the default behavior to implement more efficient asynchronous operations relying on native Windows asynchronous I/O.
- The `System.Net.Sockets.Socket` class offers a big array of APM methods: `BeginAccept`, `BeginConnect`, `BeginDisconnect`, `BeginReceive`, `BeginReceiveFrom`, `BeginReceiveMessageFrom`, `BeginSend`, `BeginSendFile`, and `BeginSendTo`. Most of these methods take full advantage of the capability Windows provides for network I/O to truly happen asynchronously.
- As of the .NET Framework 2.0, the `System.Data.SqlClient.SqlCommand` type offers APM versions of its primary execution methods: `BeginExecuteNonQuery`, `BeginExecuteReader`, and `BeginExecuteXmlReader`.
- All `System.Net.WebRequest` subclasses support the `BeginGetRequestStream` and `BeginGetResponse` methods. The base class itself throws a `NotImplementedException`, but the three subclasses, `FileWebRequest`, `FtpWebRequest`, and `HttpWebRequest`, provide actual implementations.
- DNS resolution through the `System.Net.Dns` class can be done asynchronously with the `BeginGetHostAddresses`, `BeginGetHostName`, `BeginGetHostEntry`, and `BeginResolve` APM methods.
- `System.Transactions.CommittableTransaction` provides asynchronous commit operations with the `BeginCommit` and `EndCommit` methods.

In addition to all of those libraries, there are areas of the platform that interoperate with the APM in useful ways. One prime example is the ASP.NET asynchronous pages feature.

ASP.NET Asynchronous Pages

ASP.NET 2.0's *asynchronous pages* feature is an interesting case study of how the APM can be used in practice. It's widely recognized as a bad practice to block on a busy server because doing so adds some amount of overhead: a single blocked thread means other requests cannot be serviced, possibly leading to a pileup of them. The thread pool may react by injecting additional threads, also impacting performance. Nonblocking designs—using asynchronous file I/O, and the like—lead to better throughput because threads can continue to process requests while I/O (or other asynchronous work) happens “in the background.”

The asynchronous pages capability allows you to register a pair of `BeginFoo/EndFoo` methods that execute as a page is being rendered. Instead of keeping a thread blocked while the work executes, ASP.NET will let the rendering thread go back to the pool to work on additional requests. Only once the asynchronous work is done will ASP.NET then call the `EndFoo` method to retrieve results and then continue rendering the page with said results in hand.

Everything ASP.NET 2.0 does to allow the asynchronous pages feature could have been written in ASP.NET 1.0 and 1.1, but the features were not nearly as easy to access. Now if you mark your page as `Async="True"`, ASP.NET implements `IHttpAsyncHandler` for you.

```
<%@Page Async="True" ...%>
```

You can then use the `AddOnPrerenderCompleteAsync` method on the `Page` class to register an APM begin/end method pair, and ASP.NET will be careful to let the calling thread go back and service Web requests while the asynchronous operation executes.

```
public void AddOnPreRenderCompleteAsync(
    BeginEventHandler beginHandler,
    EndEventHandler endHandler
);
public void AddOnPreRenderCompleteAsync(
    BeginEventHandler beginHandler,
    EndEventHandler endHandler,
    object state
);
```

Both take event handler delegates, and the second, an optional state parameter.

```
public delegate IAsyncResult BeginEventHandler(
    object sender,
    EventArgs e,
    AsyncCallback cb,
    object extraData
);
public delegate void EndEventHandler(IAsyncResult ar);
```

You can call the `AddOnPreRenderCompleteAsync` method anytime leading up to the `PreRender` event. This registers your begin and end handlers with the current page. After the ASP.NET engine executes the `PreRender` event, it will then proceed to invoking the begin handler, passing the `state` parameter you specified during registration (if any) as `extraData`. The begin handler is responsible for initiating some asynchronous activity and returning an `IAsyncResult` in accordance with the general APM pattern. ASP.NET passes an internally managed callback that, when executed, will cause ASP.NET to use one of its worker threads to call the end handler. The thread is then resumed back to the pool so that it can continue processing Web requests. Once the handler finishes, rendering of the page is resumed.

Event-Based Asynchronous Pattern

If you are providing a higher level component whose target audience is application developers—particularly ones who will be building GUIs—then you should consider exposing the *event-based asynchronous pattern* instead. The APM is meant for lower level framework and library components where flexibility over how completion takes place is desirable. Application developers, however, are typically less concerned with performance and fine-grained control and more concerned with conveniently rendezvousing back to a GUI thread. This is the event-based asynchronous pattern's forte.

The Basics

To implement the event-based pattern instead of the APM, you will append `Async` to your method name. The transformation is similarly mechanical. Take a synchronous method.

```
T Foo(U u, ..., V v);
```

The asynchronous component version of it would look like this.

```
void FooAsync(U u, ..., V v);
```

Optionally, or in addition, extra state can be passed in that will be made available in the completion handler.

```
void FooAsync(U u, ..., V v, object userState);
```

The latter is typically needed if you're going to support multiple outstanding invocations of `FooAsync` as a unique handle to differentiate one completion from another. There is no `IAsyncResult` object returned that serves this purpose for the APM. The object is available and later passed to the event handler during completion. Many components that implement the pattern choose not to support this, in which case `FooAsync` would throw an exception if multiple invocations were detected. The modality of only permitting one outstanding request at a time can be frustrating for developers, so supporting multiple is recommended. That said, it sometimes doesn't make sense for one particular component instance to be in use concurrently, particularly for coarse-grained GUI components.

The completion of the asynchronous operation is done using an event. Unlike the APM, there is only one, simple completion mechanism. The naming convention for completion events is to add a `Completed` suffix to the operation's name. For example:

```
event EventHandler<FooCompletedEventArgs> FooCompleted;
```

It is also expected that the class on which `Foo` lives would implement the `System.ComponentModel.IComponent` interface, allowing it to be drag-and-dropped in the Visual Studio designer onto a designer surface. At that point, it becomes fairly simple to code against this asynchronous pattern. An instance is dragged on the GUI, an event handler is added for `FooCompleted` in the standard way that event handlers in GUIs are usually defined, and somewhere in the program the `FooAsync` method is invoked.

Developers familiar with the GUI style event handling paradigm will find this to be a simpler way of doing asynchronous work.

The `FooCompletedEventArgs` type contains the return value from the asynchronous operation in addition to any `out` and `ref` parameters in the original synchronous method. If the return type of the synchronous method is `void`, you can just use the existing `System.ComponentModel.AsyncCompletedEventHandler` event type, and the associated `AsyncCompletedEventArgs` class:

```
public class AsyncCompletedEventArgs : EventArgs
{
    public AsyncCompletedEventArgs(
        Exception error,
        bool cancelled,
        object userState
    );

    public bool Cancelled { get; }
    public Exception Error { get; }
    public object UserState { get; }

    protected void RaiseExceptionIfNecessary();
}
```

The `FooCompletedEventArgs` type would look like the following.

```
class FooCompletedEventArgs : AsyncCompletedEventArgs
{
    public FooCompletedEventArgs(
        T value,
        Exception error,
        bool cancelled,
        object userState
    );
    public T Result { get; }
}
```

The definition of `Result` should call `base.RaiseExceptionIfNecessary`. This ensures that the `Exception` held in the `Error` property is rethrown inside a `TargetInvocationException` (if non-null) or that an `InvalidOperationException` is thrown if `Cancelled` is `true`. The code inside of a callback using such an API should always check the state of the completion arguments before attempting to directly use the result.

For example, imagine that the `FooAsync` method was available on some class `MyComponent`. We can hook it up to some Windows Forms GUI in the following way.

```
public class MyForm : Form
{
    protected MyComponent m_myC = new MyComponent();

    void Initialize()
    {
        m_myC.FooCompleted += MyForm_FooCompleted;
    }

    void SomeButton_Click()
    {
        m_myC.FooAsync(/*... some parameter (optionally) ...*/);
    }

    void MyForm_FooCompleted(object sender, FooCompletedEventArgs e)
    {
        if (e.Error != null)
        {
            // ... paint an error on the screen ...
        }
        else
        {
            T result = e.Result;
            // ... paint the result on the screen ...
        }
    }
}
```

Something that is inherent to this example that may not be obvious is that the invocation of `MyForm_FooCompleted` will occur on the GUI thread (provided that `FooAsync` was initiated from the GUI thread). This ensures that the completion handler can properly update GUI forms with the results of the computation. Implementing this behavior properly (if you are an implementer rather than a user of the pattern) requires you to learn about GUI threading, `SynchronizationContexts`, the `AsyncOperationManager`, and the like. We'll explore those topics in much more detail in Chapter 16, Graphical User Interfaces. You may want to skip ahead to that now if you're particularly interested in learning more.

Supporting Cancellation

Another nice aspect of the event-based pattern is that it offers built in cancellation support. This is not true of the APM. For a pattern targeting GUIs, this is often a requirement. It allows a user to stop some background computation or network operation from continuing to consume machine resources when its results are no longer desired. The specific way cancellation is implemented will be discussed in other chapters: Chapter 13, Data and Task Parallelism, for cancellation of computations, and Chapter 15, Input and Output, for canceling I/O operations.

Supporting cancellation entails adding a `CancelAsync` method. Sometimes, you'll find a method that instead names the method `FooAsyncCancel` to differentiate cancellation associated with a particular asynchronous API on the component. The set of parameters this method should support depends on whether you support multiple outstanding asynchronous operations running at once. For components that only support one, there are no parameters.

```
void CancelAsync();
```

And for components that support multiple, the user state object will be used to specify which particular operation is to be canceled. This requires some way of tracking all active asynchronous operations that are currently running, for example by using an internal lookup table.

```
void CancelAsync(object userState);
```

When the `CancelAsync` method returns, there is no guarantee that the operation will have been canceled. When the event handler eventually fires, the `Cancelled` property on the event arguments will return `true` to indicate that the operation was in fact canceled. It is the responsibility of the implementation to ensure that this property is set correctly.

Supporting Progress Reporting and Incremental Results

Because this pattern is typically consumed from within GUI applications, supporting progress and incremental result reporting is often beneficial. This allows an application developer to update his or her GUI to reflect the

progress that's occurring in the background. When doing some lengthy operation such as downloading a file over the network, this feature is an important one to facilitate a good user experience.

The basic model for progress reporting entails adding another event.

```
event ProgressChangedEventHandler ProgressChanged;
```

The `System.ComponentModel.ProgressChangedEventArgs` represents the intermediary progress information with an instance of the `ProgressChangedEventArgs` class. This provides a `ProgressPercentage` property as an `int`, which represents the progress as a percentage point from `0` to `100`, and also a `UserState` property to track the optional state argument passed to the asynchronous method itself. If there are multiple asynchronous methods, you can instead name the handler `FooProgressChanged`, where `Foo` is the base name of the asynchronous method, that is, `FooAsync`.

Sometimes incremental results can be made available while progress is reported. As an example, when downloading a file over the Web, we might want to allow incremental rendering, such as what Web browsers do. To do this, `ProgressChangedEventArgs` is subclassed to contain relevant API specific state, much like subclassing `AsyncCompletedEventArgs`. When this is done, it's almost always useful to have separate progress change event handlers per each unique asynchronous operation because they are apt to offer different incremental state.

Where the EAP Is Used in the .NET Framework

The event-based pattern, much like the APM, can also be found implemented in various places throughout the .NET Framework. Here is a list of some examples.

- `System.ComponentModel.BackgroundWorker` implements the pattern in a reusable way, making it easier to write responsive GUIs. This includes cancellation support. We'll review this type in detail in Chapter 16, Graphical User Interfaces.

- The `System.Net.WebClient` component provides a plethora of asynchronous operations, in addition to cancellation support. This internally uses the APM support provided by the network classes and includes the ability to download and upload data asynchronously with `DownloadDataAsync`, `DownloadFileAsync`, `DownloadStringAsync`, `OpenReadAsync`, `OpenWriteAsync`, `UploadDataAsync`, `UploadFileAsync`, `UploadStringAsync`, and `UploadValuesAsync`.
- The `System.Media.SoundPlayer` component in the `System.dll` assembly allows you to load sound files asynchronously with its `LoadAsync` method. It also allows playing the loaded files with `PlayAsync`. Both exist so as not to interfere with the GUI thread while doing I/O.
- The `System.Windows.Documents.DocumentPaginator` component allows you to paginate XPS documents, which may entail loading data off disk and performing compute intensive work to compute pagination boundaries. It supports `ComputePageCountAsync` and `GetPageAsync` methods, and also fully supports cancellation with a `CancelAsync` method. Similarly, the serialization of XPS documents also supports asynchronous operations.

Where Are We?

We've now taken a look at the two most prevalent asynchronous programming model patterns in the .NET Framework: the APM and event-based pattern. We've seen how programs can be written to take advantage of them, most notably how to orchestrate work to be performed when asynchronous operations finish.

You'll notice that most components that implement the event-based pattern are meant to be used more with client GUI applications, while those that implement the APM tend to target lower level frameworks and server-side applications. This is consistent with the advice at the opening of this chapter with respect to how to choose one over the other if you are writing a reusable library of your own.

Next, we will wrap up our discussion of Windows concurrency mechanisms by looking at another way to schedule work: fibers.

FURTHER READING

- K. Cwalina, B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Addison-Wesley, 2006).
- J. Duffy. Implementing a High-perf IAsyncResult: Lock free Lazy Allocation. Weblog article, <http://www.bluebytesoftware.com/blog/2006/05/31/ImplementingAHighperfIAsyncResultLockfreeLazyAllocation.aspx> (2006).
- Microsoft. .NET Framework Developer's Guide: Multithreaded Programming with the Event-based Asynchronous Pattern. *MSDN whitepaper*, <http://msdn.microsoft.com/en-us/library/hkasytyf.aspx>.
- J. Prosise. 2005. Wicked Code: Asynchronous Pages in ASP.NET 2.0. *MSDN Magazine* (2005).
- J. Richter. Implementing the CLR Asynchronous Programming Model. *MSDN Magazine* (2007).

9

Fibers

A FIBER IS a lot like a thread in that it represents some in-progress work inside a process. The difference is that a fiber enjoys lightweight, cooperative scheduling and builds directly on top of the existing Windows support for preemptive scheduling. Due to their lightweight nature, careful use of fibers can sometimes yield more efficient scheduling, particularly for large amounts of work that frequently blocks. And because fibers are scheduled cooperatively, user-mode code is given more control over scheduling decisions.

Fibers are particularly interesting for the future because they are the only mechanism on Windows to allow cooperative scheduling of large amounts of work. The thread pools come close, but still rely heavily on pre-emption. Cooperative, lightweight scheduling is generally something that a massively parallel ecosystem full of software that can block will need. It's unclear whether fibers will be part of that future, but even if they aren't, they make for an interesting case study.

Before going further, I will note that fibers are not currently accessible to managed code developers. Bringing fiber support to managed code was attempted during the development of the CLR 2.0, but this support was removed just prior to shipping the final release. It is still unclear whether a future CLR will support fibers, but as of the .NET Framework 3.5 the answer is still no. Thus, this chapter will only be of interest if you're writing native code, are interested in the breadth of what Windows offers, and/or

want to keep an eye on the future. You should not feel bad about skipping to the next chapter if you’re more interested in what is necessary for concurrent programming on Windows today.

An Overview of Fibers

Each fiber executes in the context of a single OS thread at any given time, and similarly any OS thread may actively run only one fiber at a time. Any given thread can run many different fibers during its lifetime. Moreover, while a fiber can only execute on a single thread at any point in time, it may migrate between many threads during its lifetime.

In fact, fibers don’t “execute” per se: a thread assumes the identity of a particular fiber for a period of time and executes its code just as a thread always executes code. This architecture allows you to have many more fibers in the system than threads, resulting in far less resource overhead and pressure on the preemptive thread scheduler than if you simply created the equivalent number of threads.

The kernel doesn’t make any decisions about assigning fibers to threads or changing the fiber that is actively executing on a particular thread. This task is left to user-mode code. In fact, the kernel knows absolutely nothing about fibers; they are implemented entirely in user-mode Win32. The implication of this is that the code that runs on a fiber is responsible for deciding when to voluntarily relinquish its execution privilege so that another fiber can run. Typically, the component that makes this decision is referred to as a **user-mode scheduler** (UMS). The term “scheduler” is used loosely. This component can range in complexity from a 10-line function that finds a fiber’s handle from some known location and calls the appropriate fiber APIs to a full blown multithousand-line subsystem. In other words, this scheduler doesn’t necessarily require many of the traditional things that thread schedulers must implement—priority, fairness and so on—though it can.

Much like a thread, each fiber owns a set of execution state so that it can run on the hardware: a user-mode stack; a context (which includes processor register state saved at the time a fiber gets switched out); an exception chain; and, in Windows Server 2003, Vista, and subsequent OSs, **fiber-local storage (FLS)**, which provides a similar capability to thread local storage (TLS). All of

this state is copied to and from the physical thread's equivalent locations when fibers are switched, again enabling the kernel to "execute" fibers without knowing anything about fibers whatsoever. Fibers provide much of the same state that threads have, but not all of it; moreover, because the Windows kernel doesn't need to know anything about them, they are far less expensive. There are no kernel transitions required to schedule a fiber for execution, access internal fiber state, and so forth. If blocking occurs with regularity, using fibers can make a positive impact on performance by eliminating these transitions.

While all of this sounds nice—better performance and more control over scheduling—there are many practical reasons why fibers aren't always the appropriate answer. In fact, the number of legitimate uses is quite small. Before moving on to the details of how to use fibers, let's review some of these pros and cons at a high level. The danger with these mechanisms is that they can easily be used inappropriately if not properly understood.

Upsides and Downsides

There are a few reasons fibers are attractive. These were already touched on above.

The Ups

Using fibers can reduce the cost of context switches. This often leads to better throughput, particularly as the amount of runnable work exceeds the number of processors and if this work blocks frequently. In fact, this is a major reason fibers were added to Windows NT 3.51: highly scalable server programs were looking for ways to cut down on context switching overhead. Given that a thread context switch for Windows running on Intel and AMD microprocessors cost *thousands* of cycles, the ability to remain in user-mode and switch to an alternative fiber in *hundreds* of cycles is great.

Because the author of the UMS also controls the cooperative scheduling algorithms, the code paths and complexity of those algorithms are also under the custom component's control. You might be able to write a more efficient locking scheme than the general purpose one that Windows uses (which, prior to Windows Vista, serializes scheduling across the entire machine), including possibly eschewing locks altogether. You can

omit possibly taxing features such as priorities and so on. And, as already noted, there are no kernel transitions required to switch from one fiber to another. Kernel transitions add thousands of cycles to the cost of an ordinary switch.

You can of course also implement heavily customized scheduling algorithms, specialized to your particular application domain and functional needs. For example, say you have a pool of threads equal in number to the count of machine processors with each thread affinitized to a different processor and each of these threads is responsible for keeping its respective processor running by switching between fibers as they block. You might decide to assign work to these threads in a round-robin fashion to per processor work queues, allowing each thread to run independently and avoiding lock contention entirely versus the traditional central work queue approach. Because this could lead to imbalanced backlogs of work, it's not a good design for most general cases. But if you know the rate of incoming work is always high, as might be the case in a database server, this design might be worth considering. The decision is completely in your hands with a fiber based UMS.

At the same time this control also means many of the complexities (and responsibilities) of scheduling are also in your hands. This point should conjure up terms like priorities, starvation, preferred processors, processor affinity, and so on. Don't underestimate the time and effort the Windows team has spent evolving their preemptive thread scheduler over the past 15 plus years, making constant improvements to the algorithms so that it works better for a broad range of workloads. It's very unlikely you will do a "better" job at a general purpose scheduler. It is possible, however, that you might be successful at building one that better solves your very specific problems.

Finally, fibers give you access to many otherwise inaccessible low-level features, or at least features you'd have to implement yourself or rely on undocumented APIs (in ntdll) to exploit, such as the ability to create a new user-mode stack, swap a thread's stack with a new one, switch around contexts, and more. While you could build a fiber-like system without Win32 fibers, it would be difficult. Having this capability implemented for you in Win32 extends beyond just cooperative UMS scenarios and has been used

in the past to implement more exotic scheduling mechanisms such as fancy enumerators and coroutines (see Further Reading, Chen, Shankar).

The often cited example of a commercial program that has been successful at using fibers is Microsoft's SQL Server relational database software. SQL Server offers a "lightweight pooling" mode in which fibers are used for scheduling. As these fibers must block, SQL Server will switch between fibers in an attempt to keep the server as close to 100 percent CPU utilization as possible. SQL Server is uniquely equipped to use fibers because it carefully controls all blocking and resource usage, ensuring they cooperate with the scheduler. SQL Server is somewhat like a miniature OS in this regard because it is a closed and carefully engineered system. To be fair, SQL Server isn't the only program that has used fibers broadly, but it is one of the few widely known systems that has used fibers successfully. Most Windows programs simply aren't architected like this.

The Downs

As already noted, fibers cannot currently be used from managed code. This will probably alarm many readers. More details on why this is true can be found later, but the reality is that the CLR supports neither running managed code on a thread that has been used to run fibers nor converting an existing managed thread into a fiber. If you attempt such things through P/Invoking to the Win32 APIs we will review later, you're likely to create a messy situation. Thus, you should only consider using fibers if you're living in a completely native world or have a clean separation between native and managed code in your process. Even in this mixed-mode case, your use of fibers must be done with extreme care. You must absolutely guarantee that fiberized threads never wander into managed code during execution and that managed threads never call out to native components that attempt to fiberize the thread and/or schedule additional fibers.

Many important pieces of information that are fully available to the kernel-mode thread scheduler are inaccessible in user-mode, making it hard to build the kind of scheduler you might need. One very important example is blocking. Normally, you'd want to switch to another fiber when the running fiber blocks. But the OS doesn't have any way to discover when a thread blocks and to prevent it from doing so. To achieve this goal, you have

to ensure all blocking calls that may occur on fiberized threads are routed through some central user-mode function under your control. Later, we'll look at a very simple UMS that offers such a function that fibers must call instead of blocking. And even with that, I/O must be treated differently, by somehow morphing synchronous I/O calls into asynchronous ones.

Worse than not doing any of this for you, Windows will get in your way. Many Win32 APIs and low-level kernel routines can block due to things like contended lock acquisitions (in user- or kernel-mode), hard virtual memory page faults, and so on. And when such things occur, the thread on which your fiber is running will block and your scheduler won't be given a chance to schedule a new fiber to run in its place. If you're trying to keep the number of running threads identical to the number of processors, this can cause one of the CPUs to drop to 0 percent utilization, something often called a **stall**. For closed systems, you may be able to devise an architecture much like SQL Server's where all blocking is cooperative (by making most of Win32 off limits), including synchronization and I/O, and where page faulting isn't a problem because all memory is managed explicitly by the system such that paging never happens. SQL Server can do this, but is fairly unique in this regard. Other systems need to deal with the fact that stalls might occur perhaps by using a "watchdog" thread that monitors for stalled threads and introduces additional threads to service work.

It is also very difficult to run fibers inside an extensible system because of **thread affinity**. Thread affinity occurs when some thread-wide state is used by code on that thread; in the fiber case, this makes it impossible to correctly migrate the fiber to another thread and often makes it impossible to schedule an alternative fiber on the thread. Aside from the blocking issues mentioned above, all it takes is one of these components to use certain parts of the CRT, VC++ exception handling and/or explicit TLS, and strange thread-affinity bugs are bound to arise. The Windows ecosystem has grown up with the assumption that threads are the units of concurrency and that any and all TLS is fair game, including a lot of Win32. Fibers defy these historical assumptions. Worse, the use of dangerous code is not something that can be detected by a UMS.

Finally, fibers do not have good tool support as threads do from Microsoft's debuggers, including Windbg and Visual Studio (see Further

Reading, Stall). If you decide to adopt fibers in your program, you will also have to bring a lot of knowledge about internal data structures, how to access them, and how to interpret the layout of these structures.

In Conclusion . . .

Many of these drawbacks are serious. If you've gotten the impression that fibers are not appropriate for extensible systems (most systems), then you have been given the intended impression. Despite all these words of warning, fibers do have their place—for highly scalable and closed systems that either carefully control extensibility points or don't have any. With care, they can also be used to implement scalable dynamic work schedulers and useful abstractions such as coroutines and agents-like simulations.

Using Fibers

Now that we've reviewed the highlights and lowlights of using fibers, let's review the mechanisms for using them. Everything shown will be in C++ and Win32. We'll return to some additional design topics later, in addition to looking at an implementation of a very simple fiber based cooperative UMS.

Creating New Fibers

A fiber is created much like a thread, with the Kernel32 function `CreateFiber` or, as of Windows XP or 2000 SP4 (and Windows Server 2003 and Vista), `CreateFiberEx`.

```
LPVOID WINAPI CreateFiber(
    SIZE_T dwStackSize,
    LPFIBER_START_ROUTINE lpStartAddress,
    LPVOID lpParameter
);
LPVOID WINAPI CreateFiberEx(
    SIZE_T dwStackCommitSize,
    SIZE_T dwStackReserveSize,
    DWORD dwFlags,
    LPFIBER_START_ROUTINE lpStartAddress,
    LPVOID lpParameter
);
```

You'll notice that `CreateFiber` looks a lot like `CreateThread`, so most of the arguments to this API are probably obvious. Note that because fibers were added in a Windows NT 3.5 service pack, you must define the `_WIN32_WINNT` symbol to be `0x0400` or higher before including `Windows.h` to access any of the functions we'll review in this chapter.

`lpStartAddress` refers to the function at which the fiber will begin execution.

```
VOID CALLBACK FiberProc(PVOID lpParameter);
```

Unlike thread start routines that return a `DWORD` exit code, a fiber's start routine doesn't return anything. That's because a fiber doesn't have an exit code as a thread does. The `lpParameter` argument to `CreateFiber` and `CreateFiberEx` is passed to the start routine as its `lpParameter` argument. Its purpose is the same as with `CreateThread`: it enables the creating thread to pass arbitrary data to the callback.

During fiber creation, a new user-mode stack will be allocated. The `dwStackSize` parameter to `CreateFiber` is interpreted the same way as `CreateThread`'s `dwStackSize` parameter: that is, `0` for the default stack size, taken from the current executable, and the commit (rather than reservation) size otherwise. There is no way to specify an alternative reserve size with `CreateFiber`. Instead, you must use the `CreateFiberEx` API, which allows you to specify reservation and commit sizes as independent arguments: `dwStackCommitSize` specifies how many bytes to commit and `dwStackReserveSize` specifies the number of bytes to reserve. Either of these arguments can be `0`, which indicates that the default value for that particular value should be taken from the process. If both are specified, the reserve size must equal or exceed the commit size.

(Please refer to the section on thread stacks in Chapter 4, Advanced Threads, for a detailed description of the differences between reserved and committed virtual memory, the layout of stacks, and so on. User-mode stacks for fibers are treated the same as with threads: the fiber implementation allocates, manages, and swaps the target thread's stack with the new fiber's without requiring kernel support by using a combination of documented and undocumented APIs.)

The only legal value that can be passed for `dwFlags`, aside from 0, is `FIBER_FLAG_FLOAT_SWITCH`. If this is specified, floating point registers are captured and restored when the fiber's `CONTEXT` is taken from or restored to a particular thread. If the flag is not specified, these registers are left as is and therefore multistep floating point operations that span a fiber switch may cause or observe data corruption. If you remember, in Chapter 4, Advanced Threads, we discussed `GetContext`, which means the `CONTEXT_FLOATING_POINT` flag will or will not be passed by the fiber switching library on X86 and X64 systems based on the presence or absence of `FIBER_FLAG_FLOAT_SWITCH`, respectively.

Conveniently, in addition to `lpParameter` supplied to the fiber creation routines being passed to the `FiberProc`, it is also stored ambiently in a global per fiber location so you can retrieve it subsequently with the `GetFiberData` macro:

```
PVOID GetFiberData();
```

Notice that the return value for both `CreateFiber` and `CreateFiberEx` is a `LPVOID`; this is in contrast to a `HANDLE`, as is returned by `CreateThread`. Recall that fibers are implemented entirely in user-mode, meaning that the Windows kernel doesn't know anything about them. A fiber therefore has no associated kernel object (like threads do) and, thus, has no true handle in the capital `HANDLE` sense. But, among other things, you will need the returned value to run the fiber on a thread, so the opaque pointer returned is something of a user-mode handle. The main difference is that the `LPVOID` value is not reference counted at all as `HANDLEs` generally are, so once the fiber has been deleted any subsequent uses of the `LPVOID` will cause problems.

When you create a fiber, it doesn't begin executing until it's been scheduled onto an already executing thread (often, but not always, the one calling `CreateFiber` itself). Fibers don't "run"; they are mapped to threads that run. For a fiber to execute, it must be "switched to" by a running OS thread with a call to the `SwitchToFiber` Win32 API (which will be examined soon). The fiber remains running on that thread as long as the thread remains running, as decided by the Windows preemptive scheduler. When that thread is switched out, the fiber goes with it; the next time the thread runs, that fiber also runs.

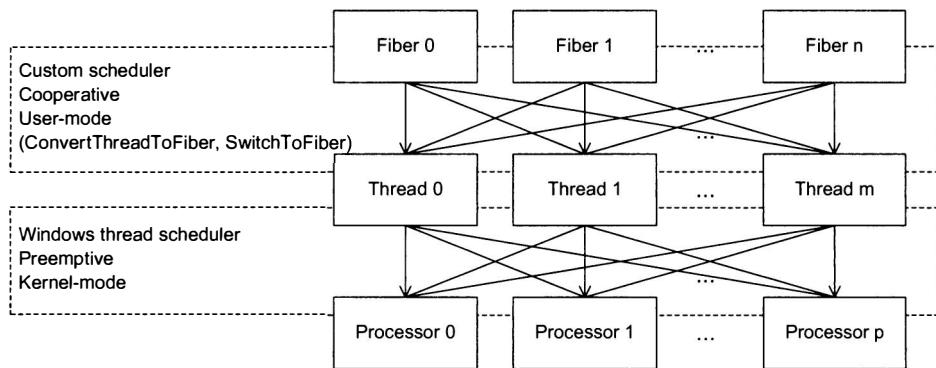


FIGURE 9.1: Relationship between fibers, threads, and processors

The requirement that a fiber be explicitly switched onto a thread is the cooperative aspect to fiber scheduling. Notice that scheduling isn't 100 percent cooperative with fibers because we still rely on Windows' ordinary preemptive scheduling process for a fiber to physically execute. The relationship between fibers, threads, and processors is depicted in Figure 9.1.

Converting a Thread into a Fiber

At this point, we've seen how to create new fibers. However, before you can run one of these new fibers on a thread, you must first **fiberize** the target thread. This just means that the thread is prepared by the fiber implementation so that it is capable of running fibers, in addition to converting the thread itself into a fiber so that it can be subsequently swapped in and out with the fiber switching APIs.

This step is done with `ConvertThreadToFiber` or `ConvertThreadToFiberEx`,

```

LPVOID WINAPI ConvertThreadToFiber(LPVOID lpParameter);
LPVOID WINAPI ConvertThreadToFiberEx(LPVOID lpParameter, DWORD dwFlags);

```

Calling either one allocates a new fiber data structure, such as `CreateFiber`, though it uses the current thread's user-mode stack rather than creating a new one (hence the simpler parameter list). And it doesn't take a fiber-start routine argument because the calling thread is already running when the call is made. Both functions return the fiber's address as a `LPVOID` (the fiber's "handle") and take an `lpParameter` argument that is

subsequently accessible via `GetFiberData`, just as with the `lpParameter` argument to `CreateFiber` and `CreateFiberEx`.

This function prepares the necessary internal data structures in the TEB that will be subsequently used to track and execute fibers. There's a more fundamental reason for calling this though. Without doing so, there would be no way to recover the original thread context that existed before switching to another fiber. After this is called on a thread, the current thread's newly created fiber is actively running, and once it has been switched out, the original thread's context can later be restored by running the associated fiber again. You can even restore the newly converted fiber to a separate thread, though you clearly have to be careful about any thread affinity that may have already existed before getting to this point.

As with `CreateFiberEx`, you can specify the `FIBER_FLAG_FLOAT_SWITCH` in the `dwFlags` argument, and this has the same exact meaning as was described earlier for `CreateFiberEx`, that is, floating point registers are captured and restored when switching.

If the return value is `NULL`, it means converting the thread to a fiber failed. If `GetLastError` subsequently returns `ERROR_ALREADY_FIBER`, it means that the thread is already a fiber and doesn't need to be converted a second time. It is safe to proceed when this error is returned, and you'll have to use `GetCurrentFiber` to access the currently executing fiber's handle. In older versions of Windows, trying to convert a thread to a fiber multiple times would result in unpredictable behavior (see Further Reading, Chen).

Determining Whether a Thread Is a Fiber

Before Windows Vista there was no way, other than the `ERROR_ALREADY_FIBER` error, to determine whether a thread had already been fiberized. The new `IsThreadAFiber` function allows you to inquire about this. If the thread has already been converted to a fiber, this function returns `TRUE`, and otherwise it returns `FALSE`.

```
BOOL WINAPI IsThreadAFiber();
```

Assuming the current thread has actually been converted to a fiber, you can also retrieve the current fiber pointer with the `GetCurrentFiber` macro.

```
PVOID GetCurrentFiber();
```

You must use `GetCurrentFiber` carefully. If the current thread isn't a fiber, instead of returning `NULL` and permitting you to check for a certain error code, this function will actually retrieve what may look like a valid pointer. (It's just a pointer taken from the TEB that may have been used for other purposes if the thread hasn't been fiberized.) If you try to use this returned pointer with any of the fiber APIs, you're likely to crash your program with an AV or cause other data corruption. Most fiber enabled programs are carefully written so you absolutely know a thread is a fiber before calling `GetCurrentFiber`. Usually threads are fiberized at a very specific point in their lifetime—rather than dynamically or lazily—but in those cases for which this isn't so, `IsThreadAFiber` can be helpful. And it's useful for diagnostics.

You may have noticed that both `GetCurrentFiber` and `GetFiberData` are macros instead of Win32 functions. These routines inline access the `FiberData` field of the TEB, much like the `NtCurrentTeb` macro from Chapter 4, Advanced Threads. The result is a very efficient lookup: on X86 it accesses the segmented register `FS:0x10`, on X64 the segmented register `GS:0x20`, and on IA64 accesses the `FiberData` field from the `_NT_TIB` whose pointer is found in the `IntR13` register. Note that the current fiber pointer points to the `PVOID` fiber data, so `*((PVOID *)GetCurrentFiber())` is the same value as `GetFiberData()`, although this is an implementation detail that shouldn't be relied on.

Switching Between Fibers

We've seen how to create a new fiber and convert the current thread into a fiber (which continues to run after conversion), but we have yet to focus on how to schedule a new fiber onto the current thread. The `SwitchToFiber` function performs this: it takes a fiber's `LPVOID` "handle" as its sole argument, and switches to it. You must only call this on a fiberized thread.

```
VOID WINAPI SwitchToFiber(LPVOID lpFiber);
```

This function captures the current fiber's data—which is taken from the currently executing thread)—including the thread's `CONTEXT`, stack base and limit, and the current thread's exception chain, so that the current fiber can be rescheduled for execution again later. It then fixes the current thread to hold the new incoming fiber's previously saved information, concluding

by restoring the incoming fiber's CONTEXT back to the processor's registers. The result is that the call to `SwitchToFiber` returns on a separate stack from the one on which it was called: the processor jumps to the newly scheduled fiber's saved EIP (which got pushed onto its own stack during its last call to `SwitchToFiber`) and the fiber is now running on the calling thread. It's extraordinary if you stop to think about it.

A call to `SwitchToFiber` cannot fail: it doesn't allocate memory and doesn't perform any validation that the address passed refers to a valid fiber. This lack of validation speeds things up, but can cause problems. If the LPVOID is invalid, you may see a crash and / or memory corruption.

There is also another subtle implication due to the lack of validation. You need to ensure you don't accidentally try to switch to an already running fiber. The results can be amusing if you accidentally run the same fiber on many threads at once. These multiple threads will run code using the same user-mode stack. The resulting behavior is very unpredictable.

If a fiber unwinds its stack entirely, the thread running that fiber will exit and the fiber is automatically deleted. This also means that an unhandled exception from a fiber will tear down the thread running that fiber. Unless you have special code at the top of each fiber's stack, both of these points of thread exit make it difficult to maintain control over the work running in all of the fibers in the system, and it is another reason fibers are hard to use in an extensible system. If you have a thread with a top-level exception handler and switch to a fiber without a top-level handler, a failure on that fiber can completely destroy your error handling logic. One of the more successful uses of fibers is to implement work scheduling via thread pools, in which case you can easily handle both situations because you typically own the code on the top of each fiber's stack.

Deleting Fibers

Once a fiber has completed execution, it should be deleted with `DeleteFiber`, which frees its associated resources, including its user-mode stack.

```
VOID WINAPI DeleteFiber(LPVOID lpFiber);
```

After this call, the LPVOID is garbage and mustn't be used anymore. Any pointers to memory on that fiber's stack are now invalid. If the target

fiber is the one actively running on the calling thread, `ExitThread` is automatically invoked on the current thread by `DeleteFiber`. Trying to delete a fiber that is already running on a separate thread will yield unpredictable (and undesirable) behavior. Proper usage typically entails some form of synchronization in order to achieve clean shutdown of all fibers inside a system.

If a thread no longer needs to run any fibers, but must continue running normal code, then you can call the `ConvertFiberToThread` routine.

```
BOOL WINAPI ConvertFiberToThread();
```

This releases any resources that were allocated by `ConvertThreadToFiber` and also deletes the fiber currently running on the thread without deallocating its stack. Once this function has been called, the thread may no longer run any fibers unless it calls `ConvertThreadToFiber` again.

That's about it, from a mechanisms' standpoint. The fiber support in Win32 is composed of a handful of APIs. Fibers are deceptively simple, assuming you can get your head around the switching aspect. Let's look at a quick sample and move on to some more practical usage topics.

An Example of Switching the Current Thread

Here's a small program that illustrates fibers in action. This also shows some of the power (and amazing properties) that fibers offer. We will do several things: (1) fiberize the current thread, `t0`, in our `main` routine to create `f0`; (2) create a second fiber that we'll call `f1`; (3) spawn a new thread, `t1`; (4) switch to `f1` on `t0`; and (5) switch to `f0` on `t1`. Lastly, `t1` will finish running the `main` function, which, you'll recall, started executing on `t0` back in step 1. We've effectively moved work from one thread to another through the use of fibers.

```
#include <stdio.h>
#define _WIN32_WINNT 0x0400
#include <windows.h>

PVOID g_pFiber0;
HANDLE g_pSwappedOutEvent;

DWORD CALLBACK RunOtherFiber(PVOID lpParameter)
{
```

```
// (We leak the converted fiber -- OK for this sample.)
ConvertThreadToFiber(NULL);

// S2
printf("%d: 'RunOtherFiber': wait for swap notification\r\n",
       GetCurrentThreadId());
WaitForSingleObject(g_pSwappedOutEvent, INFINITE);

printf("%d: 'RunOtherFiber': resuming main...\r\n",
       GetCurrentThreadId());

// S5
SwitchToFiber(g_pFiber0);
return 0;
}

VOID CALLBACK FiberMain(PVOID lpParameter)
{
    // S4
    printf("%d: running 'FiberMain': notify and wait for ack\r\n",
           GetCurrentThreadId());

    SetEvent(g_pSwappedOutEvent);

    printf("%d: 'FiberMain': done\r\n", GetCurrentThreadId());
}

int main(int argc, wchar_t * argv[])
{
    // S0
    printf("%d: 'main': starting main\r\n", GetCurrentThreadId());

    g_pFiber0 = ConvertThreadToFiber(NULL);
    g_pSwappedOutEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    // S1: Create a thread to run the current stack.
    HANDLE hThread = CreateThread(
        NULL, 0, &RunOtherFiber, NULL, 0, NULL);

    // S3: Now create a new fiber to run on this thread.
    PVOID pFiber1 = CreateFiber(0, &FiberMain, NULL);
    SwitchToFiber(pFiber1);

    // S6
    printf("%d: 'main': ending main\r\n", GetCurrentThreadId());
    CloseHandle(hThread);

    return 0;
}
```

Let's walk through the sequence of events that occur when you run this code. I've numbered the particularly interesting regions of code with a statement numbering scheme (S0, S1, and so on) to make it easier to refer back to the sample.

- S0. The `main` function begins on `t0` (`t0` is a symbol here; the thread ID returned by `GetCurrentThreadId` and printed to standard output depends on the whims of the OS thread ID numbering scheme). We then immediately convert `t0` to a fiber, storing its fiber handle in the global `g_pFiber0` variable. At this point, the thread is running `g_pFiber0(f0)`.
- S1. We create a new thread, which we'll call `t1`, from our `main` function whose thread start routine is the `RunOtherFiber` function.
- S2. Inside of `RunOtherFiber`, on `t1`, we wait for an event `g_pSwappedOutEvent` that will be set once `t0` has switched to a separate fiber. We need to wait for this to happen before `t1` can run `g_pFiber0` because until the event is set, `t0` is still actively running its original fiber, meaning we can't touch it from `t1`.
- S3. Meanwhile, `t0` continues, creating a new fiber `pFiber1` whose fiber start routine is `FiberMain`. It then switches to it. At this point no thread is running `g_pFiber0`: that is, its stack is not active on any thread.
- S4. The `FiberMain` function, being run on thread 0 as part of executing `pFiber1(f1)`, sets the `g_pSwappedOutEvent` on which `t1` is waiting, prints some information to standard output, and returns. The thread may or may not exit the system entirely before `t1` notices that the event has been set.
- S5. After we're sure `t0` is definitely not using `g_pFiber0`, `t1` switches to it via `SwitchToFiber`. (Note that we didn't save the `LPVOID` returned when `t1` called `ConvertThreadToFiber`; normally this would be bad because we would no longer be able to recover it: the resources associated with it, including its stack, would be completely leaked. But in this simple example, we can ignore this minor point, just like we're ignoring the fact that this example doesn't check for error conditions at all.)
- S6. Once `t1` has switched to `g_pFiber0`, control on `t1` transfers back to the `main` routine where `t0` had left off with its own previous call to

`SwitchToFiber` (when it switched to `pFiber1`). What happened was that `t0` made the call to `SwitchToFiber` inside `main`, while `t1` later returned from this same function call. This thread now prints information to standard output—you’ll notice the thread ID printed here is different than the one printed in `S0`—and then returns. Once both `t0` and `t1` have exited, the program will exit.

This example is of very little practical value. But if you follow the sequence of events, studying this example should help to solidify your mental model and understanding of how fibers work. Extending this something more useful (such as a coroutine-like system) is not difficult.

Additional Fiber-Related Topics

Here we review some additional topics that aren’t fundamental to using fibers, but can be useful, either because they provide additional functionality or can help deepen your understanding of how fibers integrate with real-world systems. After this, we’ll move on to building an experimental UMS.

Fiber Local Storage (FLS)

Just as you can store arbitrary information local to a thread using TLS, you can store arbitrary information isolated within a fiber. The functions are nearly identical in capability to the `T1s` family of Win32 APIs described in Chapter 3, Threads, with some notable differences. Because FLS was added only as recently as Windows Server 2003, you must define `_WIN32_WINNT` to be `0x0502` or higher to access the function definitions from `Windows.h`.

To use FLS, you must first dynamically allocate a new FLS slot using the `FlsAlloc` function. This returns a `DWORD` which is the unique slot index that can be subsequently used by any fibers in the system to access the new FLS slot:

```
DWORD WINAPI FlsAlloc(PFLS_CALLBACK_FUNCTION lpCallback);
```

The contents of this newly allocated slot are automatically zeroed. You must check the return value from `FlsAlloc`: if it is `FLS_OUT_OF_INDEXES`, the FLS slot was not created and the return index is not an index at all, it’s an error code. `GetLastError` will return the cause of this problem. If this

happens it's typically because, like TLS, there are only a finite number of slots that can be created. In fact, the number is far fewer for FLS than it is for TLS. Whereas recent versions of Windows allow over 1,000 TLS slots in a process, there are only 128 FLS slots available in any one process.

The `lpCallback` argument leads us to an interesting difference between TLS and FLS. Normally (in a DLL) you will use the `DllMain` function to call `TlsAlloc` during the `DLL_PROCESS_ATTACH` notification. And then it's common for all subsequent `DLL_THREAD_ATTACH` notifications to also initialize some relevant TLS data in the slot generated by the initial allocation, and for `DLL_THREAD_DETACH` notifications to free this data. Unfortunately, you don't get equivalent DLL notifications like this when fibers enter and exit the system, so we need to use a different strategy for FLS initialization and cleanup. This is the purpose of the callback. If you supply an `lpCallback`, it will be invoked whenever one of three things happens: a fiber is destroyed with `DeleteFiber`, the thread that is running a fiber exits, or the FLS slot is freed. This gives you a chance to clean up whatever FLS state has been stored in the FLS slot so that memory and resources are not leaked. In all cases, the callback runs on the thread (and fiber) which initiates the specific event.

The callback isn't required, so passing `NULL` is a perfectly legitimate thing to do. Without it, however, it's difficult to ensure clean up of resources stored in FLS so it's commonly used.

`PFLS_CALLBACK_FUNCTION` refers to a function of the following signature:

```
VOID WINAPI FlsCallback(PVOID lpFlsData);
```

When invoked by the system, the `PVOID` value currently held in the respective FLS slot is passed as `lpFlsData`. The callback should then simply free the memory, resources, and so forth. Note that this callback does not execute if the `PVOID` in an FLS slot holds the value of `NULL`.

A FLS slot can be later freed using the `FlsFree` function.

```
BOOL WINAPI FlsFree(DWORD dwFlsIndex);
```

Once a slot has been allocated, fibers may freely set and retrieve any arbitrary `PVOID` value with the `FlsSetValue` and `FlsGetValue` functions:

```
BOOL WINAPI FlsSetValue(DWORD dwFlsIndex, PVOID lpFlsData);
PVOID WINAPI FlsGetValue(DWORD dwFlsIndex);
```

These do what their names imply: `F1sSetValue` stores `lpF1sData` in the `dwF1sIndex` slot for the current fiber's FLS, and `F1sGetValue` retrieves existing data from the same slot. If an invalid `dwF1sIndex` value is supplied, `F1sSetValue` returns FALSE while `F1sGetValue` returns NULL. This latter case is indistinguishable from an FLS slot containing a true NULL value (the default), though `GetLastError` will provide failure details. `F1sSetValue` can also fail because it has to lazily allocate storage for the slot.

Thread Affinity

When a fiber runs, it has access to all thread local state. This is both good and bad. It can be convenient, because you can use many of thread based services in a fiber based system. And storing data on the physical thread ensures that it flows with the logical continuation of work, no matter what APIs are called or how interwoven the stack becomes, and is, therefore, "always" accessible. This avoids having to figure out how to pass data in arguments to flow information during execution.

But this practice can also lead to some serious problems in a fiber based system. The general problem here is referred to as thread affinity. This term is meant to cover any situation in which a component depends strongly on the identity of a thread remaining consistent across multiple operations for correctness. In fact, thread affinity poses problems for the future of parallelism on the Windows platform because software that engages in this practice is tightly coupled to threads as the execution mechanism. Even if fibers aren't the way of the future, decoupling logical work from the physical thread is probably a key component of the future. But, setting the future aside, thread affinity impacts any usage of fibers today.

Many services on Windows have traditionally associated state with the executing thread to keep track of certain ambient contextual information. The examples are many. Error codes are stored in the TEB (accessible via `GetLastError`), as are impersonation tokens and locale IDs. Arbitrary program and library state can also be—and routinely is—stashed away into TLS for retrieval later on. COM introduces an even worse form of affinity with its "threading" apartment model,

particularly Single Threaded Apartments (STAs), in which components created on an STA are only ever accessed from the single STA thread in that apartment. And let's not forget all of the Windows GUI frameworks, which are built assuming only the GUI thread will run the message loop (as we explore further in Chapter 16, Graphical User Interfaces). Finally, since the introduction of the multithreaded C Runtime library, functions that historically relied on global variables now rely on TLS instead.

As a simple example of how this affects systems that use fibers, take Windows CRITICAL_SECTIONS. Once a call to `EnterCriticalSection` succeeds, the data structure is tagged so that the physical OS thread that made the call appears as the owner. In other words, it relies on thread affinity. Imagine we were to make a call to `EnterCriticalSection`, then call in code that called `SwitchToFiber`, and, only after that, make a call to `LeaveCriticalSection`. That is:

```
CRITICAL_SECTION cs;

void f()
{
    EnterCriticalSection(&cs);
    g();
    LeaveCriticalSection(&cs);
}

void g() {
    ...
    SwitchToFiber(...);
}
```

There are two major things that might go wrong.

1. The new fiber itself may try to call `EnterCriticalSection` on the same section. What would you expect to happen in this case?

Because critical sections are reentrant and because lock ownership is based on the OS thread ID, this is just like a recursive lock acquire to Windows. And so it permits the new fiber to acquire the same critical section recursively even though the work that will be done under the lock is presumably logically distinct. This fiber will then proceed to

execute under the protection of the lock, possibly seeing partial state updates in progress by the old fiber and probably corrupting data or crashing the process. If we were using a nonreentrant lock instead, such as a `SRWLock`, the same scenario would lead to deadlock.

2. Assuming the process stays alive and we return to the original fiber, it will only be able to release the critical section it has acquired if it is later restored to the same thread on which it performed the acquisition. This is possible. But if your scheduler tries to run it elsewhere, the call to `LeaveCriticalSection` will corrupt the `CRITICAL_SECTION` data structure, leaving behind a time bomb that will undoubtedly lead to surprising behavior.

If you have complete control over all of the code inside of the critical region, you can be careful and ensure that a call to `SwitchToFiber` doesn't creep inside. Our sample UMS component later makes liberal use of `CRITICAL_SECTIONS` and is careful about this. But this is just one example out of the many cited sources of thread affinity.

Any serious fiber based system must virtualize as much of the thread local state as possible, ensuring that contextual information is carried around with the logical work on the fiber instead of the physical OS thread. Some thread local state is already virtualized by the fiber system itself. The exception chain, as an example, is automatically switched when a fiber switches, ensuring that Windows SEH still works correctly if fiber switching occurs nested inside a try block. But there's plenty of state that isn't, including all of the TLS in the calling thread. The affinity problem and how to virtualize resources is explored briefly in the following case study where we look at the CLR's (now defunct) support for running in fiber-mode in more depth.

A Case Study: Fibers and the CLR

The CLR tried to add support for fibers in version 2.0, with the main goal of enabling SQL Server 2005 to continue running in its "lightweight pooling" mode (a.k.a. fiber mode) when the CLR was hosted in-process. After years of hard work, mostly due to schedule pressure and many difficult bugs at the tail of the project that affected only fiber-mode, the CLR team declared

fibers completely unsupported (see Further Reading, Viehland). Given the choice between fixing bugs that impact the majority of customers—which almost exclusively use CLR running in thread-mode—and fixing the fiber-related bugs that would impact very few, the choice wasn’t difficult. This decision impacts SQL Server customers that want to run managed code while using fiber mode, but there are fewer of them than customers who want to run in thread mode.

But this is also the key to all of the earlier warnings about managed code and fiberized threads not mixing well. You might be wondering why it matters: What does the CLR need to know about fibers anyway? We’ll briefly review below what the CLR does specially to support fibers—or at least, what it did—which should help to paint a more complete picture. It’s a fascinating case study of what kinds of problems are apt to be encountered when attempting to add fibers to an existing, real-world system.

Runtime Support Details

Perhaps the biggest thing the CLR needed to do to support fibers intrinsically in the runtime was to decouple the CLR thread object from the physical OS thread.

Because most managed code accesses thread-specific state through the facade of an internal CLR thread object, the runtime can redirect calls to threads or fibers as appropriate. The whole runtime is written to call out to CLR hosts so they can override certain task management functions, enabling a cooperative scheduling host to override policies and do its job, such as making decisions about when to switch fibers when a blocking call is made. When a CLR host with certain host management overrides is detected, the CLR also defers many tasks to it that it would ordinarily implement with straight OS calls. For example, instead of just creating a new OS thread, the CLR will call out through the `IHostTaskManager` interface so that the host can create a fiber instead if it wishes.

In addition to this, the runtime does various other things of interest.

1. Because the CLR thread object can be per fiber (by choice of the host), any information hanging off of it is also per fiber. This encompasses many bits of thread local information. For example,

`Thread.ManagedThreadId` returns a stable ID that flows around with the CLR thread and that isn't dependent on the identity of the physical OS thread. Therefore, using it creates no form of OS thread affinity and each fiber running on the same thread over time sees different IDs. Impersonation and locale information is also carried with the CLR thread instead of the OS thread, and lock information for CLR monitors uses the managed thread ID for ownership, meaning that it flows with the CLR thread too (avoiding the `CRITICAL_SECTION` problem noted earlier). All of this allows a fiber to continue moving code between threads.

2. Managed TLS is stored in FLS if a fiber is being used (and provided FLS is available). This includes the `ThreadStaticAttribute` and `Thread.GetData` and `Thread.SetData` methods. The use of these APIs, therefore, also implies no form of OS thread affinity and remains safe.
3. Since the list of CLR thread objects is always known by virtue of callouts to the host, the list of all user-mode stacks active on threads and inactive on nonrunning fibers is always known. This enables the runtime to correctly walk stacks, propagate exceptions correctly, and report all of the active roots held on all stack frames to the GC. Without close coordination with the host, any one of these would pose a serious problem for the runtime: live references on stacks whose fiber wasn't actively running could be missed; subsequent accesses would then try to use reclaimed GC memory, crashing or corrupting along the way.
4. Any time the CLR blocks for synchronization, a call is made to the host's `TaskManager` so that it may call `SwitchToFiber`. This includes calls to `WaitHandle.WaitOne`, contentious calls to `Monitor.Enter`, `Thread.Sleep`, and `Thread.Join`, as well as any other APIs that use those internally. This approach still isn't perfect. Some managed code blocks by P/Invoking, either intentionally or unintentionally, and there is a separate I/O host interface for nonsynchronization waits. The existing loopholes can be problematic and prevent a host from switching in fiber-mode. The lack of coordination with blocking in the Windows kernel also makes it way too easy to accidentally stall a CPU for lengthy periods of time.

5. The CLR will do some things during a fiber switch to shuffle data in and out of TLS to ensure that the incoming fiber and the target thread are in alignment. Remember the `SwitchToFiber` routine leaves all TLS state intact, so the CLR needs to squirrel some important data away manually. This includes copying the current thread object pointer and AppDomain index from FLS to TLS, for example, as well as doing general book-keeping that is used by the internal fiber switching routines (`SwitchIn` and `SwitchOut`).
6. CLR internal critical sections coordinate with the host and anytime the runtime creates or waits on an event it goes through a thin wrapper that calls out to the host. This meant sacrificing some freedom around waiting, such as doing away with `WaitForMultipleObjectsEx` with `WAIT_ANY` and `WAIT_ALL`, but ensures seamless integration with a fiber-mode host.
7. All thread creation, aborts, and joins are host aware and call out to the host so they can ensure these events are processed correctly, given the alternative scheduling mechanisms.

None of this logic takes effect if fibers are used underneath the CLR. It all requires close coordination between the host, which is doing user-mode scheduling, and the CLR, which is executing the code running on those fibers. If you call into managed code on a thread that was converted to a fiber and later switch fibers without involvement with the CLR, things will break badly. The CLR's stack walks and exception propagation might rely on the wrong fiber's stack, for example, and the GC would fail to find all active roots in the process because it wouldn't see the fiber stacks that weren't live on threads at the time, among many other likely problems.

Important areas of the BCL and runtime can introduce thread affinity and make a call that might block, and later release, this thread affinity—such as the acquisition and release of an OS `CRITICAL_SECTION` or `Mutex`—have been annotated with calls to `Thread.BeginThreadAffinity` and `Thread.EndThreadAffinity`. These APIs call out to the host, which maintains a recursion counter to track regions of affinity. If a blocking operation happens inside such a region (i.e., the affinity count > 0), the host must avoid rescheduling another fiber on the current thread and/or moving the

current fiber to another thread. This can cause stalls, so overusing these APIs is generally not advised, but it's sometimes unavoidable and is better than the consequence of pretending that affinity doesn't exist.

In reality, there is little code that uses these APIs faithfully. Large portions of the .NET Framework were not modified to use these calls and thus are stall prone. In fact, many of the affinity problems are inherited from Win32 and simply lie dormant. The fact that fiber-mode is no longer available makes this perfectly OK.

But were fiber-mode put back into the system, the lack of annotations would have a dramatic impact on reliability and correctness of these libraries when used in a fiber-mode host. Switching a fiber that has acquired OS thread affinity can result in data being accidentally shared between units of work (such as the ownership of a lock) or movement of work to a separate thread (which then expects to find some TLS, but is surprised when it isn't there). Both are very bad. If anybody was serious about supporting fibers underneath managed code, it would probably entail a full audit of all of the libraries to find dangerous unmarked P/Invokes and OS thread affinity.

The `ICLRTask::SwitchOut` API (see `mscoree.idl`) was actually cut from the 2.0 release of the CLR, meaning it always returns `E_NOTIMPL`, which means you physically cannot write a host that switches out a task while it is in the middle of running. This in turn makes it impossible to build and experiment with a fiber-mode host for the CLR. Re-enabling it for those playing w/Shared Source CLI (SSCLI) 2.0 should be a trivial exercise.

In the end, remember that the CLR team decided to cut fiber support because of stress bugs. Most of these stress bugs wouldn't have blocked simple, short running scenarios, but would have plagued a long running host like SQL Server that places a premium on reliability. Given that the niche for fibers tends to be these sorts of high demand, scalable server programs, cutting it was the appropriate decision to make.

Building a User-Mode Scheduler

Let's walk through the process of building a straightforward fiber based cooperative user-mode scheduler (UMS). This will help illustrate how

fibers can be used. Feel free to skip straight to the next chapter if this is not of interest. While the concepts will be intellectually interesting for many readers, they are not material to learning how to write concurrent programs on Windows.

The UMS scheduler we will build is very much like a thread pool, with the primary difference that all blocking is cooperative with the scheduler so that it can use fibers to keep the threads running without having to create more threads than processors. Note that what we're about to see is for illustration and education purposes only. You wouldn't want to go ahead and reuse the code verbatim as listed here, but my hope is that it gives you some ideas about how fibers might be used in the real world.

Here is a summary of our scheduler's structure. We will define a `FiberPool` C++ class. When instantiated, this pool will create a certain number of threads to execute work, as specified by a number passed as an argument. This number should ideally be set to the number of processors on the machine. Each thread created is responsible for running one or more fibers, and each fiber is responsible for dequeuing and executing elements out of a shared work queue. Occasionally, work running on a fiber may have to block. Such blocking must cooperate with our scheduler in order for us to do anything intelligently, which means the callback must invoke a special `Block` method on the `FiberPool`, passing the `HANDLE` we'd like to wait to become signaled as an argument. This must be done instead of, say, calling `WaitForSingleObject`, directly by the callback and therefore constraints what it can do (e.g., callbacks cannot perform message waits unless we add explicit support for them). Our pool attempts to keep all threads running at all times by switching between fibers. Only when there is no real work to perform will the pool block a thread.

Before moving on, some caveats are in order. We'll take some fairly naïve shortcuts in this pool to keep the amount of code we'll look at manageable. For instance, we will share global lists protected by pool-wide synchronization mechanisms, even though that means all fibers will be constantly contending with each other. And we'll be taking locks more frequently than is ideal in order to simplify the code. Other more scalable approaches are possible—such as isolating state in TLS—but would quickly

complicate what is meant to be a simple example. In addition, the code shown does not check for all error conditions. Clearly a nontoy scheduler would need to be more careful here. Expediency motivated shortcuts aside, the code presented is realistic enough to facilitate a better understanding of what building a UMS might entail.

The Implementation

There are five primary public APIs that users of our `FiberPool` will use: a constructor, a `QueueWork` method to ask that a new work callback be scheduled to run, a `Block` method called from inside a callback whenever it needs to wait, a `Shutdown` method that shuts down and synchronizes with the pool's threads, and a destructor to clean up the resources allocated and used internally by the pool.

Fiber Pool Data Structures

The state managed by each `FiberPool` instance includes the following.

- An array of `HANDLEs` referring to the pool's threads, `m_threadHandles`, and a count of threads, `m_threadCount`. The count is supplied at construction time and remains fixed throughout the pool's lifetime.
- An STL deque of blocked fibers, `m_pBlockedFiberQueue`. Each entry in this list is a fiber managed by the pool that is currently waiting for a `HANDLE` to become signaled and is of type `FiberBlockingInfo *`. Each blocking info data structure contains a pointer to some information about the fiber itself (`FiberState *`) as well as the specific `HANDLE` it is waiting for.
- An STL set of runnable fibers, `m_pRunnableFiberList`, comprised of `FiberState *` entries. Each `FiberState` entry defines some information about the fiber, including the `PVOID` fiber "handle." Fibers are added to this list when they are available to run additional work. This is used to determine whether the pool needs to create a new fiber versus allowing one of the existing runnable fibers to perform the work instead.
- An STL deque, `m_pFiberQueue`, that contains a list of pointers referring to each fiber that has been created by the pool. Each entry is of

type `FiberState *`, and this list allows the pool to delete the fibers when it is destroyed with `~FiberPool`.

- Another STL deque, `m_pWorkQueue`, containing a set of work callbacks that have been queued to the pool with the `QueueWork` API and that are waiting to be run. Callbacks that are actively executing are not contained in this queue. Each entry is of type `WorkCallback *`, which is comprised of a `LPTHREAD_START_ROUTINE` and `PVOID` pair, as are most thread pool style work callbacks.
- A `HANDLE` to an auto-reset event, `m_blockedFiberQueueNewEvent`, which is used to notify blocked threads when a new entry has been added to the blocked queue. The need for this is caused by a tricky implementation detail: we'll see how this is used when we review the implementation later on.
- A `HANDLE` to an auto-reset event, `m_workQueueNewEvent`, which notifies blocked threads when a new piece of work has been placed into `m_pWorkQueue`. If threads have to wait for blocked fibers, a wait-any wait is used so they will wake up and process the new work.
- A Win32 `CRITICAL_SECTION` to protect each of the STL data structures: `m_blockedFiberQueueCrst`, `m_runnableFiberListCrst`, `m_fiberQueueCrst`, and `m_workQueueCrst`.
- A shutdown flag, `m_shutdownFlag`, and a manual-reset event `HANDLE`, `m_shutdownEvent`, both used to communicate the desired shutdown with all of the worker threads in our pool. These threads poll the flag periodically and also wait on the event whenever they must block, ensuring decent responsiveness to any shutdown requests.

Here's the definition of `FiberPool`, `FiberState`, `FiberBlockingInfo`, and `WorkCallback` data types.

```
// Fwd-decls.
struct FiberState;
struct FiberBlockingInfo;
struct WorkCallback;

// A pool of threads on which fibers are scheduled and work items run.
class FiberPool
```

```
{  
    // Threads in the pool.  
    HANDLE * m_threadHandles;  
    LONG m_threadCount;  
  
    // A queue of blocked fibers.  
    CRITICAL_SECTION m_blockedFiberQueueCrst;  
    std::deque<FiberBlockingInfo *> * m_pBlockedFiberQueue;  
    HANDLE m_blockedFiberQueueNewEvent;  
    CRITICAL_SECTION m_runnableFiberListCrst;  
    std::set<FiberState *> * m_pRunnableFiberList;  
  
    // All fibers in the system.  
    CRITICAL_SECTION m_fiberQueueCrst;  
    std::deque<FiberState *> * m_pFiberQueue;  
  
    // The queue of work that needs to be assigned to a fiber.  
    CRITICAL_SECTION m_workQueueCrst;  
    std::deque<WorkCallback *> * m_pWorkQueue;  
    HANDLE m_workQueueNewEvent;  
  
    // To instruct threads in the pool to exit.  
    BOOL m_shutdownFlag;  
    HANDLE m_shutdownEvent;  
  
public:  
    FiberPool(LONG threadCount);  
    ~FiberPool();  
  
    BOOL Block(HANDLE hBlockedOn);  
    void QueueWork(WorkCallback * pWork);  
    void QueueWork(LPTHREAD_START_ROUTINE lpWork, PVOID pState);  
    void Shutdown();  
  
    // Internal.  
    WorkCallback * ContextSwitch(BOOL bBlocked);  
    DWORD ThreadWorkRoutine();  
    void FiberWorkRoutine(LPVOID lpParameter);  
};  
  
// Info about a fiber.  
struct FiberState  
{  
    PVOID m_pFiber;  
    FiberPool * m_pPool;  
    WorkCallback * m_pWork;  
  
    FiberState(PVOID pFiber, FiberPool * pPool)
```

```

    {
        m_pFiber = pFiber;
        m_pPool = pPool;
        m_pWork = NULL;
    }
};

// A simple structure describing a fiber and what (if anything) it
// is blocked on.
struct FiberBlockingInfo
{
    FiberState * m_pFiber;
    HANDLE m_hBlockedOn;
    FiberState * m_pWakingFiber;

    FiberBlockingInfo(FiberState * pFiber, HANDLE hBlockedOn)
    {
        m_pFiber = pFiber;
        m_hBlockedOn = hBlockedOn;
        m_pWakingFiber = NULL;
    }
};

// The closure representing work queued to the pool.
struct WorkCallback
{
    LPTHREAD_START_ROUTINE m_pCallback;
    PVOID m_pState;

    WorkCallback(LPTHREAD_START_ROUTINE pCallback, PVOID pState)
    {
        m_pCallback = pCallback;
        m_pState = pState;
    }
};

```

The constructor for our `FiberPool` is simple. It performs the rote initialization of all of the data structures and then spawns the number of threads requested.

```

FiberPool::FiberPool(LONG threadCount)
{
    // Create queues and associated critical sections and events.
    m_pBlockedFiberQueue = new std::deque<FiberBlockingInfo *>();
    m_pRunnableFiberList = new std::set<FiberState *>();
    m_pFiberQueue = new std::deque<FiberState *>();
    m_pWorkQueue = new std::deque<WorkCallback *>();
}

```

```
InitializeCriticalSection(&m_blockedFiberQueueCrst);
InitializeCriticalSection(&m_runnableFiberListCrst);
InitializeCriticalSection(&m_fiberQueueCrst);
InitializeCriticalSection(&m_workQueueCrst);

m_blockedFiberQueueNewEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
m_workQueueNewEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

// Initialize our shutdown handle.
m_shutdownFlag = FALSE;
m_shutdownEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

// Create our threads. These threads will access the pool
// before we are even done constructing it.
m_threadCount = threadCount;
m_threadHandles = new HANDLE[threadCount];
for (int i = 0; i < threadCount; i++)
    m_threadHandles[i] = CreateThread(
        NULL, 0, &_CallThreadRoutine, this, 0, NULL);
}
```

Keeping with the original disclaimer of no error checking, we don't validate that any of the initialization actually happened correctly. This can cause some serious problems when used in low resource conditions. This is true of much of the code we're about to review. I won't repeat myself for each case, but this same caveat always applies.

Thread and Fiber Routines

The `_CallThreadRoutine` thread-start routine is a simple function that shunts over to the `ThreadWorkRoutine` member on the `FiberPool`, which was supplied via `lpParameter`. All the routine does is convert the newly created thread into a fiber, add it to the global list of fibers in the system, and call the main fiber routine.

```
DWORD WINAPI CALLBACK _CallThreadRoutine(LPVOID lpParameter)
{
    return
        reinterpret_cast<FiberPool *>(lpParameter)->
        ThreadWorkRoutine();
}

DWORD FiberPool::ThreadWorkRoutine()
{
    // Convert the thread to a fiber.
```

```

FiberState * pFiber = new FiberState(NULL, this);
pFiber->m_pFiber = ConvertThreadToFiber(pFiber);

// Add it to the global list.
EnterCriticalSection(&m_fiberQueueCrst);
m_pFiberQueue->push_back(pFiber);
LeaveCriticalSection(&m_fiberQueueCrst);

// Now run the main worker.
_CallFiberRoutine(pFiber);

return 0;
}

```

The `_CallFiberRoutine` function is a wrapper on top of a call to the `FiberPool`'s `FiberWorkRoutine` method.

```

void WINAPI CALLBACK _CallFiberRoutine(LPVOID lpParameter)
{
    FiberState * pState = reinterpret_cast<FiberState * >(lpParameter);
    pState->m_pPool->FiberWorkRoutine(pState);

    // Ensure the fiber we're about to destroy (by exiting the thread)
    // is marked as deleted to avoid double frees.
    pState->m_pFiber = NULL;
}

```

The reason the additional logic is needed after the call to `FiberWorkRoutine` is subtle and should become more apparent when we use `_CallFiberRoutine` in another context later (i.e., when we create additional fibers). The `FiberPool`'s destructor will eventually try to call `DeleteFiber` on each fiber that was ever created by the pool. When a shutdown is triggered, however, the pool cleanly shuts down all threads, which means that some of the fibers will be deleted by virtue of the thread on which they are active exiting. We need to ensure we don't try to delete those fibers twice. Because `_CallFiberRoutine` is always at the top of all fiber stacks in our system, we can hook these exits and fix up state to prevent a subsequent double delete. We do this by setting the `m_pFiber` field on the ambient fiber (retrieved from `GetFiberData`) to `NULL`. Precisely why this works will become obvious when we look at `~FiberPool` later on.

Dispatching Work

We're ready to move on to the scheduler's core functionality. The `FiberWorkRoutine` method is what sits in a loop, dequeuing and executing work items.

```
void FiberPool::FiberWorkRoutine(LPVOID lpParameter)
{
    FiberState * pState = reinterpret_cast<FiberState *>(lpParameter);
    WorkCallback * pWork = pState->m_pWork;
    pState->m_pWork = NULL;

    while (!m_shutdownFlag)
    {
        // If we have work to run, then run it.
        if (pWork)
        {
            pWork->m_pCallback(pWork->m_pState);
            delete pWork;
        }

        // Now grab the next work item or schedule a fiber on the
        // current thread, depending on what the algorithm determines
        // is best. We pass FALSE since we're not blocking. This call
        // will block the current thread until there's work to be done.
        pWork = ContextSwitch(FALSE);
    }
}
```

Sometimes it is the case that the `m_pWork` field of our `FiberState` structure will have already been supplied a `WorkCallback *`. This happens when a fiber is created to run a piece of work. If so, we execute that right away. Otherwise or afterwards, we consult the `ContextSwitch` routine repeatedly to retrieve the next callback to run. This method handles blocking the thread when there isn't any work to do, so `FiberWorkRoutine` isn't a big spin-wait loop. Whenever we have a callback, we run it, passing its `m_pState` as the sole argument, free the `WorkCallback` memory, and continue going for more. We keep looping around until `m_shutdownFlag` has been set to `TRUE`, which occurs when somebody calls the `FiberPool`'s `Shutdown` method.

Cooperative Blocking

Before reviewing `ContextSwitch`, let's take a look at the `Block` routine. That's the only other place the `ContextSwitch` is invoked. When `Block` calls

it, it passes TRUE as the argument, versus FiberWorkRoutine, which always passes FALSE. We'll see what differences result in a moment.

Code running on a fiber can make a call to the method Block, which accepts as an argument a HANDLE. This API places the fiber on a global list of blocked fibers and checks to see if there is work to be done. If there isn't work to be done, or while the thread that made the call to Block is doing additional work, one of the threads in the system may wait on the HANDLE and see that it has become signaled. The blocked fiber will be resumed and the call to Block returns, but possibly on a different thread from the one on which the call was made. This is the only fiber safe way to block in our simple system. Recall earlier that we noted it's difficult to make a fiber based system work correctly unless all blocking goes through the custom fiber aware code, and that's the sole purpose of the Block routine: it gives our scheduler a chance to run additional work if possible, instead of stalling a CPU. Note that a similar approach could be taken for I/O, provided that you were to use asynchronous I/O. This has been omitted here for brevity.

Here's the code for the Block API. It's pretty simple. Again, ContextSwitch is where most of the complicated work happens. In the case of a block, ContextSwitch will never return a new work callback to be run because we do not allow reentrancy in our scheduler.

```
BOOL FiberPool::Block(HANDLE hBlockOn)
{
    // We need to put the current fiber in the queue as blocked.
    FiberState * pFiber =
        reinterpret_cast<FiberState *>(GetFiberData());
    FiberBlockingInfo * pInfo = new FiberBlockingInfo(pFiber, hBlockOn);
    EnterCriticalSection(&m_blockedFiberQueueCrst);
    m_pBlockedFiberQueue->push_back(pInfo);
    LeaveCriticalSection(&m_blockedFiberQueueCrst);

    // Switch may run new work. When it returns we can continue
    // executing whatever the caller was doing, though we may be
    // on a new thread at that point.
    ContextSwitch(TRUE);

    // It's possible we need to add the fiber that just switched
    // to us back to the queue of available fibers.
    if (pInfo->m_pWakingFiber)
```

```

    {
        EnterCriticalSection(&m_runnableFiberListCrst);
        m_pRunnableFiberList->insert(pInfo->m_pWakingFiber);
        LeaveCriticalSection(&m_runnableFiberListCrst);
    }

    delete pInfo;

    // We may have woken up because a shutdown was initiated, vs.
    // an actual handle being signaled. The caller must check for this.
    return !m_shutdownFlag;
}

```

The only additional thing worth noting right now about `Block` is the reason it returns a `BOOL`. (Ignore the bit about the `m_pWakingFiber`. We'll see why that's needed once we look at `ContextSwitch`.) The call to `ContextSwitch` may return for one of two reasons. The first is, that `hBlockOn` has become signaled (in which case we return `TRUE`). The second, however, is that a shutdown was initiated and the thread was unblocked (in which case we return `FALSE`). The caller of our API must check for this condition and terminate whatever they are doing as quickly as possible to ensure a responsive shutdown. Alternative strategies might include throwing an exception from `Block` or even calling `ExitThread`, although for reasons outlined in previous chapters, this approach can prove problematic.

Queueing Work

Briefly, let's look at the `QueueWork` functions because that's the only way that work gets entered into the system. These are extremely simple; they place the callback into the queue and set the auto-reset event so that any threads waiting for new work are awakened.

```

void FiberPool::QueueWork(WorkCallback * pWork)
{
    EnterCriticalSection(&m_workQueueCrst);
    m_pWorkQueue->push_back(pWork);
    LeaveCriticalSection(&m_workQueueCrst);
    SetEvent(m_workQueueNewEvent);
}

void FiberPool::QueueWork(LPTHREAD_START_ROUTINE lpWork, PVOID pState)
{
    QueueWork(new WorkCallback(lpWork, pState));
}

```

One possible optimization is to avoid setting the event if there are no blocked threads. Each call to `SetEvent` requires a kernel transition, so it's not cheap. This is left as an exercise to the motivated reader.

Context Switches

Now it's time to see the `ContextSwitch` logic. Because this function is very long, complicated, and contains a lot of subtle decision choices and implications, we'll review it piece by piece. This is the core of our UMS.

`ContextSwitch` sits in a loop until `m_shutdownFlag` has been set and starts off by looking for new work in the `m_pWorkQueue`. If the work queue is nonempty, it will dequeue the head and arrange for the work to be run. This arrangement happens in one of two ways. If the `bBlocked` argument is FALSE (i.e., it was called from `FiberWorkRoutine`), the work is returned from `ContextSwitch` and the caller will execute it, as we saw above. If the argument is TRUE, however, we cannot run the work directly because we're deep within a callstack that has blocked (i.e., we were called from `Block`). Therefore we must marshal the work to a separate fiber for execution. There are two ways this can happen, and this is where the runnable fiber list comes into play. If there's a fiber already available to run the work, we switch to it; otherwise, we will create a new fiber and switch to it. Using a heuristic to throttle injection of new fibers is probably a good idea. Regardless, the work will then be passed to the switched to fiber inside of its `FiberState`'s `m_pWork` field.

```
// Tries to run an existing fiber if one is available, return a new
// work item for the caller to run (if the caller isn't blocking),
// create a new fiber to run work if all fibers are running or blocked,
// or return NULL if the caller was blocked and their wait has been
// satisfied.
WorkCallback * FiberPool::ContextSwitch(BOOL bBlocked)
{
    FiberState * pState =
        reinterpret_cast<FiberState *>(GetFiberData());
    WorkCallback * pWork = NULL;

    while (!m_shutdownFlag)
    {
        if (!pWork)
        {
            // If the work queue is non-empty, retrieve the new work.
            EnterCriticalSection(&m_workQueueCrst);
```

```
if (!m_pWorkQueue->empty()) {
    pWork = m_pWorkQueue->front();
    m_pWorkQueue->pop_front();
}
LeaveCriticalSection(&m_workQueueCrst);
}

if (pWork)
{
    if (!bBlocked)
    {
        // If we're not blocking, return the work and the
        // caller will execute it.
        return pWork;
    }
    else
    {
        // If the caller is in fact blocking, we cannot run
        // additional work on this thread (to avoid creating
        // reentrant stacks). We will instead switch to another
        // fiber which isn't blocking (if any). If there are
        // no candidates, we will have to create a new fiber.
        FiberState * pRunnableFiber = NULL;

        EnterCriticalSection(&m_runnableFiberListCrst);
        if (!m_pRunnableFiberList->empty())
        {
            std::set<FiberState *>::iterator it =
                m_pRunnableFiberList->begin();
            pRunnableFiber = *it;
            pRunnableFiber->m_pWork = pWork;
            m_pRunnableFiberList->erase(it);
        }
        LeaveCriticalSection(&m_runnableFiberListCrst);

        if (!pRunnableFiber)
        {
            // No runnable fiber found, create a new fiber.
            pRunnableFiber = new FiberState(NULL, this);
            pRunnableFiber->m_pFiber = CreateFiber(
                0, &_CallFiberRoutine, pRunnableFiber);
            pRunnableFiber->m_pWork = pWork;

            // Add it to the global list for cleanup later.
            EnterCriticalSection(&m_fiberQueueCrst);
            m_pFiberQueue->push_back(pRunnableFiber);
            LeaveCriticalSection(&m_fiberQueueCrst);
        }

        SwitchToFiber(pRunnableFiber->m_pFiber);
    }
}
```

```

        // Once we have been resumed, we can be assured
        // we're done blocking.
        return NULL;
    }
}
...

```

Note that after the call to `SwitchToFiber`, it is safe to return `NULL`. The reason is that if `bBlocked` is `TRUE`, we are assured that we previously added the fiber to the `m_pBlockedFiberQueue`. The only possible way that another thread in the system would call `SwitchToFiber` passing this current fiber's `PVOID` would be if it has noticed the `HANDLE` we are waiting for has become signaled. And, therefore, we can return to `Block`, because that's the precise event that `Block` is waiting for.

But what if there isn't work to be done, i.e., `m_pWorkQueue->empty()` returns `TRUE`? Threads that get this far will have to block. This is accomplished with a wait-any style call to `WaitForMultipleObjects`. We wait for any of a number of events to become signaled: the shutdown event, the new work event, the blocked fiber event, and up to `MAXIMUM_WAIT_OBJECTS - 3` of the `HANDLEs` from the blocked fiber list. Blocked fiber entries are removed from the list as the `HANDLEs` are accumulated to ensure that multiple threads do not end up waiting on the same `HANDLE` simultaneously. This is a design decision that isn't strictly necessary and impacts the behavior of our scheduler. While this approach complicates some things slightly—i.e., we get less overlap among fibers in the waits and, therefore, need to introduce the blocked fiber event—it also avoids a bunch of really difficult races that would otherwise arise—i.e., we would need to have synchronization logic to ensure that only one thread switched to a particular fiber, which for persistent signals means cooperation among threads. This is simply a tradeoff.

```

...
// If we got here, there's no additional work to run and
// therefore we will physically block the current thread. We do
// this by waiting for any of the fiber's handles to be
// signaled, or for a new work item to be enqueued, whichever
// comes first. We remove items from the wait queue as we go to
// ensure there is no concurrent waiting on the same handles.

```

```
const int cReserved = 3;
FiberBlockingInfo * ppDequeuedFibers[MAXIMUM_WAIT_OBJECTS -
                                         cReserved];
HANDLE pToWaitOn[MAXIMUM_WAIT_OBJECTS];
pToWaitOn[0] = m_shutdownEvent;
pToWaitOn[1] = m_workQueueNewEvent;
pToWaitOn[2] = m_blockedFiberQueueNewEvent;

// Now build up the list of handles to wait for.
EnterCriticalSection(&m_blockedFiberQueueCrst);
int cDequeuedFibers = 0;
while (!m_pBlockedFiberQueue->empty() &&
       cDequeuedFibers < MAXIMUM_WAIT_OBJECTS - cReserved)
{
    ppDequeuedFibers[cDequeuedFibers] =
        m_pBlockedFiberQueue->front();
    pToWaitOn[cDequeuedFibers + cReserved] =
        ppDequeuedFibers[cDequeuedFibers]->m_hBlockedOn;
    m_pBlockedFiberQueue->pop_front();
    cDequeuedFibers++;
}
LeaveCriticalSection(&m_blockedFiberQueueCrst);

// And lastly, perform the real wait.
DWORD dwRet = WaitForMultipleObjects(
    cDequeuedFibers + cReserved, &pToWaitOn[0], FALSE, INFINITE);
...
```

Note that there is one potential issue with this code. We gather up as many `HANDLE`s from the blocked fiber list as we can pass to the `WaitForMultipleObjects` API, which, in our case, means 61 (i.e., `MAXIMUM_WAIT_OBJECTS` minus the 3 reserved slots we use for pool events). Some `HANDLE`s may not be waited on if we have a large number of blocked fibers. Specifically, if we have more blocked fibers than the count of threads times 61, then some `HANDLE`s won't be waited on until earlier `HANDLE`s have been signaled. If there are dependencies between callbacks such that some `HANDLE`s are only signaled after seeing that others have become signaled, it may lead to deadlock. One approach to solving this might be to use the `RegisterWaitForSingleObject` API when we notice we have more `HANDLE`s than we can wait on at once. Furthermore, it could be that there are other threads that have already begun to wait with non-full wait sets, in which case we might consider waking them up so that they can rebuild and fill their wait set. For the sake of time and space, neither approach is explored here.

There is also an opportunity for a minor optimization here. If we have more than 61 events to wait on, we could remove `m_blockedFiberQueue-NewEvent` from our list and possibly wait on a sixty-second. The `m_blockedFiberQueueNewEvent` event, as we'll see, is set only when we'd like another blocked thread to wake up and try to accumulate more `HANDLEs` for its wait. Since we already have a full set, there is no need to for this thread to participate.

Finally, there is one other design decision that is worth contemplating. Notice that we only check to see whether a wait has been satisfied when the work queue becomes empty. It might be worth checking `HANDLEs` occasionally, perhaps with a 0 timeout instead of `INFINITE`, so that we don't starve blocked callbacks in favor of always running newly enqueued work. This solution wouldn't complicate the implementation too much. We'd just periodically run the existing blocking logic with a different timeout.

We've almost enumerated all of the details. Nobody said building a custom UMS would be easy. We need to look at what happens when the wait returns. There are four basic success cases.

1. If the wait returned because the shutdown event was set (`dwRet` equals `WAIT_OBJECT_0`), we can immediately return `NULL`. We don't bother worrying about the fact that the blocked fiber queue is now missing entries (since we dequeued them) because the pool is terminating anyway. Both the `FiberWorkRoutine` and `Block` method check the shutdown flag, so they will do the right thing when we return.
2. If the wait returned due to new work arriving in the work queue (`dwRet` equals `WAIT_OBJECT_0 + 1`), we will enqueue the blocking information we removed back into the queue so other threads can wait on these events instead, set the `m_blockedFiberQueueNewEvent` so threads that are already waiting can add the `HANDLEs` to their wait set, and then go back around our loop to retrieve the work from the queue and run it.
3. If we were awakened because the blocked fiber event was set (`dwRet` equals `WAIT_OBJECT_0 + 2`), this is just a hint by another thread that we should rebuild our wait list. While there are opportunities for optimization here, we currently loop back around and execute the

same logic above. If we find the work queue is empty, we'll rebuild our wait set and reissue the wait.

4. Finally, we may have been awakened because one of the blocked fibers' HANDLES was signaled. If that is the case, we will just add all of the removed waits back to the blocked fiber queue, minus the one that woke up, and switch to the awakened fiber so it can execute. When we do this, we pass the calling fiber's FiberState as `m_pWakingFiber`. As we saw earlier in the `Block` routine, this causes the awakened fiber to enqueue the calling fiber back into the runnable list. We do this so that if subsequent work is found and a runnable fiber is needed, the aforementioned logic will find this particular fiber and pass the work to it.

And finally, we omit any detailed discussion of how to handle errors. (Also note that we make no special mention of `WAIT_ABANDONED_0`. Using mutexes in a fiber based system is a little silly because they imply thread affinity.) Here's the code that implements all of this logic, concluding the `ContextSwitch` function.

```
...
if (WAIT_OBJECT_0 <= dwRet &&
    dwRet < (WAIT_OBJECT_0 + cDequeuedFibers + cReserved))
{
    int index = dwRet - WAIT_OBJECT_0;
    if (index == 0)
    {
        // We got the shutdown signal. Terminate immediately.
        return NULL;
    } else if (index == 1 || index == 2) {
        // Either new work arrived at the queue or additional
        // waits were added. Restore the queue and then loop
        // back around to dispatch the work or regather waits.
        if (cDequeuedFibers > 0)
        {
            EnterCriticalSection(&m_blockedFiberQueueCrst);
            for (int i = 0; i < cDequeuedFibers; i++)
                m_pBlockedFiberQueue->push_front(
                    ppDequeuedFibers[i]);
            LeaveCriticalSection(&m_blockedFiberQueueCrst);

            // Notify other threads there are available waits.
            if (index == 1)
                SetEvent(m_blockedFiberQueueNewEvent);
        }
    }
}
```

```

        continue;
    } else {
        // A specific wait was satisfied. Dispatch the fiber.
        index -= cReserved;

        // First add other waits back to the queue.
        if (cDequeuedFibers > 1)
        {
            EnterCriticalSection(&m_blockedFiberQueueCrst);
            for (int i = 0; i < cDequeuedFibers; i++)
                if (i != index)
                    m_pBlockedFiberQueue->push_front(
                        ppDequeuedFibers[i]);
            LeaveCriticalSection(&m_blockedFiberQueueCrst);
            SetEvent(m_blockedFiberQueueNewEvent);
        }

        // Now switch to the fiber and go.
        if (ppDequeuedFibers[index]->m_pFiber != pState)
        {
            // If not a blocking fiber, ask that they add us
            // to the runnable list.
            if (!bBlocked)
                ppDequeuedFibers[index]->
                    m_pWakingFiber = pState;

            SwitchToFiber(
                ppDequeuedFibers[index]->m_pFiber->m_pFiber);
        }

        // Once we've been resumed, waiting is done. Our state
        // might contain work that we need to perform.
        return pState->m_pWork;
    }
}
else
{
    // Need to handle other return values here.
    return NULL;
}
}

// The shutdown flag was true.
return NULL;
}

```

Shutdown

The only thing left to look at is the `Shutdown` method and the `~FiberPool` destructor. It's a requirement that `Shutdown` be called on the pool before

deleting it, otherwise the threads instantiated by the pool will try to concurrently access the data structures and resources that the destructor frees. Shutdown handles the synchronization and blocks until all threads have been terminated cleanly. Note that runaway work in the callbacks can cause this to block forever, so some form of cancellation or time based escalation to a more aggressive shutdown policy (via TerminateThread) may be worth considering.

Shutdown is simple. It sets the shutdown flag, sets the event, and then waits on and closes each of the thread's HANDLEs, ensuring it doesn't return until all threads have been shut down completely.

```
void FiberPool::Shutdown()
{
    // Notify threads to exit and wait for them.
    m_shutdownFlag = TRUE;
    SetEvent(m_shutdownEvent);
    for (int i = 0; i < m_threadCount; i++)
    {
        WaitForSingleObject(m_threadHandles[i], INFINITE);
        CloseHandle(m_threadHandles[i]);
    }
}
```

And as you would imagine, `~FiberPool` is the inverse of `FiberPool`, that is, all of the allocated resources are freed. It also enumerates the global list of all fibers allocated and deletes any of them that haven't already been deleted by virtue of the fact that they were active on a thread at the time of shutdown.

```
// Note that this is only safe after the pool's been shut down.
FiberPool::~FiberPool() {
    // Close our event and critical sections.
    CloseHandle(m_shutdownEvent);
    CloseHandle(m_workQueueNewEvent);
    CloseHandle(m_blockedFiberQueueNewEvent);

    DeleteCriticalSection(&m_workQueueCrst);
    DeleteCriticalSection(&m_fiberQueueCrst);
    DeleteCriticalSection(&m_runnableFiberListCrst);
    DeleteCriticalSection(&m_blockedFiberQueueCrst);

    // Delete the fibers and associated state.
    for (std::deque<FiberState *>::iterator it = m_pFiberQueue->begin();
         it != m_pFiberQueue->end();
         it++)
```

```
{  
    FiberState * pState = *it;  
    if (pState->m_pFiber)  
        DeleteFiber(pState->m_pFiber);  
    delete pState;  
}  
  
// Delete the lists.  
delete m_pWorkQueue;  
delete m_pFiberQueue;  
delete m_pRunnableFiberList;  
delete m_pBlockedFiberQueue;  
}
```

A Word on Stack vs. Stackless Blocking

A common characteristic of fiber based UMS's is that a fiber's stack remains fully intact while it blocks. This was true of our above sample. While this is the most intuitive thing to do for most Windows programmers—and the closest to what you would do in a simple, sequential program—it isn't necessarily the most efficient approach. Each stack consumes a fair amount of virtual memory address space and physical memory for the portion that has been used. Additionally, as waits are satisfied, we need to switch stacks, which, while cheaper than thread based context switching, can carry large costs due to thrashing the processor's caches and having to page back in the possibly paged out stack pages.

What other approaches might be viable as alternatives, then? We saw in Chapter 7, Thread Pools, how to register wait callbacks with the thread pool as a way of avoiding too many blocked stacks in a process. That approach is similar in that we were able to use as few physical threads as possible to perform the waiting. I also mentioned that the changes to the method of programming are fairly substantial. The callback that runs when the registered kernel object becomes signaled needs to know enough to "kickstart" the remainder of the work again. There is also the question of whether the original thread that began the work is able to just go away that easily; callers all the way up the stack may be expecting answers to be produced in a sequential fashion. For very simple, event-loop style systems this approach can be made manageable; but as a general purpose solution to arbitrary waits nested deep within complex callstacks, the burden is much higher.

The Microsoft Robotics SDK contains an interesting technology called the Concurrency and Coordination Runtime (CCR). The CCR is meant to make stackless and nonblocking asynchronous programs simpler. In fact, one of the main motivations behind the CCR's development was to solve this very problem and, therefore, you can only ever wait for an event by using a stackless continuation. The cognitive familiarity gap between synchronous, stack based programming and the CCR approach is large, but is worth exploring, even if only for educational purposes. The CCR is available only to managed code programmers and is not currently an official component in the .NET Framework.

Where Are We?

In this chapter, we took a close look at fibers. Fibers are lighter weight than threads because they are managed entirely in user-mode, avoiding kernel bookkeeping and expensive context switches. We then built a complete (albeit simple) user-mode scheduler (UMS) to manage mapping fibers onto threads, swap them when one blocks, and so on. Fibers are seriously limited on Windows because very little of the software "out there," including Win32 itself, is aware of them. Therefore their applicability is quite limited.

And with that, we've concluded the Mechanisms Section of the book. Next we turn to some of the more useful Techniques that can be used to build real concurrent programs. We will begin with a review of memory consistency models and lock free programming.

FURTHER READING

- C. Brumme. Hosting, Weblog article, <http://blogs.msdn.com/cbrumme/archive/2004/02/21/77595.aspx> (2004).
- R. Chen. Using Fibers to Simplify Enumerators, Parts 1–3, Weblog articles, <http://blogs.msdn.com/oldnewthing/archive/2004/12/29/343664.aspx>, <http://blogs.msdn.com/oldnewthing/archive/2004/12/30/344281.aspx>, and <http://blogs.msdn.com/oldnewthing/archive/2004/12/31/344799.aspx> (2004).
- K. Henderson. The Perils of Fiber Mode. *MSDN*, <http://msdn2.microsoft.com/aa175385.aspx> (2005).

- L. Osterman. Why Does Win32 Even Have Fibers? Weblog article, <http://blogs.msdn.com/larryosterman/archive/2005/01/05/347314.aspx> (2005).
- A. Shankar. Implementing Coroutines for .NET by Wrapping the Unmanaged Fiber API. Weblog article, *MSDN Magazine*, <http://msdn.microsoft.com/msdnmag/issues/03/09/CoroutinesinNET/> (2003). M. Stall. Managed Debugging Doesn't Support Fibers. Weblog article, <http://blogs.msdn.com/jmstall/archive/2005/03/01/382474.aspx> (2005).
- D. Viehland. Cooperative Fiber Mode Sample, Days 1–11. Weblog articles <http://blogs.msdn.com/dinoviehland/archive/2004/08/16/215140.aspx> (2004). D. Viehland. Fiber Mode Is Gone. Weblog article, <http://blogs.msdn.com/dinoviehland/archive/2005/09/15/469642.aspx> (2005).

PART III

Techniques

10

Memory Models and Lock Freedom

OVER THE PAST several chapters, we've seen how threads communicate with one another, often with nothing but reads from (loads) and writes to (stores) shared memory locations. We also saw that synchronization is necessary to prevent data races when doing so. All of this discussion has been oversimplified. There are forms of interthread loads and stores that can be done without heavy-handed, critical-region style synchronization. Doing this right often requires a deep understanding of your compiler and hardware architecture, specifically the atomicity and ordering guarantees made with respect to load and stores. With such an understanding, code can be written to avoid some overhead and to improve scalability and liveness. But this comes at the cost of more intricate and difficult to understand code.

This practice is often informally called **lock free programming**. Such code typically avoids full-fledged locks for hot code paths by exploiting memory model guarantees, but can still end up using hardware atomic instructions or locks in less common code paths. In some cases, locks can be avoided altogether, which falls into the category of **nonblocking programming**. In this chapter, we'll examine some aspects of lock free techniques: why they can offer advantages over lock based programming, the fundamentals you need to know to be successful with them, why

they are often difficult to get working right in practice, why many lock free algorithms can appear to run correctly on some machines only to fail on others, and conclude with useful and safe lock free programming approaches and techniques.

If this sounds difficult, it is. In the majority of all concurrent programs, low lock programming is a premature optimization. It can quickly destroy the correctness of your program, so it is not to be taken lightly. Worse, testing concurrency algorithms is still a mysterious art, even when locks are involved, and eschewing them altogether makes life more difficult. Understanding why these techniques are possible, however, is intellectually stimulating and, at the very least, will deepen your understanding of concurrency, so it is worth exploring.

Memory Load and Store Reordering

Critical regions, when built right, ensure atomicity and serializability among regions running concurrently on different threads. This is a fundamental correctness property. This guarantees that a store to memory location x inside some critical region A will be visible by the time any other thread subsequently loads the value of x from inside the same region A. We say the first thread's critical region A (including its store to x) "happens before" and "synchronizes with" the second thread's region A (including its load of x). This property is easy to take for granted, but is important to understand. We'll examine why this is so later on.

Once you leave the realm of critical regions (e.g., Win32 CRITICAL_SECTIONS and CLR Monitors), these assumptions no longer hold. We probably all expect that a multivariable update isn't safe outside of such a region (since a thread could see the update "in between"), but many would be surprised that lockless, single-variable updates aren't always safe either.

Memory operations are routinely reordered by the software and hardware responsible for executing your program.

1. Compilers often perform optimizations that result in loads and stores being moved, eliminated, or added in the process of transforming source text into compiled program instructions. This is called **code**

motion, and is done with the intent of improving performance by executing fewer instructions, optimizing register usage, accessing related memory closer together (spatial locality), and / or accessing memory less frequently. A compiler must preserve sequential behavior when moving code, but can reorder things in ways that change the code's behavior when it is run in a multithreaded setting.

2. Modern processors employ **instruction level parallelism (ILP)** techniques such as pipelining, superscalar execution, and branch prediction to overlap the execution of many instructions. The aim is to reduce the total cycle time taken to execute a set of instructions. A pair of memory loads from separate locations a and b may execute simultaneously in the processor's instruction pipeline, for instance, and, although a textually preceded b in the original source code, b may be permitted to complete before a. This may be legal if the processor believes it is harmless, that is, there is no dependency between the two.
3. The computer architectures on which Windows runs employ a hierarchy of fast caches to amortize access to main memory. Some cache can be shared among processors, while other levels in the hierarchy are not. Many processors also employ write buffers that delay stores. Although it's convenient to view memory as a big array of values that are read from and written to directly, caches break this model. They must be kept globally consistent through a hardware facility called **cache coherency**. Different architectures employ different coherency policies, governing precisely when writes will actually reach main memory and when loads must refresh the local processor cache. These factors can cause loads and stores to appear to have executed out of order.

This hierarchy of transformation can be viewed pictorially in Figure 10.1.

All three of the above categories will typically be lumped together under the term **instruction reordering**. Most programmers need not be concerned with this. But those who are interested in low level concurrent programming routinely need to think about it. Three distinct notions of "order" are important to understand.

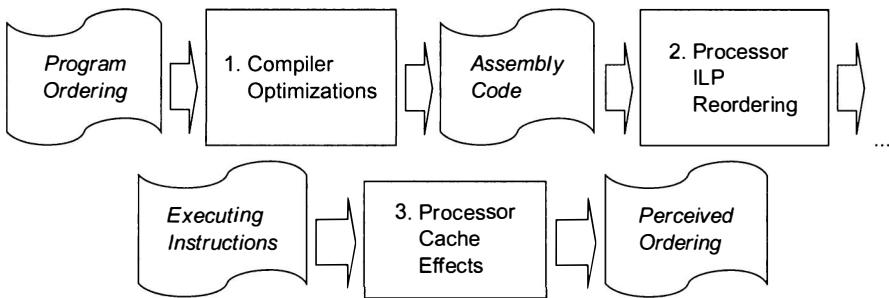


FIGURE 10.1: Transformations that lead to instruction reordering

1. **Program order.** The order in which operations appear in the textual source code.
2. **Actual execution order.** The order in which operations happened during a particular execution of some program. This includes the possibility that some operations that appeared in the original source code did not execute.
3. **Possible execution orders.** Notice that “orders” is plural here. An execution order is one of many possible execution orders that could arise, depending on various factors, such as what optimizations are turned on in your compiler, the number of processors, the layout of caches, the cache coherency policy of the target machine, and so on. This is crucial to understand for any concurrent program because if *any* erroneous execution order is possible, it does not matter whether it actually happens; it’s a bug.

Instruction reordering is not an academic or theoretical problem. It happens quite frequently. It just so happens that sequential code and concurrent code that uses locks are both shielded from these kinds of problems. Since these are (by far) the most prevalent kinds of code you’re apt to encounter, reordering seldom arises in everyday life. Systems level code and highly parallel systems more frequently have to worry about such things. Common patterns like double-checked locking usually give higher level developers first taste of these sorts of issues (more on this later).

What Runs Isn't Always What You Wrote

As a simple motivating example of what can go wrong due to instruction reordering, let's take a look at the following program. Imagine that the two shared variables, x and y , both contain the value 0 at the outset. Two threads, t_0 and t_1 , execute a separate sequence of instructions.

t ₀	t ₁
$x = 1;$	$y = 1;$
$a = y;$	$b = x;$

Is it possible that $a == b == 0$ after threads t_0 and t_1 have both run once? Aside from the mind bending nature of this problem, an answer of "yes" at first seems ridiculous. We might reason this as follows: if we plot this program's execution on a timescale, either the statement $x = 1$ or $y = 1$ must execute first; therefore, no matter what instruction is chosen to run next, the read of the written variable will occur later in time, and it should, therefore, see the previously written value.

The only legal orderings based on this reasoning would be:

Time	t ₀	t ₁ (a)	t ₁ (b)	t ₁ (c)	t ₁ (d)	t ₁ (e)
0		$y = 1$				
1		$b = x$	$y = 1$			
2	$x = 1$					
3		$b = x$	$y = 1$			
4			$b = x$	$y = 1$		
5	$a = y$					
6				$b = x$	$y = 1$	
7					$b = x$	
Values		$a == 1,$	$a == 1,$	$a == 1,$	$a == 1,$	$a == 0,$
		$b == 0$	$b == 1$	$b == 1$	$b == 1$	$b == 1$

All of these appear to have run in the original program order and all looks well.

The answer to the original question—can $a == b == 0$ occur—is “yes” (more accurately, “possibly”) because of instruction reordering. The program can be morphed into any permutation of the four instructions, either statically (by the compiler) or dynamically (by the processor or memory system). The program could appear to have been written like this instead (among other possibilities).

t0	t1
$a = y;$	$b = x;$
$x = 1;$	$y = 1;$

If that’s the code we had written, surely we’d notice a problem with it! The stores occur *after* the loads, so it’s certainly possible that both threads would see a value of 0. It is suddenly painfully obvious why the outcome $a == b == 0$ is possible:

Time	t0	t1 (a)	t1 (b)	t1 (c)	t1 (d)	t1 (e)
0			$b = x$			
1		$y = 1$	$b = x$			
2	$a = y$					
3			$y = 1$	$b = x$		
4				$y = 1$	$b = x$	
5	$x = 1$					
6					$y = 1$	$b = x$
7						$y = 1$
	Values	$a == 1,$ $b == 0$	$a == 0,$ $b == 1$			

These kinds of errors are often not easy to find. Multiple processors may need to be involved to trigger problematic behavior, code might need to have been inlined to expose the optimization that would perform problematic code motion, and so on. This specific reordering will happen with regularity in practice due to the pervasive use of store buffering.

There are trickier examples that challenge some basic assumptions about how code executes. Imagine a situation where three threads are involved, t0, t1, and t2, as well as three variables variables x, y, and z; they begin life with values of 0.

t0 <code>x = 1;</code>	t1 <code>while (x == 0) ;</code> <code>y = 1;</code>	t2 <code>while (y == 0) ;</code> <code>z = x;</code>
----------------------------------	---	---

Is it possible that after all the threads have run, the outcome would be: $x == 1$, $y == 1$, $z == 0$? This too seems ridiculous: for t1 to have written 1 to y, it must have seen x as non-0; therefore, if t2 sees y as non-0, you'd expect it to see x as non-0 too (due to something called **transitive causality**). In fact, the surprising answer is "yes," the outcome could be possible. No modern processors on which Windows runs specifically permit violation of transitive causality, although some older processor architectures did (for instance, notably the first round of Pentium 4 SMPs). If you run into an occurrence of this at the processor level, it's likely a processor bug. But this fact doesn't matter much; compilers can still perform code motion optimizations that would break the above algorithm.

Despite all of this being very compiler and processor dependent, all is not bleak. Three things bring low lock programming back into the realm of possibilities for programmers.

- No matter what, no component that affects instruction ordering will break the sequential evaluation of code. We are only worried about loads and stores used for inter thread communication.
- Related, data dependence limits what can be reordered. This makes reasoning about the possible execution orderings for a piece of code slightly simpler, as we'll look at soon.

- All platforms provide a memory consistency model, or just memory model for short, which specifies very precise rules around what possible reorderings are permitted. This more abstract model of the machine can be used to write relatively portable code that works across many architectures.

Throughout this chapter, we will examine the memory models relevant to Windows programming and various ways of controlling the possible execution orders of a given program explicitly to ensure that the execution orders that arise result in a correct execution of the program. This includes using **interlocked instructions** in place of ordinary loads and stores, keyword annotations (like `volatile`), explicit **memory fences**, and the like. Most of the remainder of this chapter is dedicated to exploring these facilities.

Critical Regions as Fences

Using critical regions shields you from all of these reordering issues. That's because critical region primitives, such as Win32's critical section and the CLR's monitor, work with the compiler, CPU, and memory system to prevent problematic instruction reordering from happening. All correctly written synchronization primitives do this. If the example above was written to use critical regions, no reordering may legally affect the end result.

<pre>t0 Enter_critical_region(); x = 1; a = y; Leave_critical_region();</pre>	<pre>t1 Enter_critical_region(); y = 1; b = x; Leave_critical_region();</pre>
---	---

As we'll see later, entering a critical region ensures there is a fence such that no code after it may move outside of the critical region. Similarly, leaving the critical region ensures no code before the release of the lock may move outside of the region. The lock implementer gets to decide whether exits employ full fences because it is typically OK for code to move from outside into the regions. Using full fences often helps to ensure a fairer system: for example, a lock release that doesn't use a fence could result in the release being delayed in a store buffer; if the releasing thread tried to acquire the lock again, it would have an unfair advantage over other threads in the system.

Most developers writing concurrent software should stick to the synchronization primitives provided by Windows and the CLR and, in doing so, can remain totally unaware of memory reordering. We'll see why this works a bit later when we look at fencing mechanisms.

Data Dependence and Its Impact on Reordering

There are some basic restrictions on what type of reordering can happen in practice, without need for changes to your program. Compilers and processors are careful to respect **data dependence** between operations when moving them around. Not doing so would render correctly written algorithms incorrect, even when run sequentially.¹ In this context, data dependence applies only to operations in a series of instructions executing on a single processor or thread. In other words, dependencies between code running on separate processors are not considered.

There are three kinds of data dependence.

The first kind, **true dependence**, a.k.a. **load-after-store** dependence, occurs when some location is loaded from after having been stored to. The load cannot move before the store or the program would see an old, out of date value.

```
x = 1; // S0
y = x; // S1
```

In this code, a store to `x` is made at S0 and then a load of `x` is made at S1. If the order of instructions were swapped, the result would be wrong. Imagine that `x` originally held the value 0. Because `x` would be read before the value 1 had been written to it, then `y` would erroneously contain 0 (instead of 1) after executing this code.

The second type of data dependence, **output dependence**, or **store-after-store**, occurs when the same variable is written to multiple times. We cannot reorder these instructions, or else earlier stores would pass later ones, and overwrite their values,

```
x = 0; // S0
x = 1; // S1
```

1. Processors like Alpha are known to perform some suspicious reordering that can violate data dependence. Modern versions of Windows need not consider Alpha architectures.

If we were to swap S0 and S1, the variable x would contain the value 0 instead of 1 after they were done. This is incorrect, and, therefore, this reordering must be disallowed. Compilers often combine such writes into one, deleting the first, but this preserves the end value and is not the same as reordering them.

The third and final type of data dependence is **antidependence**, a.k.a. **store-after-load**. If a value is written to after it has been read, the program author probably expects the load to observe the variable's value as it was before the store happened.

```
y = x; // S0  
x = 1; // S1
```

If we imagine x originally holds the value 0 in this particular example, moving the store at S1 before the load at S0 would erroneously cause y to equal 1 instead of 0.

Data dependencies are also transitive. For example.

```
x = 1; // S0  
y = x; // S1  
z = y; // S2
```

In this particular example, S2 has a true dependence on S1 and S1 has a true dependence on S0. Because this dependence is transitive, S2 therefore also has a true dependence on S0.

Hardware Atomicity

Modern processors provide physical atomicity at a fine-grained level. Recall from Chapter 2, Synchronization and Time, that the basic purpose of a critical region is to provide logical atomicity at a higher level. Critical regions are typically implemented through a combination of software and hardware, taking advantage of the kinds of atomic operations we're about to see. These same atomic operations are the building blocks out of which low lock code is written too. We'll later use these guarantees and various primitives discussed in this section to build some real examples of low lock code.

But first: What kinds of atomicity, if any, do ordinary load and store instructions enjoy?

The Atomicity of Ordinary Loads and Stores

Aligned loads and stores of pointer sized values (a.k.a. **words**) are atomic on the kinds of processors on which Windows code runs. A pointer sized value in this regard means 4 bytes (32 bits) on a 32-bit processor and 8 bytes (64 bits) on a 64-bit processor. Load and store atomicity is therefore directly dependent on how memory is allocated and the target architecture's **bitness**.

An aligned chunk of memory begins at an address that is evenly divisible by the particular unit of memory in question: so, for instance, an address `0x0000000C` (12 decimal) is 4-byte aligned (i.e., it is evenly divisible by 4) but is not 8-byte aligned (i.e., it is *not* evenly divisible by 8); an address of `0x0000000D` (13 decimal) is neither. It is also important to consider the size of the value when determining whether accessing memory will be atomic. For example, if some value is only 2 bytes in size, reading and writing it will be atomic as long as it is within an alignment boundary, such as a field of another aligned data structure. But operations will possibly impact surrounding memory. Similarly, a value that is larger than the size of a pointer can be aligned, but still spans a boundary. This can cause some difficulties, as we'll soon see.

Alignment is controlled by the memory management mechanisms used (for heap memory) and your compiler (for type layout and stack memory). Both are platform dependent, and so we'll discuss what policies VC++ and CLR both use shortly.

Consider what atomicity gives us. An atomic load or store guarantees that it will complete with one indivisible instruction at the level of processor and memory. So, say we have two threads running concurrently: one is constantly loading the value of some shared memory location *x*, and the other constantly changes *x*'s value from 0 and 1, back to 0 again, back to 1, and so on. Assuming the loads and stores involved are atomic—that is, they are aligned and *x* is less than or equal to a pointer in size—then the reading thread will always observe a value of either 0 or 1, as you would expect. It will never see a corrupt value. The corollary is also important to understand and is the topic of the next few paragraphs.

Torn Reads

Loads and stores that do not satisfy these criteria may involve multiple instructions, opening up the opportunity for **torn reads**. Torn reads involve races among reads and writes in which part of a value is loaded *prior* to a

write occurring, while the other part is loaded *after* the write completes. The resulting value is a strange blend of the pre- and post-write state, often falling outside of the legal range for the variable in question. A torn read is not atomic at all. For sequential programs, this hardly matters. But for concurrent ones, a torn read can be a painful event, especially since they are so hard to diagnose.

Torn reads affect the simplest of statements—such as `r0 = *a` and `*a = r0`—in the two cases mentioned above: when `a` is a misaligned, or when it refers to a value that is larger than a pointer. The latter is more common than you'd think because most languages support single-statement loads and stores of large data types. This includes things such as the 64-bit `Int64`, 64-bit `Double`, and 128-bit `Decimal` data types in .NET, `LONGLONG` and `FILETIME` in Win32, and any custom structures copied by-value whose fields add up to more than the size of a pointer.

To illustrate a torn read, imagine we have a static variable, `s_x`, which is defined as a 64-bit `long` in C#. (The same example is obviously applicable to native code too.) Some function `g` reads the value of `s_x` and writes its value to the console, and some function `f` changes its value back and forth between `0L` and `0x1111222233334444L`.

```
class TornReads
{
    static long s_x = 0L;

    static void f()
    {
        if (s_x == 0L) s_x = 0x1111222233334444L;
        else           s_x = 0L;
    }

    static void g()
    {
        Console.WriteLine("{0:X}", s_x);
    }
}
```

Imagine that `f` and `g` are called continuously from two threads running concurrently. Based on the program's definition, we'd probably expect that `g` will only ever witness `s_x` having the value `0L` or `0x1111222233334444L`. But it's entirely possible that `g` may observe the value `0x1111222200000000L` or `0x0000000033334444L` instead. The CLR ensures proper alignment of

64-bit values on 64-bit machines (more on that later); but what if this code ran on a 32-bit machine? In this case, the load and store operations are compiled into multiple machine instructions by the CLR's JIT compiler. The same would be true of a 32-bit C++ compiler.

```
MOV [s_x], 0x33334444
MOV [s_x + 4], 0x11112222
```

And corresponding loads of `s_x` will also consist of two memory moves. (The specific order in which values get written is compiler specific and depends on endianness.) With multiple instructions involved, a red flag should pop up in your head. They can be interleaved concurrently, creating the unwanted behavior above.

To illustrate how this might occur, imagine a thread `t0` is calling `f`, storing the value `0x1111222233334444` into `s_x` and another thread `t1` is calling `g`, to load `s_x`'s value.

Time	<code>t0</code>	<code>t1</code>
0	<code>MOV [s_x], 0x33334444</code>	
1		<code>MOV EAX, [s_x] #0x33334444</code>
2		<code>MOV EAX, [s_x+4] #0x00000000</code>
3	<code>MOV [s_x+4], 0x11112222</code>	

After `t0` has written, the first 4 bytes `0x33334444` to `s_x`, `t1` runs and loads both the low and high 4 bytes. Because `t0` hasn't yet written the `0x11112222` portion, `t1` sees a strange blend of values. After `t1` runs to completion, `t0` finally gets around to finishing its write, but not before it's too late: `t1` has seen a corrupt value of `0x0000000033334444L` and may do any range of peculiar things depending on the program's logic. If this were a pointer value, the program could subsequently dereference it and access memory that lives who-knows-where in the address space. The result won't be good.

With this particular code sequence, it's also not immediately obvious whether `0x1111222200000000L` could also be seen. It doesn't seem possible since `0x33334444` is always written first (though this is of course compiler

dependent). In fact, because of memory reordering, the loads and stores could occur such that this outcome is possible. I mention this only because for very low-level code, it is sometimes possible to exploit the order in which individual words of memory are read and/or written; due to reordering, you must be extraordinarily careful.

Torn reads are often the result of flawed synchronization. Most circumstances call for using locks, which hide these issues entirely. A critical region surrounding the statement `t = *a` or `*a = t` encloses the whole set of compiler-generated load and store instructions, maintaining the appearance that they execute as atomic operations (assuming all access throughout the program is protected appropriately). It's only when a lock is forgotten or lock freedom has been used that this is an issue. A common temptation is to write multiple variables within a lock, but to avoid the lock on the read when only one variable is needed. This is sometimes possible, but you must ensure the reads are atomic. Interlocked instructions of the kind we'll review below also enable you to avoid taking locks when reading or writing large data types under some circumstances.

Alignment and Compilers

Your memory manager and compiler take care of most alignment issues for you. This includes the CLR's GC, the VC++ and the CLR's JIT compilers, and the CRT memory allocation functions `_aligned_malloc`, `_aligned_free`, and related ones.

There are actually two distinct components to alignment: the inherent alignment of a data structure's fields, and the address at which the data structure is allocated. For instance, a data structure with fields properly aligned does little good if the allocator does not respect this alignment. Type layout is typically handled by your compiler, and allocation is done either by your favorite memory allocator when heap allocation is used, or your compiler again when stack allocation is used. As a general rule of thumb, both C++ and .NET align pointer sized values by default across the board: type layout, in addition to heap and stack allocation.

Features are provide for custom alignment in native and managed code, such as aligning at 8-bytes on a 32-bit processor or even to generate misaligned data structures. Moreover, the CRT offers unaligned allocators, although the CLR does not. In VC++, the keywords `__unaligned` and

`_declspec(aligned(#N))` provide the ability to control type layout, and you can of course use the alignment options provided by the aligned `malloc` and `free` CRT functions, opt to use the unaligned ones, or even use a custom memory allocators. In .NET, you can use `System.Runtime.InteropServices.StructLayout` to control the placement and padding of fields. Details of all of these features are outside of the scope of this book.

In some circumstances, alignment leads to wasted space. Imagine two consecutive calls to `malloc`, each demanding 14 bytes of memory. If adjacent memory is chosen, the only way to ensure the second request is aligned on a 4-byte boundary is to waste the trailing 2 bytes from the first request. Many allocators are clever about reducing the amount of wasted space used for padding, but some amount is typically unavoidable.

A compiler can deal with an improperly aligned access in one of two ways: recognize it as such and emit multiple instructions, or attempt to use a single instruction. The latter constitutes a misaligned memory access and, depending on the processor architecture, will result in either a silent fixup by the hardware, a costly fixup by the OS, or a fault (as is the case [by default] on IA64). For data structures that are larger than a word of memory, emitting multiple instructions is necessary, but any of those could be misaligned too. Some newer processors guarantee that misaligned loads and stores are carried out atomically, as long as they fit within the boundary of a cache line, although depending on this is asking for trouble.

The CLR's GC moves allocated memory during compaction and, no matter the alignment of a type's fields and the initial allocation of a value, makes no stronger guarantee than pointer sized alignment about where it will subsequently place the data. For instance, in order to use SSE instructions (e.g., via P/Invokes), you must guarantee 16-byte alignment of data. Even if you manage to allocate data on the heap that happens to be 16-byte aligned, the GC may move it later such that it no longer is. If you want to do this, you'll need to stack allocate memory (because stacks don't move), pin, or use a different memory allocator altogether (such as `Marshal.AllocHGlobal` or `P/Invoking` to `VirtualAlloc` and related functions). For more details about this, see Further Reading, Duffy.

Torn reads can also violate type safety. If you've got a misaligned pointer, reading it could tear, and subsequently dereferencing it could lead you to access an effectively random range of memory as a wrong type. If you're

lucky, this will trigger an access violation. If you’re not, you’ll corrupt some random region of memory. The CLR disallows this because it could compromise type safety. While the default type layout will never generate a type containing a misaligned object reference field, it’s possible to use custom value type layout to generate one. If you ever try to load such a type, a `TypeLoadException` will be thrown, stating “Could not load type ‘Foo’ from assembly ‘Bar’ because it contains an object field at offset N that is incorrectly aligned or overlapped by a nonobject field.” The same guarantees are not made for native.

Alignment is a deceptively complex topic, so we will halt the discussion right here. The above overview should have been enough to give you the basic idea, but for a more thorough treatment on the topic, please refer to the wonderful MSDN article Windows Data Alignment on IPF, x86, and x64, by Kang Su Gatlin (see Further Reading).

Interlocked Operations

Having atomic reads and writes of single memory words is useful, but there is a limit to what can be done with this capability. It’s generally not feasible to implement a critical region primitive based on it, for instance, because doing so requires multiple memory operations. For situations like this, processors offer special primitive instructions specifically for atomic loads and stores in addition to more sophisticated **compare-and-swap** style operations (a.k.a. **CAS**), wherein a memory location may be modified atomically based on some condition.

Other kinds of low-level primitives can be built on top of these special **interlocked** instructions, such as critical regions, events, and lock free code. Interlocked operations also imply certain kinds of **memory fences** that interact with the memory model of the system very directly—and in fact there are variants of them that allow you to control which kinds are used—but we will wait to discuss this until the dedicated section on fences coming shortly.

Interlocked instructions use interprocessor synchronization in the hardware. Years ago, in the pre-Pentium Pro architectures, issuing an interlocked instruction asserted a lock on the entire system bus while it ran. These days, interlocked operations execute within the purview of the cache coherence hardware, using a special mutual exclusive mode when acquiring cache lines. This dramatically reduces their cost. These instructions are still not

cheap, however, and still do sometimes lock the bus when contention is high or when accessing a misaligned address.

A common misconception is that interlocked operations will not work *at all* on misaligned addresses. While this can be less efficient (due to the bus lock noted above) and leads to faults on IA64 as with ordinary load and store instructions, atomicity will never be compromised.

In any case, an interlocked operation typically costs in the neighborhood of hundreds of cycles: typically 50 to 150 cycles on single-socket architectures, but reaching costs as high as 500 cycles on multisocket architectures. NUMA machines will incur even larger overheads, due to internode synchronization. Generally speaking, the more complicated and greater in size the memory hierarchy on the target architecture, the more costly synchronization operations will be, and the more impact to system scalability they will present. It is therefore critical when building low-level software to reduce the number of interlocked operations issued to a minimum.

Exchange

The most basic interlocked primitive is *exchange*: it enables you to read a value and exchange it with a new one as a single, atomic action. On X86-based instruction sets, this translates into an instruction called **XCHG**. Unless you're programming in assembly, or looking at disassembled code, you won't see this instruction being used directly—there are higher level APIs that we'll look at momentarily. Most other instructions that we'll look at also require a **LOCK** prefix to be emitted in the assembly code for them to be truly atomic across multiple processors, but **XCHG** is the one instruction that differs in this regard: a **LOCK** prefix is implied by its usage.

Since most of us aren't programming in assembly, there are Win32 and .NET APIs available from `Windows.h` that allow you to utilize the **XCHG** primitive.

```
LONG InterlockedExchange(LONG volatile * Target, LONG Value);
```

This function is implemented as an intrinsic on all architectures, so no overhead for calling a function is paid. It's as if you wrote assembly code that uses the instructions directly. You can call the intrinsic `_InterlockedExchange` from VC++, although there's no particular reason to do so (since the Win32 function translates directly into the intrinsic).

And in .NET, there is a static method on the `System.Threading.Interlocked` class.

```
public static int Exchange(ref int location1, int value);
```

Both act identically. The first argument is the location that is to be modified, and the second is the value to place into the target location. Notice that the native version requires the location to be marked `volatile`; .NET doesn't verify this, and the compilers complain if you try to take a reference to a `volatile` location. In both cases, and despite the annoying compiler warnings, it's usually a good idea (for reordering reasons) but is not strictly necessary. The returned value is the value that was seen *prior* to modifying the location, that is, as it was just before the call. This is guaranteed to be atomic so that no other value can exist in between the value returned and the one placed there. In this sense, the instruction enables an atomic operation comprised of a read/write pair.

To briefly illustrate a use of XCHG, imagine we want to create a simple spin lock.

```
struct SpinLock
{
    private volatile int m_taken = 0;

    public void Enter()
    {
        while (Interlocked.Exchange(ref m_taken, 1) != 0) /*spin*/;
    }

    public void Exit()
    {
        m_taken = 0;
    }
}
```

This code is not “production quality” because spinning on an XCHG instruction will be costly. The hardware needs to jump through a lot of hoops to make the atomicity guarantees I mentioned before. This incurs cache coherency traffic and grows in cost on multisocket machines. But in any case, this code is interesting because it shows that the `Enter` function needn't perform any comparisons. For every time `m_taken` is assigned the value of 0, only one other thread will witness this value and swing it around to 1.

Because only those threads that exit `Enter` will call `Exit`, mutual exclusion is guaranteed. This may be somewhat surprising because the interlocked operation functions correctly even when `Exit` uses an ordinary store.

There are separate functions in Win32 for manipulating 64-bit and pointer locations.

```
LONGLONG InterlockedExchange64(
    LONGLONG volatile * Target,
    LONGLONG Value
);
PVOID InterlockedExchangePointer(
    PVOID volatile * Target,
    PVOID Value
);
```

The 64-bit function must be emulated on 32-bit architectures, although you may be surprised to find out that 32-bit systems *do* support 8-byte (64-bit) atomic operations. We'll see how later (it depends on the yet to be described but related, `CMPXCHG8B` instruction). Obviously the `InterlockedExchange`-`Pointer` can always be implemented as an intrinsic. There are also variants of each of these that have the suffix `Acquire`—that is, `InterlockedExchange`-`Acquire`, `InterlockedExchangeAcquire64`, and `InterlockedExchange`-`PointerAcquire`—which we will not discuss right now; we'll return to what the acquire means when we discuss fences later.

Similar to Win32, .NET also supports a wider array of convenient `Interlocked`.`Exchange` overloads in addition to the simple `int` based one.

```
public static double Exchange(ref double location1, double value);
public static long Exchange(ref long location1, long value);
public static IntPtr Exchange(ref IntPtr location1, IntPtr value);
public static object Exchange(ref object location1, object value);
public static float Exchange(ref float location1, float value);
public static T Exchange<T>(ref T location1, T value) where T : class;
```

The generic overload of `Exchange` limits `T` to reference types. The reason is that this ensures the size of `T` is not too large, that is, because it'll always be the size of a pointer. If `T` could be a custom `struct`, there would be no limitations to its size, which would require runtime validation and exceptions to safeguard. None of these are implemented as an intrinsic currently, as of .NET 3.5. Future versions of the CLR's JIT compiler may choose to inline them.

There is also some overhead to all interlocked operations that target object fields on the CLR. The reason is that they must go through the GC's write barrier to ensure they are safe. The write barrier is an implementation detail that ensures collections scan the right subset of objects in the heap, based on whether a Generation 0, 1, or 2 collection is happening. Although an implementation detail, it does add some unavoidable overhead that may show up if you ever benchmark native vs. managed performance with respect to interlocked operations.

Compare and Exchange

The XCHG instruction works for simple atomic read/write operations. But some algorithms call for more sophisticated read-compare-and-swap sequences. Each operation like this consists of three independent steps; if written naively, as with ordinary reads and writes, the operation could be interrupted after any such independent part, breaking atomicity.

```
if (destination == comparand)
    destination = value;
```

This is broken: a concurrent update could invalidate destination's value immediately after we've ensured that it is equal to *comparand*, invalidating the whole sequence. In other words, this code is not atomic.

Processors provide a CMPXCHG variant on the XCHG instruction, which not only takes the target location and a value to atomically write to it but also a *comparand* that guards the write; only if the comparand value is found in the target location will the new value be placed there. Otherwise, the location is left unchanged, much like the little code snippet shown before. In either case, the observed value will be returned to the caller. This is a true compare and swap (CAS) operation, and the hardware ensures the whole sequence is atomic when using the LOCK prefix. All of the Win32 and .NET APIs we're about to discuss use this prefix by default.

The CMPXCHG variant is slightly less efficient than XCHG. The reason might be obvious: it has more work to do, needing to perform a comparison and a write. There's a less obvious component to this. After acquiring the cache line, CMPXCHG may find that it needs to give it back and most often the software is responsible for recomputing some state and retrying the operation.

All of this leads to a bit more cache line ping-ponging between processors in situations that exhibit high degrees of contention.

CAS is available to Win32 code through functions in Windows.h.

```
LONG InterlockedCompareExchange(
    LONG volatile * Destination,
    LONG Exchange,
    LONG Comparand
);
```

As with other interlocked instructions, this is commonly implemented as a compiler intrinsic. The intrinsic is available directly in VC++ as `_InterlockedCompareExchange`.

And the .NET Framework exposes a method on the static `Interlocked` class.

```
public static int CompareExchange(
    ref int location1,
    int value,
    int comparand
);
```

To illustrate its use, imagine that, instead of a simple “taken” flag, we want to store the ID of the thread that currently owns the spin lock. This might be useful for debugging purposes. But it cannot be implemented with a simple XCHG because a thread must not overwrite the current value if another thread holds the lock. In managed code, we could make a slight modification to the original algorithm by switching to `CompareExchange` to implement this.

```
struct SpinLock
{
    private volatile int m_taken = 0;

    public void Enter()
    {
        int mid = Thread.CurrentThread.ManagedThreadId;
        while (Interlocked.CompareExchange(
            ref m_taken, mid, 0) != 0) /*spin*/;
    }

    public void Exit()
    {
        m_taken = 0;
    }
}
```

The code behaves nearly identically to the earlier example. It's very common to find algorithms that use CMPXCHG in this way. In other words, where the success criterion for the calling is that the write actually happened. A convenient helper function could be used instead.

```
static bool CompareAndSwap(ref int location, int value, int comparand)
{
    return Interlocked.CompareExchange(
        location, value, comparand) == comparand;
}
```

Just like the XCHG primitive, there are the obvious variants in both Win32 and .NET.

```
LONG LONG InterlockedCompareExchange64(
    LONGLONG volatile * Destination,
    LONGLONG Exchange,
    LONGLONG Comparand
);
LONGLONG InterlockedCompareExchangePointer(
    PVOID volatile * Destination,
    PVOID Exchange,
    PVOID Comparand
);
```

And here are the additional overloads in .NET for different data types.

```
public static double CompareExchange(
    ref double location1,
    double value,
    double comparand
);
public static long CompareExchange(
    ref long location1,
    long value,
    long comparand
);
public static IntPtr CompareExchange(
    ref IntPtr location1,
    IntPtr value,
    IntPtr comparand
);
public static object CompareExchange(
    ref object location1,
    object value,
    object comparand
);
```

```
public static float CompareExchange(
    ref float location1,
    float value,
    float comparand
);
public static T CompareExchange<T>(
    ref T location1,
    T value,
    T comparand
) where T : class;
```

Notice that 64-bit compare-exchange operations are available, even on 32-bit processors, thanks to the `CMPXCHG8B` instruction supported broadly by all modern Intel and AMD processors. This is exposed through `InterlockedCompareExchange64` in Win32 and the 64-bit data type overloads in .NET, such as `long` and `double`.

Atomic Loads and Stores of 64-bit Values

Due to this last point, it is sometimes possible to atomically load and store nonatomic-sized memory locations. In fact, the CLR offers a `public static long Read(ref long location)` method on the `Interlocked` class that exploits this fact. It internally just uses a `CompareExchange` that overwrites the value if it's currently 0, but otherwise leaves it as is, enabling you to read its current contents as an atomic operation, even on 32-bit machines.

You can use this capability to generally perform 64-bit atomic reads and writes on 32-bit processors, avoiding torn reads, and can even conditionalize its use to avoid the cost of an unnecessary interlocked instruction on actual 64-bit machines. In C++, you'd `#ifdef` out uses of `InterlockedExchange64` to become ordinary loads and stores on 64-bit machines, and in managed code you can use a fast runtime check:

```
static void AtomicWrite(ref long location, long value)
{
    if (IntPtr.Size == 4)
        Interlocked.Exchange(ref location, value);
    else
        location = value;
}

static long AtomicRead(ref long location)
```

```
{  
    if (IntPtr.Size == 4)  
        return Interlocked.CompareExchange(ref location, 0L, 0L);  
    else  
        return location;  
}
```

If we're lucky, the `if` check will be optimized away by the JIT compiler, since `IntPtr.Size` (a.k.a., `sizeof(void*)`) is a constant known at JIT compile time. Notice that the `AtomicRead` function has been written out long-hand, to use `Interlocked.CompareExchange`, rather than being defined in terms of the existing `Interlocked.Read` function. This is just for illustration purposes. We specify a value of `0` for the comparand and value so that unless the current value of the target is `0` there is no actual write performed. But if one is performed, the value is unchanged. Because `CompareExchange` returns the value seen, we just return that.

Using this trick for loads is patently not the most efficient way to perform a read operation: an interlocked operation unconditionally acquires the target address's cache line in exclusive mode, possibly invalidating other processors' cache lines in the process and causing cache coherence traffic and contention. This is particularly wasteful because we don't need to write at all. If many such reads are used close together, this technique can become more expensive (on 32 bit) than using a simple spin lock to protect the sequence. As with any lock free technique, use this with care, and measure, measure, measure. But if you are primarily targeting 64-bit and can tolerate worse performance on 32-bit architectures, this is a perfectly fine approach.

128-bit Compare Exchanges

Some 64-bit architectures support 128-bit (16-byte) interlocked operations. X86 does not support them at all, most X64 processors do, and IA64 does, but in a different way than X64.

Let's first look at what X64 supports. Much like the `CMPXCHG8B` instruction, nearly all X64 processors offer a `CMPXCHG16B` that is atomic in the same way that `LOCK CMPXCHG` is. Some early 64-bit AMD chips didn't offer the same level of support as modern X64 chips do, meaning you technically need to use a `CPUID` to test whether support is present. This makes it harder to write

portable 64-bit code and is the reason why 128-bit interlocked operations are hard to find in the Win32 APIs and are entirely unsupported in .NET.

Aside from writing assembly, the only current way to access CMPXCHG16B is to use the `_InterlockedCompareExchange128` C++ intrinsic.

```
unsigned char _InterlockedCompareExchange128(
    __int64 volatile * Destination,
    __int64 ExchangeHigh,
    __int64 ExchangeLow,
    __int64 * ComparandResult
);
```

The `Destination` pointer refers to a 128-bit location: that is, two adjacent 64-bit values. The `ExchangeHigh` and `ExchangeLow` values are 64-bit values representing the values to place into the destination. And the `ComparandResult` pointer refers to a 128-bit location, such as `Destination`, that contains the 128-bit value to use as a comparison: that is, if the current value doesn't equal that stored in `ComparandResult`, the CAS will fail. It returns 1 to indicate the swap succeeded and 0 to indicate that it failed. In either case, after the call `ComparandResult` will contain the value seen in `Destination` during the attempt.

As with 64-bit interlocked operations above, this capability can be used to simulate atomic loads and stores of 128-bit values.

The support for 128-bit interlocked operations is slightly different on IA64 processors. For this architecture, there is an `InterlockedCompare64Exchange128` Win32 API that does exactly what it says: 64-bits are used for the comparison, but the value to be written is 128-bits.

```
LONG64 InterlockedCompare64Exchange128(
    LONG64 volatile * Destination,
    LONG64 ExchangeHigh,
    LONG64 ExchangeLow,
    LONG64 Comparand
);
```

This operation can be used for situations where the least significant bits contain data to be validated, but the most significant bits are used as a value to be replaced. While certainly much less useful in general than a full CMPXCHG16B instruction, this capability can still be used in limited cases, such as to avoid ABA problems with lock free stacks (as we examine later).

There are also related intrinsics that are preceded with underscores and also acquire and release variants to control the kind of barrier implied by its use. These intrinsics also emulate this operation on X64 processors that don't offer native instructions, although it does so using the aforementioned CMPXCHG16B instruction.

The IA64 processor also supports `_load128`, `_load128_acq`, `_store128`, and `_store128_rel` intrinsics that enable atomic loads and stores of 128-bit data types. There is a little-known secret that certain SSE instructions such as MOVDQU provide atomic 128-bit operations on some architectures. Processors do not guarantee this atomicity, so any implementations that happen to provide it are subject to change in the future.

Bit-Test-and-Set and Bit-Test-and-Reset

Many uses of XCHG are used to swing a single bit between 0 and 1, as shown in the previous example of a spin lock. For this purpose, a special family of bit-test instructions is offered by many, but not all, processors: X86 and X64 offer them, but IA64 does not. There are two variants: bit-test-and-set and bit-test-and-reset, whose instructions are BTS and BTR, respectively. As the names imply, they enable you to test a single bit in a destination memory location and change its value: to on (in the case of a bit-test-and-set) or off (in the case of bit-test-and-reset). When prefixed with LOCK, these instructions execute atomically.

The bit operations are not available in .NET, but are in Win32.

```
BOOLEAN WINAPI InterlockedBitTestAndSet(
    LONG volatile * Base,
    LONG Bit
);
BOOLEAN WINAPI InterlockedBitTestAndSet64(
    LONGLONG volatile * Base,
    LONGLONG Bit
);
BOOLEAN WINAPI InterlockedBitTestAndReset(
    LONG volatile * Base,
    LONG Bit
);
BOOLEAN WINAPI InterlockedBitTestAndReset64(
    LONGLONG volatile * Base,
    LONGLONG Bit
);
```

Each takes a pointer to the location that will be modified, and the index of the bit to test and modify. Notice that the bit argument is not a mask: it's the bit's index itself. The return value will be TRUE if the bit was found to be on before modification, and FALSE otherwise. No matter the return value, the bit will have been changed by the instruction. On processors that support it, any calls to these functions will be compiled into an intrinsic; otherwise the CMPXCHG instruction will be used to emulate the calls.

As an example of the bit-test-and-set instruction, let's return to the spin-lock example from earlier. This time we'll write it in C++:

```
class SpinLock
{
    volatile LONG m_state;

public:
    void Enter()
    {
        while (InterlockedBitTestAndSet(&m_state, 0)) /*spin*/;
    }

    void Exit()
    {
        m_state = 0;
    }
};
```

The only difference here is that we use `InterlockedBitTestAndSet` in the loop. We continue looping until it returns FALSE, meaning we witnessed the bit in the off position.

Any algorithm that uses these functions could have been instead used XCHG; so why would we care about having both? Bit-test-and-set and -reset are slightly more efficient than a XCHG operation. If all you need to do is set or clear a single bit (and you're writing code in C++ and), you should prefer using one of them instead.

Other Kinds of Interlocked Operations

There are a few other useful interlocked operations to accommodate common update patterns. Each of them could be implemented using an

ordinary CAS operation, but are more efficiently done completely in hardware. This includes:

- An XADD instruction, enabling you to atomically add a particular value to a numeric location (when prefixed with LOCK). This capability is exposed to Win32 with the `InterlockedAdd` and `InterlockedAdd64` functions and .NET with the `Int32` and `Int64` overloads of `Interlocked.Add`.
- When prefixed with a LOCK, the INC, DEC, NOT, and NEG single operand logical instructions are carried out atomically. The first two are exposed to Win32 with the `InterlockedIncrement`, `InterlockedIncrement64`, `InterlockedDecrement`, and `InterlockedDecrement64` functions, and to .NET with the `Interlocked.Increment` and `Interlocked.Decrement` static methods, both of which have `Int32` and `Int64` overloads.
- When prefixed with a LOCK, the ADD, SUB, AND, OR, and XOR binary logical operations are also carried out atomically. All but SUB has a function in Win32 exposing its capability: `InterlockedAdd`, `InterlockedAdd64`, `InterlockedAnd`, `InterlockedAnd64`, `InterlockedOr`, `InterlockedOr64`, `InterlockedXor`, and `InterlockedXor64`. None have corresponding methods in .NET.

Although some functions don't have corresponding APIs in one platform or another, you can implement any of these using CAS. In fact, you can even parameterize the modification logic to create a sort of general purpose update routine.

```
static void InterlockedUpdate(ref int location, Func<int, int> func)
{
    int oldValue, newValue;
    do
    {
        oldValue = location;
        newValue = func(value);
    }
    while (Interlocked.CompareExchange(
        location, newValue, oldValue) != oldValue);
}
```

Say you want a routine that XORs some value with another. You could write it easily.

```
static void InterlockedXor(ref int location, int xorValue)
{
    InterlockedUpdate(location, (x) => x ^ xorValue);
}
```

The same example could be written in VC++ instead, and looks nearly identical.

```
void InterlockedUpdate(volatile LONG * pLocation, LONG (*func)(LONG))
{
    LONG oldValue, newValue;
    do
    {
        oldValue = *pLocation;
        newValue = func(value);
    }
    while (InterlockedCompareExchange(
            pLocation, newValue, oldValue) != oldValue);
}

struct XorClosure
{
    LONG m_xorValue;
    XorClosure(LONG xorValue) { m_xorValue = xorValue; }
    LONG doXor(LONG input) { return input ^ m_xorValue; }
};

void InterlockedXor(volatile LONG * pLocation, LONG xorValue)
{
    XorClosure xor(xorValue);
    InterlockedUpdate(pLocation, &xor->doXor);
}
```

Finally, Figure 10.2 contains a chart illustrating some performance differences between four things: code that reads and writes to a shared variable, code that uses an interlocked exchange to publish a new value (keeping in mind this doesn't prevent lost updates), code that uses an atomic increment, and code that uses a custom compare-exchange loop to prevent lost updates. Each of these is called in a tight loop, and the test has been run on several architectures, including single socket all the way up to a 4 socket quad core architecture. A delay of between 10 to 100ns is present

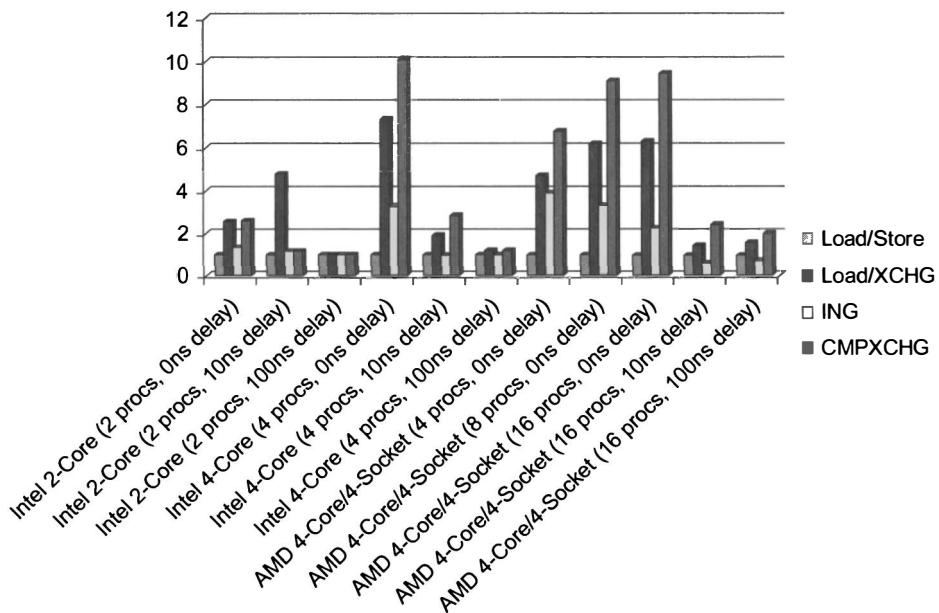


FIGURE 10.2: Illustration of the relative costs of some interlocked operations

in some of the loops to reduce the contention; as you'll see, the relative cost of interlocked operations goes up when this delay is omitted due to the increase in cache contention. The numbers plotted on the graph are relative, so that you can get an understanding of cost relative to ordinary reads and writes. Please don't try to extrapolate any absolute costs; they are apt to vary greatly on different architectures.

Memory Consistency Models

We're now in a good position to tackle the complicated topic of **memory consistency models**, a.k.a. **memory models** for short. If you followed along closely throughout this chapter leading up to this point, the following section should be a breeze.

A memory model specifies precisely which kinds of loads and stores may be moved, under what conditions they may be moved, and to where they may move with respect to one another. The possible memory models fall on

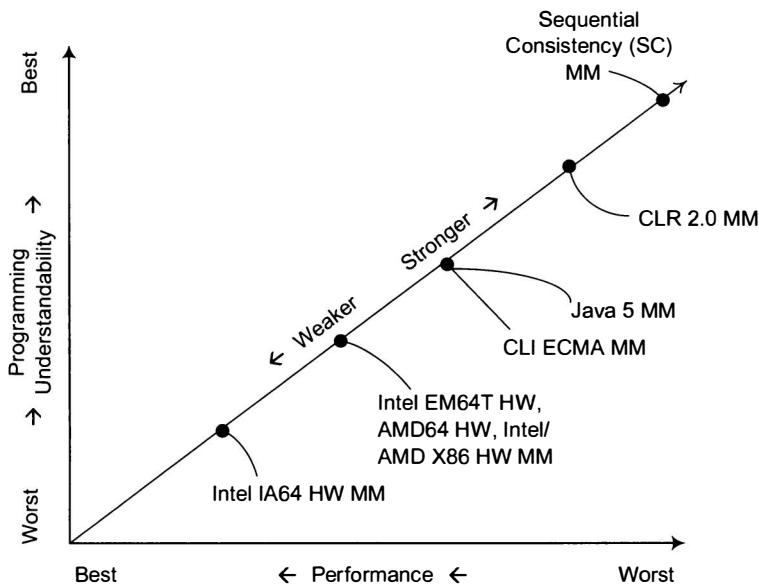


FIGURE 10.3: A spectrum of memory consistency models

a continuous spectrum from *weak* to *strong*. This spectrum is illustrated in Figure 10.3.

The weakest possible memory model allows all loads and stores to be reordered, while still preserving the sequential correctness of the original program (which means not violating data dependence). The strongest possible memory model—referred to as **sequential consistency**—prohibits all reordering, such that what executes is precisely what was written in the text of the program itself (i.e., its program order). Weak memory models offer greater chance for optimizations, while they are harder to program against; strong memory models provide a more understandable and programmable model, but at the expense of optimizations. Anything weaker than sequential consistency is typically called a *relaxed* memory model.

In an ideal world, we would all be programming with sequential consistency. That is, if sequential consistency didn't carry enormous performance implications. As in-order execution becomes more popular in future architectures—to reduce power and complexity—it may become more attractive to pursue sequentially consistent architectures. But for

the time being, those who develop memory models are responsible for analyzing these tradeoffs with their target audience in mind and developing the rules that will deliver the greatest value to their customers.

Because reordering can happen in several places (e.g., compiler versus processor reordering), defining a memory model is a layered process. This affects hardware and compilers.

All hardware architectures must define a memory model. While the reasons for particular kinds of movements aren't always spelled out, movement occurs for the reasons outlined at the outset of this chapter: speculative execution, caches, and other processor level optimizations. The model must be specified fairly clearly so that low-level software developers can program the machine, particularly compiler writers and operating system developers. Taking a dependency on the hardware memory model from higher levels of software is usually problematic because of the discrepancies from one processor implementation to the next and because your compiler also has a say in what kinds of orderings are possible. Hardware vendors are known to specify weaker models than are actually implemented to avoid being forever tied to the stronger model. In other words, they want to reserve the right to implement more clever optimizations in the future that weaken the implemented model.

Some compilers go a step further and define a memory model irrespective of the runtime hardware. The CLR has a strong memory model that presents a consistent model regardless of the architecture being targeted, to make portable code easier to write. This requires special instructions to be emitted on certain architectures, and restricts the kinds of compiler optimizations possible. This is great: it means a programmer may safely depend on the memory model because it will never be weakened and because no knowledge of particular hardware models is required. VC++, on the other hand, doesn't go so far, though it does offer manual controls to restrict the way certain code may be reordered.

We will first look briefly at the various hardware architectures supported by Windows and what sort of memory model guarantees they make. This is useful particularly if you're a compiler writer or do the bulk of your programming in VC++. We'll then move on to fencing, and the additional memory model guarantees made by the .NET platform.

Hardware Memory Models

Instead of spending page after page dissecting each particular kind of memory model in detail, let's begin looking at a high level summary of particular reorderings that you might be concerned with and which architectures that Windows runs on will exhibit them (see Further Reading, AMD x86-64 Architecture Programmer's Manual Volumes 1-5, Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference, Intel Itanium Architecture Software Developer's Manual Volume 3: System Architecture, Intel 64 Architecture Memory Ordering White Paper).

	X86	Intel64	IA64	AMD64
Load-Load	No (except for store buffer/ forwarding)	No (except for store buffer/ forwarding)	Yes	No (except for store buffer/ forwarding)
Load-Store	No	No	Yes	Yes
Store-Store	No	No	Yes	No
Store-Load	Yes	Yes	Yes	Yes

The rows indicate a particular kind of reordering, such as whether a load may move after another load (Load-Load), after another store (Load-Store), and so on. They apply transitively to a stream of instructions. Columns are dedicated to the four architectures with which we are concerned, X86 (which includes IA32 and 32-bit AMD processors), Intel64 (such as the EM64T and modern Intel 64-bit processors like the 64-bit Core Duo), IA64, and AMD64. Each entry represents whether the particular architecture permits the reordering in the row (Yes) or not (No). The more reordering allowed, the weaker the memory model. As you can see, X86, Intel64, and AMD64 are all the strongest, with IA64 being the weakest.

(Those who desire a more thorough and theoretical treatment of memory models are encouraged to read some of the material from the Java JSR133 memory model specification process. These documents use a mechanism called **happens-before** and **synchronizes-with** to describe legal reorderings in terms of causality and visibility. While useful for proving theoretical

properties about an abstract model, the result makes for some rather complicated reading. See Further Reading, Manson, Pugh, and Adve.)

Notice that substantially weaker models, such as Alpha and PowerPC, are not described because current versions of Windows do not run on them. Only certain Windows SKUs, such as Windows Server, currently run on IA64, but that's enough for VC++ and .NET programs to need to consider this architecture during development. In some sense, this is unfortunate because IA64 is the weakest model Windows runs on and yet is rare to encounter in practice (and moreover the hardware is very costly, making it hard to test). This means that IA64 specific memory reordering bugs are the ones that most frequently slip through software development and testing.

Based on recent Intel and AMD processor documentation, the X86, Intel64, and AMD64 memory models prohibit *most* forms of Load-Load reordering, despite what the table shows. Specifically, they permit loads to reorder when satisfying pending writes in the local processor's write buffer. That may cause loads to *appear* to reorder (abstractly) although no physical reordering has occurred. Needing to think in terms of very specific conditions such as this complicates matters, so when in doubt it is safer to simplify to an answer of "Yes, these processors permit Load-Load reordering." In some cases, you can exploit the special rules, but this can add difficulties to writing and maintaining portable (and correct) code.

A few interesting points from this table are worth noting.

- This table doesn't call out the impact of having fences, even though they prohibit *certain* instances of the reorderings identified in the table. Most often, a fence is meant to avoid a certain one of those rows. We'll return to fences soon.
- Processors must maintain single processor consistency, so any movements affecting the same memory location are prohibited due to data dependence.
- Only IA64 freely permits loads to reorder, due to out-of-order execution and a desire to allow speculative and cache-hit loads to retire in the most optimal order possible. X86, Intel64, and AMD64 only allow loads to reorder as a result of local store buffering.

- All four architectures allow stores to move after loads. This is due to the pervasive use of store buffering in all of the aforementioned processors.
- All architectures except IA64 enforce global store ordering. In other words, stores become visible in the order in which they are executed. The lack of global store ordering can be the source of some significant portability issues on IA64.
- All of the above processors ensure transitive causality. An example of transitive causality was shown earlier, where three variables are involved and processors seeing individual writes but not others would cause a great deal of problems.

Some processors have different policies when it comes to instruction caches versus data caches, and, specifically, the ordering of load and store operations. We've limited discussion to ordinary data caches for this chapter. Instruction caches are most concerning to compiler writers with self modifying code, such as JIT compilers that do code pitching or rewriting, for example, Java HotSpot VM. Please refer to the relevant processor documentation for details.

Memory Fences

For a variety of reasons, many of which we'll explore later while looking at lock free algorithms, it is necessary to prevent loads and stores from reordering. The great thing about a fence is that, no matter what architecture you are targeting, and no matter what reorderings that architecture permits, memory fences prevent loads and stores from moving in a very specific way. Fences also come at a cost, however, because they prevent optimizations.

Common Kinds of Fences

Many fence varieties are commonplace.² But only one kind is consistently supported across all of the architectures in which we are interested.

- **Full fence:** Ensures no load or store moves across the fence, in either direction. In other words, instructions that come before the fence

2. It's common for fences to be called **barriers** also. Intel seems to prefer the "fence" terminology, while AMD prefers "barrier." I also prefer "fence," so that's what I use in this book.

will not move after the fence, and instructions that come after the fence will not move before the fence. Most architectures expose a dedicated instruction (e.g., MFENCE) for this.

The fact that the full fence is the only consistently supported fence is acceptable because it's the strongest fence possible. The other kinds of fences are optimizations; a full fence would be correct, but the variants allow certain kinds of loads and stores to move across the fence to avoid unnecessary optimization limitations. Let's review a few of those architecture specific fences.

First, there are two-way fences that apply only to stores or loads. These fences are available in X86 and X64 hardware, but not in IA64.

- **Store fence.** Similar to a full fence, except it only applies to store instructions and freely permits loads to move across the fence in either direction. This is commonly expressed via an SFENCE instruction.
- **Load fence.** Similar to the store fence, except it only applies to load instructions and freely permits stores to move across the fence in either direction. This is commonly expressed with an LFENCE instruction.

As optimizations, these can be useful. For example, a load fence will prevent certain kinds of speculation but will not impact the processor's ability to buffer stores. Likewise, a store fence will prevent some store buffering, but allows the processor to continue speculating.

The next two fences are used on IA64 and in compiler optimizations. They are sometimes called one-way fences, because they allow movement across in a single direction.

- **Acquire fence.** Ensures no load or store that comes *after* the fence will move *before* the fence. Instructions before it may still move after the fence.
- **Release fence.** Ensures no load or store that comes *before* the fence will move *after* the fence. Instructions after it may still happen before the fence.

See Figure 10.4. Notice that instead of applying only to loads and stores, they apply only to a certain direction of movement. These allow certain optimizations to remain, specifically those that result in moving instructions across the fence in the particular direction permitted.

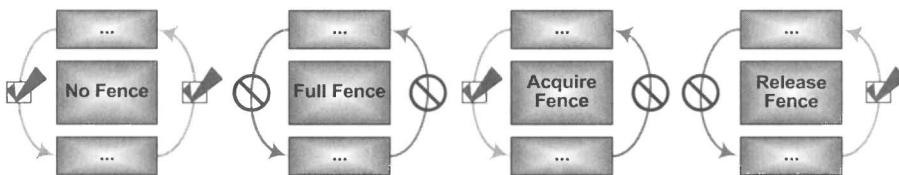


FIGURE 10.4: Kinds of fences and their impact on reordering

Using the variants is a matter of performance: full fences can always be used instead. Using a weaker variant can make reasoning about lock free correctness *more* difficult since some particular reorderings remain legal. While the kind of performance improvement seen by relaxing the fence can make a real difference for low-level code that is called time and time again (e.g., a common OS interrupt routine), as a general rule of thumb, the optimizations are not overly crucial. When in doubt, and when you don't want to write architecture specific code, you can usually rely on full fences to prevent reordering.

It is important to point out that there's a big difference between a full fence at the *compiler* level, a full fence at the *processor* level, and a fence that applies to both. Recall that a myriad of reordering is possible at each level in the software stack. A full fence that only pertains to the compiler does not prohibit reordering at the processor level, and vice versa. If you need to absolutely guarantee that a particular load or store never moves, you'll need a fence that applies to both. It is crucial to recognize the difference, so we'll call it out where applicable.

Creating Fences in Your Programs: Volatiles, Etc.

At this point, you may be wondering how to achieve a fence in your code. It turns out that all of the interlocked operations we just reviewed incur a full fence at the processor level (minus those suffixed with `Acquire` and `Release`—we'll return to that shortly). The fact that C++ requires you to pass a pointer to a `volatile` location *almost* ensures a full fence in the compiler

too (we'll see why this isn't quite true in a bit), and .NET's JIT compilers will truly respect the presence of an interlocked operation as a full fence. So this is the simplest way to achieve a fence and is why most locks (built out of interlocked operations) remain correct and prevent reordering that would break the desired serializability of critical regions.

Creating Fences in .NET. Fences in .NET are simple. Using any method on the `Interlocked` class creates a full fence, as does acquiring a lock, such as the `Monitor` or `ReaderWriterLockSlim` (since both are implemented using interlocked operations). This is great because it ensures that code with lock based synchronization isn't subject to any strange bugs to do with memory reordering. Additionally, you can call the `Thread.MemoryBarrier` static method directly, which also emits a full fence. All of these fences apply both at the JIT compiler and processor level.

Reading a `volatile` variable or using the `Thread.VolatileRead` method is logically an acquire fence and writing to a `volatile` variable or with `Thread.VolatileWrite` method is logically a release fence. (It turns out that `volatiles` aren't always true fences in the emitted assembly code: the .NET JIT compilers rely on specific hardware memory models to make these more efficient.) These fences apply at both the compiler and processor level too and also prevent problematic compiler optimizations like hoisting volatile loads outside of a loop so that concurrent changes are missed. We'll see later when we look closely at memory models that certain loads and stores on .NET imply certain kinds of fences automatically.

Creating Fences in VC++. Fences in VC++ are trickier because the notion of compiler versus processor level is highly controllable. Moreover, the variants of fences are available to you, unlike in .NET, so you can write processor specific code to use one kind over another. Similar to .NET, loads and stores of VC++ `volatile` variables incur acquire and release fences, respectively, and also prevent compiler optimizations such as hoisting outside of loops. There is, however, one *huge* difference between VC++ and .NET: these fences apply only at the compiler level and do not carry through to the processor. This is usually surprising to people the first time they hear about it. Similarly, there is a `MemoryBarrier` macro in `Windows.h`.

that emits a two-way barrier at the processor level, but does not guarantee any effect at the compiler level.

A set of compiler intrinsics forces both compiler and processor level fences in VC++: `_ReadWriteBarrier` emits a full fence, `_ReadBarrier` emits a read-only fence, and `_WriteBarrier` emits a write-only fence. You may also emit certain kinds of acquire and release fences through the use of the Win32 `InterlockedXxAcquire` and `InterlockedXxRelease` family of functions. These have corresponding VC++ intrinsics named `_InterlockedXx_acq` and `_InterlockedXx_rel` that are used when compiling for IA64. On all other architectures, these fall back to using full fences.

Beware of the Release-Followed-by-Acquire-Fence Hazard

One of the trickiest and most often overlooked reordering scenarios is when you have two adjacent fences, specifically a release fence followed by an acquire fence. In both VC++ and .NET, for example, this arises when you have a store of a volatile variable followed by a load of another volatile variable. Notice that the definitions of release and acquire *do not* prevent the two adjacent fences and the operations preceding and following them from being reordered.

As an illustration, let's go back to an example we used earlier.

t0		t1
<code>t0(0):</code>	<code>x = 1;</code>	<code>t1(0): y = 1;</code>
<code>t0(1):</code>	<code>a = y;</code>	<code>t1(1): b = x;</code>

In this snippet, `x` and `y` are shared variables: each thread writes 1 into one, and then reads the other into a local variable (`a` and `b`). One might decide to "fix" this problem by marking `x` and `y` as `volatile` variables. This does not work because both the acquire fence and the subsequent load can move before the store and release fence. The reverse is not true.

The solution is to place a full fence in between the instructions, that is:

t0		t1
<code>t0(0): x = 1;</code>		<code>t1(0): y = 1;</code>
<code>t0(1): _ReadWriteBarrier();</code>		<code>t1(1): _ReadWriteBarrier();</code>
<code>t0(2): a = y;</code>		<code>t1(2): b = x;</code>

.NET Memory Models

Now that we've reviewed the hardware memory models, how to emit fences in your programs, and the like, there's very little else to say. But the .NET memory model does make a couple interesting strengthening guarantees, so we'll look at a table much like the one reviewed earlier in the context of hardware architectures. The memory model detailed in the ECMA and ISO Common Language Infrastructure (CLI) specification is considerably weaker than what .NET 2.0 and beyond implement. This is worth understanding for anybody writing portable code, including code that needs to run on Mono, Silverlight, or Moonlight. Volatile loads and stores are treated differently and are thus called out separately:

	ECMA 1.1	(volatile)	CLR 2.0+	(volatile)
Load-Load	Yes	No	Yes	No
Load-Store	Yes	No	Yes	No
Store-Store	Yes	No	No	No
Store-Load	Yes	Yes	Yes	Yes

The major difference in the stronger 2.0+ model is that it prevents stores from being reordered. (The rules for `volatiles` have always been stronger.) It's not that ECMA 1.1 explicitly *allowed* movement, but it didn't explicitly *disallow* movement either. When the CLR 2.0 was ported to IA64, its initial development had happened on X86 processors, and so it was poorly equipped to deal with arbitrary store reordering (as permitted by IA64). The same was true of most code written to target .NET by non-Microsoft developers targeting Windows.

The result was that a lot of code in the framework broke when run on IA64, particularly code having to do with the infamous double-checked locking pattern that suddenly didn't work properly. We'll examine this in the context of the pattern later in this chapter. But in summary, if stores can pass other stores, consider this: a thread might initialize a private object's fields and then publish a reference to it in a shared location; because stores can move around, another thread might be able to see the reference to the

object, read it, and yet see the fields while they are still in an uninitialized state. Not only did this impact existing code, it could violate type system properties such as `initonly` fields.

So the CLR architects made a decision to strengthen 2.0 by emitting all stores on IA64 as release fences. This gave all CLR programs stronger memory model behavior. This ensures that programmers needn't have to worry about subtle race conditions that would only manifest in practice on an obscure, rarely used and expensive architecture.

In addition to the above rules, there are some subtle restrictions placed on the JIT to do with traditional compiler optimizations. Loads and stores of `volatile` variables can never be introduced or removed, both in .NET and VC++, because they are assumed to be constantly changing. As such, they aren't eligible for being considered loop invariant and hoisted outside of loops: hoisting out of a loop removes all but the first load or store. But for non-`volatile` variables, the question is still an interesting one. VC++ makes no additional restrictions for such variables, requiring a programmer to thoroughly annotate variables as `volatile` where introduction or removal would be a problem, but .NET does.

As an example of when a load might be introduced, consider this code.

```
MyObject mo = ...;
int f = mo.field;
if (f == 0)
{
    // ... do something ...
    Console.WriteLine(f);
}
```

If the period of time between the initial read of `mo.field` into variable `f` and the subsequent use of `f` in the `Console.WriteLine` was long enough, a compiler may decide it would be more efficient to reread `mo.field` twice.

```
MyObject mo = ...;
if (mo.field == 0)
{
    // ... do something ...
    Console.WriteLine(mo.field);
}
```

A compiler might decide this if keeping the value would create register pressure, lead to less efficient stack space usage, and/or if the branch

would be seldom taken (and hence the original value not needed more than once anyway). Doing this would be a problem if `mo` is a heap object and threads are writing concurrently to `mo.field`. The `if`-block may contain code that assumes the value read into `f` remained `0`, and the introduction of reads could break this assumption. In addition to prohibiting this for `volatile` variables, the .NET memory model prohibits it for ordinary variables referring to GC heap memory too.

Removing reads can happen when a compiler detects that one or more of them are superfluous. Similarly, removing writes will happen when a compiler detects that a value is immediately overwritten and that eliminating the intermediary write has no effect on the sequential stream of instructions it is analyzing. The .NET memory model permits coalescing of multiple adjacent loads or multiple adjacent stores to the same location, since it's generally not possible for anybody to notice. This is true even if they are volatile. It's not required for the loads or stores to be adjacent in the program text for this optimization to occur. If some other code motion causes them to become adjacent, the compiler may choose to coalesce them.

Lock Free Programming

As the name implies, **lock free programming** is the practice of writing concurrency-safe code without locks. This sounds simple enough, but it's an error prone practice that requires a deep understanding of everything described in this chapter thus far (actually everything described in this book so far). What we describe here is typically called *nonblocking* in academic papers and the like. There are three kinds of nonblocking algorithms with which we are concerned.

- **Obstruction freedom** means that any thread can always make forward progress through an algorithm if all other threads in the system were to be suspended. In other words, no other thread in the system holds a lock or shared resource that this particular thread would need to wait for in order to proceed.
- **Lock freedom** is stronger than obstruction freedom, and means that anytime a thread fails to make forward progress, we are guaranteed that it is because another thread in the system has made forward

progress. The system as a whole makes forward progress although any one particular thread may be starved.

- **Wait freedom** is the strongest of the three. It means that any given thread in the system is ensured that it will complete in a finite number of steps. In other words, it is not possible for the thread to be starved as with lock freedom.

The distinctions are not overly important for many real systems and are mostly of theoretical interest. So we'll generally refer to all algorithms as lock free when we actually mean nonblocking. There is an important point lurking within: lock free algorithms may still use atomic hardware instructions in the implementation, provided they satisfy the previous criteria. Some might find this misleading because an interlocked operation can be as costly as a lock. There are certainly several lock free algorithms that don't require interlocked operations, but they are less common than those that do. We will even bend the meaning of lock freedom in some cases. For example, double-checked locking can require the acquisition of a lock, but has a lock free component. We will lump discussion of such things in with other lock free programs.

One of these points is worth embellishing: a lock free algorithm can consist of fewer synchronization operations than a lock-based counterpart in some circumstances. For instance, CLR monitors require two interlocked exchanges per acquire/release pair; an algorithm that can achieve the same effect using a single interlocked operation may fare better from a micro-benchmark standpoint. This is not always possible: in fact, lock free algorithms can require *more* synchronizing operations, due to the need for extra fences to avoid reordering problems.

The main benefit for lock free algorithms is actually in the non-blocking nature. Because no threads ever block, and because no one thread can prevent others from making forward progress, the resulting scalability is usually far superior. Context switching is reduced and throughput is increased. (That said, lock free algorithms can often be subject to livelock.)

An additional (less obvious) benefit to lock freedom is reliability. Since the granularity of forward progress must necessarily be compressed down to

a single atomic operation, failure of a single thread cannot compromise the consistency of a lock free data structure. This point is interesting for important OS data structures, for example, but less interesting for user-mode data structures in which a failure part-way through updating a data structure is often catastrophic and results in the whole process being torn down.

Lock free data structures take extra care to implement correctly. Because critical regions can't be used to protect other threads from concurrently seeing the structure in an inconsistent state, the data structure simply cannot *ever* enter into an inconsistent state. In some sense, this makes coding them simpler; if nothing else, the realm of possible algorithms is far smaller and simpler because every update must boil down to a single atomic operation (usually an interlocked operation). This single operation is the **linearization point**—as described in Chapter 2, Synchronization and Time—which is the point at which the update takes effect and becomes visible. If we jotted down the data structure's invariants or even checked them, a typical requirement of lock free code is that the invariants are *never* violated (each atomic update must move the structure from one legal state to another legal state). What typically complicates matters is relying on the memory model, which, as we've seen before, can be tricky business.

Examples of Low-Lock Code

Let's take a look at a few popular and safe examples of low-lock code.

Lazy Initialization and Double-Checked Locking

The *double-checked locking* pattern for lazy initialization is infamous. This is due to its popularity as an efficient initialization mechanism, plus the fact that it fails on several popular hardware memory models. These hardware architectures include Alpha and IA64. It's worth mentioning that *most* variants on the pattern work without a hitch on X86, Intel64, and AMD64. And the CLR 2.0 memory model also ensures that double-checked locking works correctly.

Lazy Initialization in .NET

Here we will see several variants on the idea for .NET. We'll develop a useful and reusable `LazyInit<T>` class that can be used wherever you need lazy initialization.

Double-Checked Locking: The Basic Pattern. Lazy initialization is often used for the singleton pattern. The CLR offers class constructors (a.k.a. static constructors) for static variable initialization, which is often suitable for this.

```
class Singleton
{
    private static Singleton s_inst = new Singleton();

    public static Singleton Instance
    {
        get { return s_inst; }
    }
    ...
}
```

The `s_inst` variable will be initialized by the time the first attempt to access it succeeds. The CLR internally uses a double-checked locking mechanism exactly like that which we're about to discuss to guarantee that no two threads racing to access the `s_inst` field will cause the `new Singleton()` statement to execute more than once. This involves locking when concurrent accesses are detected. Although you should use this built-in mechanism wherever possible, there are a few reasons it may be insufficient for all cases.

- The CLR doesn't guarantee *when* the class constructor will run other than to say it will happen at least in time for the first field access. Popular languages like C# and VB emit code so that it happens lazy upon the first access to the `Singleton` class anywhere in the program.
- There is only a single class constructor per class. If there are several variables to initialize, involving complicated or costly logic, you may not want to initialize them all on the first access to `Singleton`. Instead, you may want to manage each one individually.
- The guarantees this provides may be too strong. We will look, in a while, at a variant on the basic double-checked locking pattern that permits multiple objects to be created but ensures that only one gets published. This avoids locks.
- Finally, and perhaps most importantly, the class constructor mechanism only works for static variables. They won't work for cases in which you'd like to use lazy initialization for the instance fields of an object.

As a first approximation of a lazy initialization routine—and as an example to motivate why the trickier pattern is required—let’s look at a naïve (and poorly performing) attempt.

```
class LazyInit<T>
{
    private T m_value;
    private bool m_initialized;
    private object m_sync = new object();
    private Func<T> m_factory;

    public LazyInit(Func<T> factory) { m_factory = factory; }

    public T Value
    {
        get
        {
            lock (m_sync)
            {
                if (!m_initialized)
                {
                    m_value = m_factory();
                    m_initialized = true;
                }
            }
            return m_value;
        }
    }
}
```

Briefly, the data structure consists of four fields: the value that is lazy initialized (`m_value`), a flag specifying whether initialization has occurred (`m_initialized`), a synchronization object used for locking (`m_sync`), and a delegate that, when invoked, lazily initializes the object in question. Inside the `Value` accessor, we immediately acquire the lock and if the object hasn’t been initialized, we invoke the factory method, save its value, and set the initialization flag. We then return the value that got created.

Now the `Singleton` data structure above could be written as such.

```
class Singleton
{
    private static LazyInit<Singleton> s_inst =
        new LazyInit<Singleton>(() => new Singleton());

    public static Singleton Instance
```

```
{  
    get { return s_inst.Value; }  
}  
  
...  
}
```

All those examples of lazily initialized events, for example, can now simply be replaced with:

```
new LazyInit<EventWaitHandle>(() => new ManualResetEvent(false))
```

This attempt is correct. All initialization happens inside a lock, so there are no tricky memory model issues to consider. We used a reference type, but, in this particular example, `LazyInit<T>` could have been a value type to avoid the overhead of allocating another heap object. In many cases, lazy initialization is used to defer expensive resource allocation, which usually dwarfs the cost of having an extra object around.

The simplicity of this approach is also its downfall. Since synchronization is technically only needed while the value is initially created, it's a shame we're taking the lock each time the value is subsequently accessed. The popular solution to this problem is the double-checked locking pattern. A check is first made outside of the lock to see whether the value was initialized yet; if it was, it can be retrieved with no synchronization; if it wasn't, the lock can be entered and the value initialized. The subtle aspect to this pattern is that another check is done inside the lock to ensure another thread didn't concurrently initialize the value.

```
class LazyInit<T> where T : class  
{  
    private volatile T m_value;  
    private object m_sync = new object();  
    private Func<T> m_factory;  
  
    public LazyInit(Func<T> factory) { m_factory = factory; }  
  
    public T Value  
    {  
        get  
        {  
            if (m_value == null)  
            {  
                lock (m_sync)
```

```
        {
            if (_m_value == null)
                _m_value = _m_factory();
        }
    }
    return _m_value;
}
}
```

Contrary to popular belief, this *does* work in .NET 2.0+. (The popular misconceptions are largely due to other popular languages—namely, VC++—not guaranteeing that the pattern will work across platforms.) For it to be absolutely correct, you must mark the `_m_value` field `volatile`. The reason this needs to be `volatile` is similar to the reason that double-checked locking doesn't work on some non-.NET platforms.

The `_m_factory` delegate probably refers to a method that creates, initializes, and returns a new object, that is, as with the above example where it is `new Singleton()`. Fields of the newly constructed object will be initialized in the process. And this is the reason this pattern doesn't work on many memory models: on platforms where stores may be reordered, the write of the newly allocated object's reference to `_m_value` could happen *before* the writes to its fields. A caller seeing that `_m_value` is nonnull (and hence initialized) may proceed to using the object, and yet its fields will contain garbage, uninitialized data. The .NET 2.0 memory model disallows store reordering.

But a similar issue lurks with loads of the fields. Because all of the processors mentioned above, in addition to the .NET memory model, allow load-to-load reordering in some circumstances, the load of `_m_value` could move *after* the load of the object's fields. The effect would be similar and marking `_m_value` as `volatile` prevents it. Marking the object's fields as `volatile` is *not* necessary because the read of the value is an acquire fence and prevents the subsequent loads from moving before, no matter whether they are `volatile` or not. This might seem ridiculous to some: how could a field be read before a reference to the object itself? This appears to violate data dependence, but it doesn't: some newer processors (like IA64) employ *value speculation* and will execute loads ahead of time. If the processor happens to guess the correct value of the reference and field as it was before the reference was written, the speculative read could retire and create a problem. This kind of

reordering is quite rare and may never happen in practice, but nevertheless it is a problem.

If you're watching closely, you probably noticed we restricted `T` to a reference type. That's done so we can use `m_value` being `null` instead of a separate initialization flag to determine whether we must initialize the value. We can extend the above example to accommodate value types by introducing an initialization variable, similar to the opening code.

```
class LazyInit<T>
{
    private T m_value;
    private volatile bool m_initialized;
    private object m_sync = new object();
    private Func<T> m_factory;

    public LazyInit(Func<T> factory) { m_factory = factory; }

    public T Value
    {
        get
        {
            if (!m_initialized)
            {
                lock (m_sync)
                {
                    if (!m_initialized)
                    {
                        m_value = m_factory();
                        m_initialized = true;
                    }
                }
            }
            return m_value;
        }
    }
}
```

We must be careful because we need to ensure that loads of the initialization flag never get reordered with respect to the value itself, in addition to any fields being initialized. This is done by annotating `m_initialized` as `volatile`. This also works around another tricky issue: we can't mark non-reference and open-ended variables of type `T` with the `volatile` modifier; having the `m_initialized` field `volatile` avoids the reordering problems just mentioned.

A Slight Variant: Allowing Multiple Instances. The previous example prevents multiple invocations of the `m_factory` delegate by using a lock. Often this is what you want, particularly if the object that is being lazily allocated is expensive to create and destroy. But this is strictly stronger than necessary to prevent multiple objects from being published. It also disqualifies the `LazyInit<T>` primitive from being nonblocking because, under certain circumstances, threads may block, specifically, if they all race to initialize the object simultaneously.

We can make a slight change to the above algorithm to enable this relaxation and to provide our first example of a truly wait free algorithm.

```
class LazyInitRelaxedRef<T> where T : class
{
    private volatile T m_value;
    private Func<T> m_factory;

    public LazyInit(Func<T> factory) { m_factory = factory; }

    public T Value
    {
        Get
        {
            if (m_value == null)
                Interlocked.CompareExchange(
                    ref m_value, m_factory(), null);
            return m_value;
        }
    }
}
```

The code has become simpler. If `m_value` is seen to be `null`, a thread will attempt to perform an `Interlocked.CompareExchange`: if `m_value` is still `null` after creating a new object by invoking `m_factory`, this new object will be published. No matter whether this succeeds or not, we always return `m_value`. This is actually wait free because a thread will complete the operation in one step, no matter if it succeeds or not. No single thread can prevent progress of another in the system.

If the `Interlocked.CompareExchange` fails, we will have created a garbage object. Given that lazy initialization is typically meant for expensive object creation, it is likely that such objects will implement `IDisposable`; in

such case, it's likely advantageous to call `Dispose` on this object immediately instead of just letting it go. This complicates the example slightly.

```
class LazyInitRelaxedRef<T> where T : class
{
...
    if (m_value == null)
    {
        T obj = m_factory();
        if (Interlocked.CompareExchange(
            ref m_value, obj, null) != null &&
            obj is IDisposable)
            ((IDisposable)obj).Dispose();
    }
    return m_value;
...
}
```

Notice again that we've constrained `T` to be a reference type. The reason is that we can't always publish the whole structure with a single `Interlocked.CompareExchange`. To facilitate this, we need to wrap the value type in a heap allocated object.

```
class LazyInitRelaxedVal<T> where T : struct
{
    class Boxed
    {
        internal T m_value;
        internal Boxed(T value) { m_value = value; }
    }

    private volatile Boxed m_value;
    private Func<T> m_factory;

    public LazyInit(Func<T> factory) { m_factory = factory; }

    public T Value
    {
        get
        {
            if (m_value == null)
                Interlocked.CompareExchange(
                    ref m_value, new Boxed(m_factory()), null);
            return m_value;
        }
    }
}
```

Lazy Initialization in VC++

Because VC++ doesn't strengthen the model of the underlying machine, it can be problematic to write portable lazy initialization in native code. Technically speaking, you can do it, as we'll see. But we will conclude this section by looking at new Windows Vista APIs that allow you to write portable lazy initialization code without needing to worry about the memory model. The code is more verbose, albeit the various portability concerns are handled by the OS for you: which you prefer is purely a tradeoff in complexity versus flexibility.

Double-Checked Locking: The Basic Pattern. Many of the above ideas apply equally to native code. You have to be very careful, however, in your placement of `volatile` keywords and memory fences to prevent the plethora of reordering problems on all platforms. Because VC++ `volatiles` don't imply fences in the emitted assembly code at the processor level, you need to add some fences in precarious places.

```
template<typename T>
class LazyInit {
    volatile T * m_pValue;
    CRITICAL_SECTION m_crst;
    T (m_pFactory *)();

public:
    LazyInit(T (pFactory *)())
    {
        m_pValue = NULL;
        m_pFactory = pFactory;
        InitializeCriticalSection(&m_crst);
    }

    ~LazyInit()
    {
        // Possibly delete/cleanup m_pValue.
        DeleteCriticalSection(&m_crst);
    }

    T getValue()
    {
        if (!m_pValue)
        {
            EnterCriticalSection(&m_crst);
            if (!m_pValue)
            {
                T pValue = m_pFactory();
```

```
        _WriteBarrier();
        m_pValue = pValue;
    }
    LeaveCriticalSection(&m_crst);
}
_ReadBarrier();
return m_value;
}
};
```

This looks a lot like the C# version earlier, except for two interesting fences. A `_WriteBarrier` is found after instantiating the object, but before writing a pointer to it in the `m_pValue` field. That's required to ensure that writes in the initialization of the object never get delayed past the write to `m_pValue` itself. As noted earlier, the .NET memory model disallows such movement; but VC++ does not, unless explicit fences are used. Similarly, we need a `_ReadBarrier` just before returning `m_value` so that loads after the call to `getValue` are not reordered to occur before the call. This is surprisingly needed for processors like IA64 that do pointer and value speculation.

It's unfortunate that we need this last barrier because the only dangerous period of time is immediately after construction. Because there's no fixed length on this window of time, it is generally not possible to remove the barrier. However, I will also point out that *neither* fence is required on X86, Intel64, and AMD64 processors. It's unfortunate that weak processors like IA64 have muddied the waters, but if you are willing to write entirely processor specific code, you can consider emitting the fences or writing `#ifdef IA64` around them.

Windows Vista One-Time Initialization. The one-time initialization feature that was introduced in Windows Vista is a bit like the `LazyInit<T>` shown earlier in that you must create an instance of an `INIT_ONCE` and initialize it before it can be used. Initialization only prepares the data structure for subsequent use and doesn't associate a callback as the `LazyInit<T>` data structure above did.

```
VOID WINAPI InitOnceInitialize(PINIT_ONCE InitOnce);
```

There are two modes for one-time initialization, and they correspond exactly to those we looked at above. In one model, with the `InitOnceExecuteOnce` function, you are guaranteed that only one thread will perform

the initialization through the API using locks internally. The first model is the simplest to use and is where we will begin.

```
BOOL WINAPI InitOnceExecuteOnce(
    PINIT_ONCE InitOnce,
    PINIT_ONCE_FN InitFn,
    PVOID Parameter,
    LPVOID * Context
);
```

To retrieve the value, `InitOnceExecuteOnce` is called; it internally uses double-checked locking and will call the `InitFn` callback to initialize the value when needed, finally returning the value in the `Context` argument. This callback takes the form of an `InitOnceCallback` function pointer.

```
BOOL CALLBACK InitOnceCallback(
    PINIT_ONCE InitOnce,
    PVOID Parameter,
    PVOID * Context
);
```

The `Parameter` argument is an opaque value that is passed through from `InitOnceExecuteOnce` to the callback and can be used for pertinent initialization information. If the initialization callback returns FALSE, the call to `InitOnceExecuteOnce` will also return FALSE, indicating that the lazy initialization has failed.

Here is an example of a lazy initialized event class that uses this feature.

```
class LazyInitEvent {
    INIT_ONCE m_lazyEvent;

public:
    LazyInitEvent()
    {
        InitOnceInitialize(&m_lazyEvent);
    }

    BOOL initEvent(
        PINIT_ONCE InitOnce, PVOID Parameter, PVOID * lpContext)
    {
        *lpContext = CreateEvent(NULL, TRUE, TRUE, NULL);
        return *lpContext != NULL;
    }

    HANDLE getValue()
```

```
{  
    PVOID pHandle;  
    if (InitOnceExecuteOnce(  
        &m_lazyEvent, initEvent, NULL, &pHandle))  
    {  
        // Duplicate the HANDLE so that when the caller closes  
        // it the shared object doesn't go away.  
        HANDLE pRetVal;  
        DuplicateHandle(  
            GetCurrentProcess(),  
            reinterpret_cast<HANDLE>(pHandle),  
            GetCurrentProcess(),  
            &pRetVal,  
            NULL,  
            FALSE,  
            NULL);  
        return pRetVal;  
    }  
    return INVALID_HANDLE_VALUE;  
}  
};
```

Notice that we duplicate the HANDLE returned by the `InitOnceExecuteOnce` function to ensure that multiple references to the same event object can be given out and freely closed without de-allocating the shared instance. Notice that we don't have a destructor and, thus, never get around to freeing the event. The reason is subtle: if we were to get the HANDLE value by calling `InitOnceExecuteOnce` inside a destructor, we'd be forcing allocation of an event just so that we could close it. This is wasteful. In addition to allowing multiple initializations to race to publish a value (such as the lockless hand coded version earlier), the alternative `InitOnceBeginInitialize` function allows you to check the status of the initialization. We'll soon see how to use this to free the HANDLE without forcing allocation.

In the other model, with the `InitOnceBeginInitialize` and `InitOnceComplete` functions, multiple initialization callbacks may execute but only one will "win" and have its value published to the `INIT_ONCE` data structure.

```
BOOL WINAPI InitOnceBeginInitialize(  
    LPINIT_ONCE lpInitOnce,  
    DWORD dwFlags,  
    PBOOL fPending,  
    LPVOID * lpContext  
);  
BOOL WINAPI InitOnceComplete(  
    ...  
);
```

```
    LPINIT_ONCE lpInitOnce,
    DWORD dwFlags,
    LPVOID lpContext
);
```

This model can be used for both “asynchronous” initialization—that is, where many threads attempt to initialize the value at once—in addition to the ordinary “synchronous” initialization mentioned above, where Win32 ensures the callback executes only once. To specify asynchronous, you pass `INIT_ONCE_ASYNC` to the function. If this is not specified, other threads will be blocked on calling this until the first thread finishes initialization. You may also pass `INIT_ONCE_CHECK_ONLY` as a flag that indicates that the lazily initialized value should be retrieved without actually forcing initialization. If `InitOnceBeginInitialize` returns `TRUE`, the `fPending` output parameter tells you what to do. If `INIT_ONCE_CHECK_ONLY` was specified, the value tells you whether lazy initialization has occurred already, and the value will have been stored into `lpContext`. Otherwise, if `fPending` is `TRUE`, it means the calling thread must perform the initialization, and if it’s `FALSE`, the value is already initialized and will have been placed into `lpContext`.

If a thread is responsible for initializing the value, it then goes ahead after the call returns. Notice there is no callback involved. Once complete, it calls `InitOnceComplete` to supply the initialized value in the `lpContext` argument. If `INIT_ONCE_ASYNC` was passed to the begin initialization function, it must also be passed here in `dwFlags`. It is also imperative that failed initialization attempts signal the `INIT_ONCE` data structure through `InitOnceComplete` by passing `INIT_ONCE_INIT_FAILED`, otherwise with synchronous initialization threads could become deadlocked. If the `InitOnceComplete` function returns `FALSE`, it means that another thread raced and beat the calling thread (with asynchronous initialization) and that the caller must retrieve the value now available by calling `InitOnceBeginInitialize` with the `INIT_ONCE_CHECK_ONLY` flag.

Here is a version of the `LazyInitEvent` class above that uses asynchronous initialization.

```
class LazyInitEvent
{
    INIT_ONCE m_lazyEvent;
```

```
public:
    LazyInitEvent()
    {
        InitOnceInitialize(&m_lazyEvent);
    }

    ~LazyInitEvent()
    {
        BOOL fPending;
        HANDLE hEvent;
        if (InitOnceBeginInitialize(
            &m_event, INIT_CHECK_ONLY, &fPending,
            reinterpret_cast<PVOID>(&hEvent)) && fPending)
            CloseHandle(hEvent);
    }

    HANDLE getValue()
    {
        HANDLE hEvent;
        BOOL fPending;
        if (!InitOnceBeginInitialize(
            &m_lazyEvent, INIT_ONCE_ASYNC, &fPending,
            reinterpret_cast<PVOID>(&pHandle)))
            return INVALID_HANDLE_VALUE;

        if (fPending)
        {
            // We need to create an event and publish it.
            hEvent = CreateEvent(NULL, TRUE, TRUE, NULL);
            if (!InitOnceComplete(
                &m_lazyEvent, INIT_ONCE_ASYNC, hEvent)) {
                // We lost the race. Close our handle.
                CloseHandle(hEvent);
                InitOnceBeginInitialize(
                    &m_event, INIT_ONCE_CHECK_ONLY, &fPending,
                    reinterpret_cast<PVOID>(&hEvent));
                if (!fPending) return INVALID_HANDLE_VALUE;
            }
        }

        // Duplicate the HANDLE so that when the caller closes
        // it the shared object doesn't go away.
        HANDLE pRetVal;
        DuplicateHandle(
            GetCurrentProcess(),
            hEvent,
            GetCurrentProcess(),
            &pRetVal,
            NULL,
```

```

        FALSE,
        NULL);

    return pRetVal;
}
};

```

Notice that we're now able to write a destructor because we can specify INIT_ONCE_CHECK_ONLY to avoid forcing initialization of the event.

A Nonblocking Stack and the ABA Problem

There are several well-known nonblocking collections data structures, such as stacks, queues, priority queues, deques, sets, hashtables, and more. We'll take a closer look at some of these in Chapter 12, Parallel Containers. But as more of a case study—and because it's the simplest one by far—let's look at how a nonblocking stack is implemented. Although this sounds complicated, it's straightforward except for one tricky issue called the ABA problem. We can easily avoid the ABA problem in managed code, but not in VC++. Windows offers a so-called SList data structure that is nonblocking and has been written to avoid the ABA problem, making it simple to use from native code.

A Custom Nonblocking Stack

Let's start by looking at a custom written nonblocking stack in C#.

We will use a linked list for storing nodes. This is unfortunate for some reasons—such as requiring an O(N) operation to retrieve the count—but is the key point to enabling the nonblocking property. The head of the list represents the top of the stack, so pushes will replace the head with the newly enqueued node pointing to the old head, and pops will swap the head with the head's current next pointer. This algorithm is easy to implement in a non-blocking way because both pushing and popping boil down to a single compare-and-swap operation. Seeing this in practice can be quite illuminating.

```

class LockFreeStack<T>
{
    class Node
    {
        internal T m_value;
        internal volatile Node m_next;
    }
}

```

```
}

volatile Node m_head;

void Push(T value) { ... }
T Pop() { ... }
}
```

Let's look at the Push operation.

```
void Push(T value)
{
    Node n = new Node();
    n.m_value = value;

    Node h;
    do
    {
        h = m_head;
        n.m_next = h;
    }
    while (Interlocked.CompareExchange(ref m_head, n, h) != h);
}
```

You may need to look carefully at that code to convince yourself that it's right. We construct a new `Node` object to hold the value being pushed and immediately enter a `do-while` loop. Inside this loop we read the `m_head` field into a local variable `h`. We then set the new node's next pointer to `h`. Notice that although this value could be out-of-date right away, setting it is safe; because we've not yet made the new node `n` publicly visible yet, no other thread can possibly see this value. We then try to make it visible with an `Interlocked.CompareExchange`. We replace the current reference in `m_head` with the new node `n`, but only if the head we saw, `h`, is still there. If it fails, we go back and try again. The `m_head` variable is marked `volatile` to ensure we properly reread it during the next iteration of the loop.

The `Pop` operation works similarly.

```
T Pop()
{
    Node n;
    do
    {
        n = m_head;
        if (n == null) throw new Exception("stack empty");
    }
```

```
        }
        while (Interlocked.CompareExchange(ref m_head, n.m_next, n) != n);

        return n.m_value;
    }
```

We simply read the `m_head` variable into a local, `n`, and try to swap the `m_head` variable with `n`'s `m_next` reference. If this fails, we loop back and try again. Notice that we'd have a tricky issue to deal with if this were written in VC++. Specifically, another thread concurrently popping a node off the stack might try to free the memory associated with the node. If we accessed its `m_next` pointer, we'd have a problem: a null dereference and likely an ensuing AV.

This implementation is lock free but it isn't wait free. Whenever a thread fails, it's because another thread made forward progress (i.e., succeeded in its own operation). But we make no accommodation to prevent a particular thread being starved by other threads. In a real implementation, we'd also probably want to add some amount of spin-wait backoff when a thread fails to make forward progress. This would reduce contention on the shared variable and can make a big difference for very hot stacks on machines with many processors.

The ABA Problem

The ABA problem leads to CAS operations succeeding when they should have failed, rendering the algorithm shown (and many just like it) utterly broken. Although we didn't encounter it previously, due to our use of managed code, here are a couple of things could bring rise to the ABA problem.

- If we tried to pool and reuse nodes that have been popped off the stack, the same node objects could be involved in multiple concurrent operations. This might be an initially attractive way of avoiding extra allocations on the Push operation and garbage created on the Pop operation.
- If we write the above data structure in VC++, where node memory is freed and given back to a memory allocator, it can be concurrently reused.

The ABA problem stems from the fact that we use the pointer value of `m_head` to determine whether the stack has changed. But if nodes can be reused, it could be the case that after reading `m_head` as a certain value `X`, the node `X` could be concurrently popped off the stack, subsequently reused, and then pushed back on the top of the stack as `m_head`. A thread doing an interlocked compare-exchange would then find the value `X` in the location and the CAS would succeed, because it appears as if the stack never changed. Clearly this outcome is incorrect. The CAS should have failed. The list *did* change.

As a concrete example of why this can be a problem, imagine our stack has two nodes: `X` at the top, and `Y` just behind it. Say a thread tries to pop `X` off and gets as far as reading its `m_next` pointer into a local variable, seeing `Y`. But it doesn't get as far as executing the CAS, perhaps because it gets preempted by another thread—another thread, that pops `X` off and then `Y`, leaving the stack empty. Yet another thread comes along, pushes a new node, `Z`, on, and then (for whatever reason) it pushes `X` on again. If we pooled nodes, the object `X` might get reused time and time again, each time with a new value inside it. At this point, `X`'s `m_next` pointer will refer to `Z`. But when the first thread resumes and performs its CAS, the operation will succeed: it will place `Y` as the new head—even though `Y` is long gone—and `Z` will now go completely missing. This mysterious sequence of events is subtle enough to leave you frustrated and scratching your head.

Avoiding this problem typically requires additional state to be used in the CAS operation, such as a version number that is incremented upon each push and pop. In other words, instead of updating one value, we will update two at once: the pointer and a new integer version number. Implementing this either requires an extra layer of indirection, like using a separate object, or double CAS operations, such as a 64-bit CAS on a 32-bit machine or a 128-bit CAS on a 64-bit machine. Since the latter isn't always available on all architectures, this makes writing efficient and portable ABA safe data structures difficult. This situation won't happen in managed code (unless we explicitly pool nodes) because, unlike VC++, so long as a reference to an object is live, the memory will not be reused. This fact, coupled with integration of interlocked operations and the code that performs GCs, ensures ABA safety.

Win32 Singly Linked Lists (SLists)

The ABA problem is difficult and isn't immediately obvious. Instead of having to write your own ABA safety mechanisms, Win32 offers a lock free stack called an **interlocked singly-linked list** that uses the same algorithm explained before, but with embedded ABA safety. SLists are used pervasively throughout the Windows kernel itself.

SLists are represented with an instance of the **LIST_HEADER** data structure. To create an empty one, just allocate this memory somewhere, and call the initialization function.

```
void WINAPI InitializeSListHead(PSLIST_HEADER ListHead);
```

Entries take the form of **SLIST_ENTRY** data structures. Typically these will be embedded into other data structures as fields and are used for linking nodes together internally in the SList code. They also contain next pointers to other **SLIST_ENTRY** data structures. Although these pointers are managed by the SList implementation, you can freely follow them provided that you know they are in a good known state.

You can't actually manipulate the **LIST_HEADER** structure yourself, as its contents are managed by the OS and are subject to change from one architecture to the next. Once you have one, however, you can push and pop elements on and off the stack.

```
PSLIST_ENTRY WINAPI InterlockedPushEntrySList(
    PSLIST_HEADER ListHead,
    PSLIST_ENTRY ListEntry
);
PSLIST_ENTRY WINAPI InterlockedPopEntrySList(PSLIST_HEADER ListHead);
```

Both functions return a pointer to a **SLIST_ENTRY** data structure. In the case of pushing new elements, this is the old head of the list (which is now the head's next element) and is for informational purposes only. It will be **NULL** if the list was empty. In the case of popping, this is the return value of interest to you: the removed element. If it's a field embedded within a larger data structure, you'll have to perform whatever typecasts are necessary to get at the information you desire because entries contain no interesting user-mode state. Two other operations are available for SLists. You can clear the list and also compute a count of elements in the list.

```
PSLIST_ENTRY WINAPI InterlockedFlushSList(PSLIST_HEADER ListHead);
USHORT WINAPI QueryDepthSList(PSLIST_HEADER ListHead);
```

When clearing the list, you are given a pointer to the old head node. You may then traverse the list, for example, if you need to process the elements or free their associated memory.

As an example of usage, here is some code that uses a general purpose templatized struct to hold the data, initializes a new SList, pushes 10 elements onto the list, pops off half of them, and flushes the remaining contents of the list.

```
template <class T>
struct DataItem
{
    SLIST_ENTRY m_listEntry;
    T m_value;
};

// Elsewhere...

// Declare and initialize the list head.
SLIST_HEADER listHead;
InitializeSListHead(&listHead);

// Push 10 items onto the stack.
for (int i = 0; i < 10; i++)
{
    DataItem<int> * d = (DataItem<int> *)malloc(sizeof(DataItem<int>));
    d->m_value = i;
    InterlockedPushEntrySList(&listHead, &d->m_listEntry);
}

// Pop 5 items off the stack.
for (int i = 0; i < 5; i++)
{
    DataItem<int> * d = (DataItem<int> *)
        InterlockedPopEntrySList(&listHead);
    assert(d && d->m_value == (10 - i - 1));
    free(d);
}

// Now flush the remaining contents of the list.
DataItem<int> * d = (DataItem<int> *)InterlockedFlushSList(&listHead);
while (d)
{
    DataItem<int> * next = (DataItem<int> *)d->m_listEntry.Next;
```

```

        assert(d);
        free(d);
        d = next;
    }

    // We expect the list is empty by now.
    assert(InterlockedPopEntrySList(&listHead) == NULL);
}

```

Consuming Win32 SLists from managed code with P/Invokes is difficult because the unmanaged `SLIST_HEADER` and `SLIST_ENTRY` data structures contain pointers to other entries. The CLR's garbage collector doesn't know about these unless you perform special pinning operations and/or use GC-handles to track the references, both of which can be incredibly expensive. It's simpler to use the algorithm shown above when you are in .NET.

Dekker's Algorithm Revisited

For fun, let's look at an antipattern by going back to the 2-CPU example of Dekker's algorithm for mutual exclusion from Chapter 2, Synchronization and Time.

```

static bool[] flags = new bool[2];
static int turn = 0;

void EnterCriticalSection(int i) // i will only ever be 0 or 1
{
    int j = 1 - i;           // the other thread's index
    flags[i] = true;        // note our interest
    while (flags[j])        // wait until the other is not interested
    {
        if (turn == j)      // not our turn, we must back off and wait
        {
            flags[i] = false;
            while (turn == j) /* busy wait */;
            flags[i] = true;
        }
    }
}

void LeaveCriticalSection(int i)
{
    turn = 1 - i;           // give away the turn
    flags[i] = false;        // and exit the region
}

```

A common problem with this code is that the inner loop in `EnterCriticalSection`, which spins on `turn` changing, can be considered loop invariant. This means the compiler could hoist the read outside of the loop, leading to a thread busy spinning forever. Marking `turn` as `volatile` is sufficient to avoid this problem.

Similarly, a smart compiler may deduce that `i` could never equal `1 - i` and, therefore, the `flags` element read in the loop is never written to inside the loop body. Once again, the compiler may hoist the read outside of the loop and cause an infinite spinning situation. So we need to mark `flags` as `volatile` too.

Notice some other issues if we weren't to mark things as `volatile`. The write of `false` to `flags[i]`, just before spinning on `turn`, could move *after* the reads and be coalesced with the write of `true` to `flags[i]`. The result would be that we never give away our flag, causing our partner thread to spin forever waiting to see our flag become `false`.

A more fundamental problem is that, without `volatiles`, the fast-path of `EnterCriticalSection` causes no fence. Imagining the caller loads a variable immediately after entering the region, this load could be moved before the write to `flags[i]` and before the read of `flags[1 - i]`, since stores can pass loads. This has the effect of removing mutual exclusion: the variables read inside the critical region could be changing concurrently out from underneath us, which could be disastrous.

Where Are We?

This chapter covered a lot. We began by reviewing instruction reordering and its subtle implications to concurrent programs. Processors and some programming models (e.g., in the case of .NET) make strong guarantees about which operations can freely reorder, making it at least feasible for real human beings to program in a lock free way. We then saw the basic mechanisms that can be used for atomic memory operations and how fences limit processors and compilers from reordering certain instructions. Finally, we concluded with some examples of safe lock free techniques. They were not exhaustive, but at least provide a useful starting point.

Up next: we'll take a closer look at the types of hazards concurrency can cause.

FURTHER READING

- AMD x86-64 Architecture Programmer's Manual Volumes 1–5* (Advanced Micro Devices, 2002).
- C. Brumme. Memory Model. Weblog article, <http://blogs.msdn.com/cbrumme/archive/2003/05/17/51445.aspx> (2003).
- M. Chynoweth, M. R. Lee. Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures. Intel Software Network, <http://softwarecommunity.intel.com/articles/eng/2807.htm> (2003).
- J. Duffy. Revisited: Broken Variants on Double Checked Locking. Weblog article, <http://www.bluebytesoftware.com/blog/2007/02/19/RevisitedBrokenVariantsOnDoubleCheckedLocking.aspx> (2007).
- J. Duffy. Simple SSE Loop Vectorization from Managed Code. Weblog article, <http://www.bluebytesoftware.com/blog/2007/05/30/SimpleSSELoopVectorizationFromManagedCode.aspx> (2007).
- J. Duffy. 9 Reusable Parallel Data Structures and Algorithms. *MSDN Magazine* (2007).
- J. Duffy. A Lazy Initialization Primitive for .NET. Weblog article, <http://www.bluebytesoftware.com/blog/2007/06/09/ALazyInitializationPrimitiveForNET.aspx> (2007).
- K. S. Gatlin. Windows Data Alignment on IPF, x86, and x64. *MSDN* article, <http://msdn2.microsoft.com/en-us/library/aa290049.aspx> (2006).
- K. Ghrachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. In *Computer Systems Laboratory*, Technical Report CSL-TR-95-685 (Stanford University, 1995).
- Intel Itanium Architecture Software Developer's Manual: Instruction Set Reference, Volume 3* (Intel Corporation, 2002).
- Intel Itanium Architecture Software Developer's Manual: System Architecture, Volume 3* (Intel Corporation, 2002).
- Intel 64 Architecture Memory Ordering White Paper. <http://www.intel.com/products/processor/manuals/318147.pdf> (Intel Corporation, 2007).
- D. Lea. The JSR-133 Cookbook for Compiler Writers. <http://g.oswego.edu/dl/jmm/cookbook>.
- M. Maged. *ABA Prevention Using Single-Word Instructions*. IBM Research Report RC23089 (W0401-136) (2004).

- M. Maged. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 6. (2004).
- J. Manson, W. Pugh, S. V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (2005).
- V. Morrison. Concurrency: Understand the Impact of Low-Lock Techniques in Multithreaded Apps. *MSDN Magazine* (October 2005).
- R. Saccone, A. Taskov. Concurrency: Synchronization Primitives New to Windows Vista. *MSDN Magazine* (2007).
- D. Schmidt, T. Harrison. Double-Checked Locking: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In *3rd Annual Pattern Languages of Program Design* (1996).
- H. Sutter. Prism: A Principle-Based Sequential Memory Model for Microsoft Native Code Platforms. *Working Draft Proposal 0.9.3* (2006).

11

Concurrency Hazards

THROUGHOUT THE COURSE of this book, we've seen many platform services that enable concurrent programming on Windows. But as we also saw in Chapter 2, Synchronization and Time, the addition of concurrency to a program comes with many additional concerns. Concurrency is a double-edged sword: it can be used to do great things—such as creating software that scales as newer hardware with more processors is adopted, paving the way for more sophisticated software capabilities, or ensuring responsiveness and compelling user experiences in GUI programs—but if done incorrectly, it can lead to significant trouble.

Now that we've finished reviewing the fundamental mechanisms used to build concurrent software, we'll turn to some common problems you're apt to encounter. We call these things "hazards," to emphasize their negative effect and the ease with which you might accidentally stumble upon them. For sake of discussion, we'll put hazards into one of two categories.

- **Correctness hazards.** Cause programs to produce incorrect results.
- **Liveness hazards.** Cause programs to stop producing results, at least temporarily (if not permanently).

Both categories are bad but for different reasons. Correctness hazards are notoriously very difficult to uncover because of the nondeterministic nature of concurrency. Because a concurrent program takes different courses of

action each time it is run, concurrency bugs often depend on subtle runtime and time-sensitive interactions between threads. This makes such hazards hard to debug and to test. Moreover, when a hazard manifests, it may not be immediately obvious. The result could be silent corruption of important data, and it may go unnoticed for a long time. Liveness hazards are often more obvious when they occur because a program hangs and stops responding to external stimulus, but they are also often difficult to provoke. They don't always lead to data corruption—unless an impatient user kills the program in response—but can cause poor user experiences (for the client) and inefficient use of expensive hardware (on the server).

As we explore the various kinds of concurrency hazards, we'll also look at practical ways to avoid or deal with them. Eliminating hazards by construction is an important goal for which all engineers building concurrent programs should strive. By the time the code has been written, the possibility of these errors should be ruled out. This is a lofty goal, but the fact remains: attempting to find such problems after software has been written is always substantially more time consuming. Some structured approaches to your software design, development, and engineering practices can go a long way. More than anything else, however, a deep fundamental understanding of concurrency is paramount.

Correctness Hazards

Let's begin by examining various kinds of correctness hazards. This category is full of data race problems of different sorts, but also includes subtleties around lock recursion and reentrancy. We'll also see some unique problems that arise due to locks and application shutdown. This includes the possibility of orphaning locks indefinitely.

Data Races

All imperative programs contain fundamental assumptions about state, control flow, and the intertwined relationship between the two. This relationship is not always explicitly called out, but, should you violate one of the assumptions, your code is apt to do strange things. For example, if we have just written the value 5 to some memory location x , can subsequent

lines of code safely assume it will continue containing the value 5 as `x` is reread over and over again?

```
SomeType myObj = ...;
myObj.x = 5;
...
int a = myObj.x; // Still 5?
...
int b = myObj.x; // What about now?
```

If multiple threads can access `myObj` at once, this code is apt to break if it assumes that both `a` and `b` will contain the value 5. Another thread could write to `x` in between the execution of the two separate reads. Preventing this situation requires some concurrency safety: isolation (private state), data synchronization, or immutability. But what if you forget to add the necessary concurrency safety? Or what if you do it incorrectly? We won't dwell too long on this particular problem. We already discussed data races at great length in Chapter 2, Synchronization and Time, so you should know that doing these things causes your program to crash, hang, or corrupt important application and system state.

Many assumptions commonly made by sequentially oriented software are quickly invalidated by concurrency due to unexpected interactions between many threads running different parts of your program simultaneously. Another way of explaining this is in terms of **invariants**. All algorithms and data structures have invariants, even if they aren't explicitly called out. Invariants are important to be conscientious of when programming because, when broken, the surrounding program logic behaves unexpectedly. Understanding and documenting invariants is tremendously helpful in building correct and robust concurrent systems.

The term "invariant" sounds overly abstract. Here are a few concrete examples.

- Methods have **preconditions** that represent conditions that the method assumes to be true in order to function correctly. Sometimes preconditions pertain to arguments to a method, in which case they are typically checked by argument validation logic. Other times, preconditions pertain to surrounding state and the implementation may assert (or just assume) that they are true.

- Similarly, methods have **postconditions** that specify the state of the returned and surrounding state after the method has finished executing.
- **Object invariants** apply to a single object and describe expected legal states in which the object may be. For example, we might assume that the current index for a list backed by an array is always within legal range, that is, points to a valid index in the array. Were this ever to be untrue, the object's methods would probably not work correctly, that is, method preconditions often include the object's invariants.
- **Control flow invariants** are like object invariants, but are more ad hoc and local. For example, once we've exited a loop, we might expect some set of conditions to hold. Or, as in our `x = 5` example above, we might assume some earlier assignments still hold true.

Some systems even allow checking of invariants in a structured way. For example, the language Eiffel (see Further Reading, Meyer) is well known for its first class support, and research systems such as Spec# from Microsoft Research (see Further Reading, Barnett, Leino, Schulte) extend existing imperative languages (in this case, C#) with similar support for checking invariants. Use of such systems is not widespread on Windows, so most invariants take the form of asserts sprinkled throughout your code base.

The relationship between invariants and race conditions is fundamental. If your program can reach a state in which an invariant doesn't hold for state that is visible among multiple threads, your program has a race condition. Broken invariants cannot be sidestepped because many logical operations entail multiple physical steps to complete. In between steps, state may be left inconsistent. If you can write your data structures so invariants hold at each atomic state update, you've built one capable of lock freedom and might use this to your advantage when it comes to building scalable code. But for most cases, the practical implication is that state must be protected by synchronization or be kept isolated for the duration of said broken invariants. When locking is involved, we often say that invariants must hold at lock entry and exit boundaries.

Since we already reviewed the basics of synchronization at the start of this book, let's look at some of the other variants on the core idea. These include races caused by inconsistent use of locking in your program and not holding a lock long enough; we'll also see that certain kinds of **benign race conditions** are safe, can be useful, and do not result in incorrect program behavior.

Inconsistent Synchronization

Assume you're using synchronization to ensure no threads see an object as it is undergoing a state transition. It's not good enough that access to this object is performed under the protection of just *any* kind of synchronization. You need to ensure that all threads access the object do so under the *same* kind of synchronization. In other words, if you access some object *x* under lock *a* in one part of the program, and under lock *b* in another, those two parts of the program will not run mutually exclusive to one another. This might be obvious, but this mistake is easy to make. Often the results are just as bad as not having locked at all.

For example, consider this program snippet.

```
static Data s_x = ...;
static Data s_y = ...;
static object s_lockX = new object();
static object s_lockY = new object();

void f()
{
    lock (s_lockX)
    {
        s_x.f1++;
        s_x.f2++;
    }
}

void g()
{
    lock (s_lockY)
    {
        s_y = new Data(s_x); // Reads state (unsafely) from s_x.
    }
}
```

Now imagine that *f* and *g* are called on separate threads simultaneously. Can you see the problem? Even though both *f* and *g* execute under critical

regions, they do so with different monitor objects: `s_lockX` and `s_lockY`. The result is that both methods run fully concurrent with one another, meaning that `g` may read state updates being made to `s_x` by method `f` before they are complete. Even if all `g` is doing is reading from the object, there could be some invariant protecting the relationship between fields `f1` and `f2` of `Data` instances. And observing the broken invariants could lead to `g` crashing.

One of the most widely known dynamic race condition detection algorithms, called the lockset algorithm, popularized by several research systems such as Eraser (see Further Reading, Savage, Burrows, Nelson, Sobalvarro) and RaceTrack (see Further Reading, Yu, Rodeheffer, Chen) looks for these kinds of inconsistent data protection races. They even try to determine when a race is benign (i.e., all shared accesses are reads) or a potential disaster. An in-depth analysis of the algorithm itself is outside of the scope of this book, though interested readers might want to read more about it. The basic idea is as follows: the system monitors all critical regions in the program and which memory locations are accessed under the protection of these critical regions during execution of the program. The algorithm uses this information to continuously refine its guess as to which locks are candidates for protecting particular memory locations. It does so by taking the intersection of all locks held by a thread whenever a particular location was accessed. In our above example, if one thread executed `f` first, the candidate set is `{ s_lockX }`; when `g` runs, it also gets a candidate set. This set is `{ s_lockY }`, which, when intersected with the previous set `{ s_lockX }` is the empty set. The algorithm would thus (correctly) determine that there's a bug in the program shown.

There have been other recent approaches to solving this problem, including static race condition detection. For example, Abadi, et. al (see Further Reading) proposed language extensions to associate locks with fields and to check that whenever a particular field was accessed the associated lock was held by the current thread. Neither dynamic nor static race condition detection is broadly available in tools on the Windows platform today.

Composite Actions: Failing to Hold for Long Enough

A classic tradeoff when it comes to synchronization is critical region granularity. There is a constant tension between **fine granularity**—which generally gives better scalability, worse single-threaded performance (due to more lock

acquisitions), and results in far subtler and deadlock prone code—and **coarse granularity**—which generally gives superior single threaded performance, errs on the side of simplicity and correctness, but sacrifices scalability. But the tension to make critical sections as fine as possible can sometimes lead to accidentally releasing them too soon. This can expose broken invariants to other threads.

It is imperative that critical regions span the entire sequence of operations that make up some larger composite action. We've already covered serializability and linearizability, where some program action comprised of multiple steps is meant to appear as an atomic, indivisible action. For this to be achieved, the entire action must be wrapped in a critical region such that when it is released all invariants hold. The tension between performance and scaling can lead programmers to overtighten the granularity of a lock or to sneak in a few reads without using synchronization, thus introducing a general race condition.

As an example of where an overly fine-grained lock can break your program, imagine we are using a lock to protect access to a simple linked list. We want to remove the head node. This entails multiple synchronization sensitive reads and writes: first, we must read the head node; then we have to read the head's current next node; and, finally, we must store a reference to the old head's next node to the head variable. That's two reads and a single write; if we don't protect all of them by the same critical region, another thread could sneak in and change the data, causing us trouble.

Here's an incorrectly synchronized version of this algorithm.

```
class LinkedStack<T>
{
    class ListNode
    {
        internal T m_value;
        internal ListNode m_next;
    }

    private object m_lock = new object();
    private ListNode m_head = null;

    public T Pop()
    {
        // Avoid synchronization if the list is empty.
        ListNode currHead = m_head; // Read the head once.
```

```

    if (currHead == null)
        throw new Exception(...);

    // S0

    // Now that we know it's non-null, pop the head.
    lock (m_lock) {
        m_head = currHead.m_next;
    }

    return currHead.m_value;
}

...
}

```

This code is trying to be (overly) clever by reading `m_head` only once into a local variable `currHead`. This ensures we avoid synchronization when the list is empty. Another thread could add a new node as soon as we've done this check, but this would be a problem even if we took a lock. But there's a serious problem with this code. Do you see it?

Imagine that some thread `t1` reads `m_head` into `currHead`, sees it as non-null, and advances towards the critical region (`lock` statement). There is a window time between the check and when the critical region is entered. During this window, called out by `S0` above (even if `S0` consists of no program statements whatsoever), another thread `t2` can also call `Pop`, read `m_head` into `currHead`, also see it as non-null, and pop off the head. This is the same item that `t1` is about to pop. As soon as `t1` resumes and proceeds to its critical region, it will set `m_head` to the old head's `m_next` field. This will be incorrect and would have the effect of returning the same object more than once and possibly a whole chain of them if many threads popped elements during `S0`. Moreover, if other threads pushed new elements, they may be completely overwritten and lost. In C++, the effects could include an AV if nodes are freed as they are removed, since we'd try to access the `m_next` field of a freed object.

The simplest solution to this is straightforward: we take the lock around the whole operation. Technically, we can retain our unsynchronized check up front to improve the empty list case. But, this is a good example of premature cleverness, and the motivation for this optimization is questionable: it

isn't worthwhile at all to optimize synchronization for an "error" case that is not expected to occur frequently.

Here is the simpler, corrected Pop method instead.

```
...
public T Pop()
{
    lock (m_lock)
    {
        ListNode currHead = m_head; // Read the head once.
        if (currHead == null)
            throw new Exception(...);

        // S0

        // Now that we know it's non-null, pop the head.
        m_head = currHead.m_next;
    }

    return currHead.m_value;
}
...
```

Alternatively, we could have done two checks: one outside of the lock and one inside of the lock (before performing the pop).

Sometimes the motivation for breaking an operation into multiple lock acquires is to avoid blocking other threads while a compute or I/O intensive operation executes. If this is the case, it's better to refactor code so that the operation occurs outside of the lock. This can sometimes be a challenge. If it's not possible, optimistic concurrency can sometimes be used. In the original code sample, say we had to do some lengthy operation at S0 that was based on the shared data we read from inside the lock. If we associate a version number with the list, which is incremented each time a thread modifies the list and if we validate it didn't change once we reacquire the lock, we can know whether atomicity has been preserved. If the number has changed, we must throw away any calculations and start back at the beginning.

Benign Data Races

Not all access to shared data needs to happen with heavyweight synchronization. While unsynchronized access to shared data is always a data race,

some races are benign: that is, the program has been written to tolerate the race condition, and so these races are completely harmless. The reason for this was already reviewed in Chapter 10, Memory Models and Lock Freedom: individual reads and writes of word sized memory locations are always atomic.

(As an aside, benign races aren't always completely harmless: unsynchronized access to shared data is often an indication of premature cleverness and should be cause for concern when you run across it. Developers who inherit and must maintain this code might be tempted to add additional (unsafe) accesses surrounding it because they may assume some higher level synchronization has been established. Benign races can be used but only when done carefully.)

As a very simple illustration of where a benign race might be used, imagine that we have code that spawns N threads to do some work in parallel. Each task will search for some item in a collection. The collection's contents aren't sorted, so we can't use a binary search. The first thread to find a matching item can return, and then all other threads can stop searching. One solution is to have all threads synchronize with one another to check whether any of the other tasks have finished, but this would be costly. We might amortize the cost of synchronization by doing it only every so often, reducing the responsiveness once the item has been found, but improving the performance of the algorithm. But this is heavier weight than necessary.

We can take a completely different approach. Instead of using synchronization, we can use a single shared variable: any thread can atomically write the value `true` to it. Multiple threads may write it more than once, but this is OK because they write the same value. All other threads read from it continuously to notice approximately when the value changes to `true`. The variable changing to `true` is the cue to quit the search. There's no need for a critical region; the threads will remain correct without it and will perform significantly better.

```
static volatile bool s_finished = false; // Shared among tasks.

... some code elsewhere calls Find on disjoint data across N threads ...

int Find<T>(T[] data, T value, int myStartIdx, int myEndIdx)
{
```

```
// Each of the N threads do this:  
for (int i = myStartIdx; i < myEndIdx; i++)  
{  
    if (s_finished)          // Did somebody else find it?  
        return -1;           // OK: voluntarily quit.  
  
    if (Object.Equals(data[i], itemToLookFor)) // Did I find it?  
    {  
        s_finished = true; // Notify others.  
        return i;           // And return the value found.  
    }  
}  
}
```

This speculative search pattern is common in parallel programs and will be explored further in Chapter 13, Data and Task Parallelism. Many concurrent calls to `Find` may return a match. That's because just as one thread reads `s_finished` as `false`, another one could set it to `true`. At this point, the thread will have already moved on to checking for equality and potentially setting `s_finished` to `true` (overwriting the other thread) and returning its own item. More complicated schemes are possible and would prevent or tolerate this. But we have made the simplifying assumption that finding multiple is alright.

There are quite a few cases in which unsynchronized access such as this is safe. But, in general, any case should be well documented and scrutinized. It's very easy to mistakenly convince yourself that a data race is benign when, in reality, under some obscure timing, it isn't. Particularly due to memory reordering, you must tread with extreme caution. For example, do you know why the example above uses the `volatile` modifier for the `s_finished` variable? And is it strictly necessary? Knowing this requires a deep understanding of memory models and instruction reordering, as explained in the previous chapter.

Recursion and Reentrancy

Recursion and reentrancy are closely related and of interest when considering critical regions. Roughly speaking, they can be defined as follows.

- **Recursion** is a basic computer science notion, wherein a function calls itself. Each recursive call gets its own stack frame with dedicated

arguments and locals. Some algorithms are more easily expressed using recursion rather than iteration involving loops. Functional programs make heavy use of recursion, sometimes as the only kind of repeat control structure available.

- **Reentrancy** is a little more obscure. A reentrant method is one that could be interrupted at any point in favor of other code running on the same thread, possibly resulting in the same method being invoked again. This looks like recursion, but is not initiated by the method itself and is, thus, more error prone. It is more environmental than algorithmic. Reentrancy is often more pervasive in embedded systems and low-level code such as device drivers. As a simple example of user-mode reentrancy, consider APCs that may run whenever a thread does an alertable wait. As another example, both native and .NET can dispatch COM cross-apartment and GUI event handler calls as a result of pumping the message queue.

The two are related because a so-called **recursive lock** allows acquires due to recursion. But such locks often cannot differentiate between recursion and reentrancy. And so, when reentrancy occurs for a method containing a critical region, recursive locks allow reentrant acquisitions by the same thread, even though the reentrant work being performed is often logically unrelated. This can cause some surprises, as we will see later.

As noted in earlier chapters, standard synchronization mechanisms—such as Win32 critical sections and CLR monitors—support recursive acquires. If the thread holding a lock tries to acquire it again, the attempt will succeed. The implementation of these primitives increments an internal recursion counter associated with the lock; each acquisition must be paired with a release, and once the recursion counter drops to 0, only then is the lock made available to other threads. Recall from previous chapters that some locks, such as the Win32 and .NET Framework “slim” reader/writer locks, disallow recursive acquires by default.

Generally speaking, people like recursive acquires because it allows them to build larger composite atomic actions out of smaller atomic actions without having to change any code: just acquire the lock surrounding the entire composite action and forget about the smaller actions that will

(redundantly) reacquire the lock. This is most popular in higher level, object oriented application programming versus systems level programming.

As an illustration, we already have a list class with a synchronized Add method, and we want to create an atomic AddTwo method. Rather than duplicating code, we can reuse the existing Add implementation.

```
class MyList<T>
{
    private object m_lock = new object();
    private List<T> m_list = new List<T>();
    ...

    public void Add(T obj)
    {
        lock (m_lock)
        {
            m_list.Add(obj);
        }
    }

    public void AddTwo(T obj1, T obj2)
    {
        lock (m_lock)
        {
            Add(obj1);
            Add(obj2);
        }
    }
}
```

If recursion were not available, or we wanted to avoid using it, we'd need to build a separate AddNoLock method that assumes the lock is already held rather than trying to reacquire it. Both Add and AddTwo would then have to acquire the lock first, and then call AddNoLock.

```
class MyList<T>
{
    private object m_lock = new object();
    ...

    private void AddNoLock(T obj)
    {
        m_list.Add(obj);
    }
}
```

```
public void Add(T obj)
{
    lock (_lock)
    {
        AddNoLock(obj);
    }
}

public void AddTwo(T obj1, T obj2)
{
    lock (_lock)
    {
        AddNoLock(obj1);
        AddNoLock(obj2);
    }
}
```

This approach can make code a little more verbose, and, therefore, recursive acquires can be somewhat more convenient to use. With the CLR Monitor class, we cannot assert ownership in AddNoLock. This makes it easy for developers maintaining this class to make a mistake if they don't understand the purpose of the method.

Recursion can be a dangerous feature if not used carefully, however. One of the ways that programmers control this complexity and reason about their program state is by relying on some very basic rules. One of them is quite fundamental: *invariants for data protected by a lock hold at lock acquire and release boundaries*. If a program is written carefully to abide by this rule, it becomes easier to construct reliable, bug-free concurrent systems. When recursion is used, however, this property isn't always easy to guarantee. Invariants may be broken at the time recursion is introduced—particularly with reentrancy—at which point, granting access to a critical region could lead to corruption or crashes.

When it comes to recursive locks, there are three broad categories of how they get used.

1. **Recursive algorithms.** In these cases, an algorithm introduces recursion by design. Sometimes complex recursive cycles in a call graph involving multiple recursive methods, leading up to the recursive lock

acquire, can be tricky to follow and reason about, but this is the easiest to get right. This is the scenario recursive locks are meant to enable.

2. **Dynamic composition.** If you make a dynamic method call while holding a lock, it is possible that the code run dynamically will try to recursively call the subsystem in which the lock was acquired. If recursion was not intended—which is likely given the dynamic nature of this kind of recursion—the affected code may not preserve data invariants at dynamic method call boundaries, and, thus, subtle recursion bugs may arise. It is often best to simply not make dynamic method calls while locks are held.
3. **System introduced reentrancy.** There are several cases—already mentioned above—where the Windows operating system, one of its components, or the CLR introduces reentrancy. This reentrant code can do anything it wishes, including accessing state protected by locks held on the current thread. Often this will not happen, but that's by sheer luck. Because each wait in the CLR is reentrant, the possibility increases. More often than not, such bugs are extraordinarily obscure, only happen when certain components are mixed in certain ways, and are not as pervasive an issue.

To make that last point more clear, let's explore a situation where reentrancy can cause an actual problem. Imagine we have some application specific `Pair` class.

```
class Pair
{
    public int x;
    public int y;
}
```

For whatever reason, let's say there is an invariant on `Pair` that `x == y` (don't ask why). Now pretend the `Pair` is used to represent some private state on a `MyComponent` class.

```
class MyComponent : ServicedComponent
{
    private static Pair p = new Pair();
```

```

public void DoWork()
{
    lock (p)
    {
        Debug.Assert(p.x == p.y);
        p.x++;
        DoMoreWork();
        p.y++;
        Debug.Assert(p.x == p.y);
    }
}

private void DoMoreWork() { /* tolerates broken invariants */ }
}

```

Whenever the component must be updated, `DoWork` acquires a lock around the writes to both `x` and `y` to ensure that they happen in lockstep and that the invariant is preserved. Because we always update them together, we assert that the invariant holds as soon as we enter the lock. All looks well, right? Not quite.

You might not have noticed that `MyComponent` derives from `ServicedComponent`. This is a `ContextBoundObject` that lives by all of the standard COM component rules. (Don't worry about the details here if you're not a COM+ guru.) The important thing to know is that when one is instantiated inside an STA (Single Threaded Apartment), all calls to it are marshaled onto the STA thread, as is the case with ordinary single-threaded COM components. Those calls are placed into the thread's message queue, and are dispatched and run whenever the thread in the STA decides to pump messages.

Let's pretend `DoMoreWork` above did as follows.

```

void DoMoreWork()
{
    Thread.CurrentThread.Join(0);
}

```

Or perhaps it does something else that might block, such as trying to acquire another lock. No matter how the wait occurs, this will pump messages and possibly execute a reentrant call.

Now imagine that we can get this situation to occur.

- A single `MyComponent` object is created inside an STA server.
- We make two calls to `DoWork` on that object from another MTA thread.
- This requires that the MTA post messages to the STA thread's queue.
- The STA thread runs the first call, enters the lock, and performs `p.x++`.
- It then gets to the `DoMoreWork` call, which issues the `Join` and pumps.
- This causes the second call to execute on the STA thread, which enters the lock recursively and sees broken invariants. The assert fires.
- And so on.

There's a fairly obscure set of conditions leading up to the assert. That's often the case with reentrancy bugs. Putting together the precise history leading up to failure is tricky and often requires careful reasoning about the code. But the symptoms can be serious; you're lucky if you get an assert to fire versus randomly corrupting state.

As a rule of thumb, it's a good idea to avoid reentrancy within critical regions unless it is very intentional and well tested. You can achieve this by starting out using nonrecursive locks. That's the best place to start, and you can selectively enable the precise recursive acquisitions that you need for your scenario. You should also avoid dynamic method calls and potential reentrancy points within critical regions, although sometimes this is unavoidable (particularly due to the CLR's automatic pumping policy).

Locks and Process Shutdown

Reliability is of great interest (and greater risk) in concurrent programs. Due to the kinds of correctness problems we're looking at in this chapter, making mistakes that lead to unreliable software is easier to do. There are some specific topics having to do with concurrency and reliability, centered primarily on what happens if a lock is **orphaned**. An orphaned lock is one that was never properly released and yet its owning thread is no longer around. This can be a problem for many reasons. We discussed the topic in Chapter 6, Data and Control Synchronization, particularly as it relates to

CLR monitors. But now we turn to look at what happens to orphaned locks during shutdown.

When a Windows process shuts down, one of the very first things to happen is the abrupt termination of all but one thread. This sole remaining thread is then responsible for performing shutdown duties, both in kernel- and in user-mode. There is a distinction between orderly shutdowns, which notify DLLs that the process is shutting down via `DLL_PROCESS_DETACH` notifications, and rude shutdowns, which don't. Post-Windows 98, the thread anointed shutdown duty is the same thread that initiated the shutdown itself. For Windows 98 and earlier OSs, the choice was effectively random and unpredictable.

If you're programming in Win32, orderly shutdowns are triggered by calls to `ExitProcess`, whereas the rude shutdown is triggered by `TerminateProcess`. These APIs were reviewed extensively back in Chapter 3, Threads. In managed code, the CLR always coordinates closely with the OS to perform shutdown. That almost always means an orderly `ExitProcess`, but can involve a `TerminateProcess` if the CLR isn't able to guarantee a safe shutdown (or if somebody P/Invokes). The CLR also runs some extra managed code when it's shutting down, such as finalizers and `AppDomain` event notifications.

If shutdown is initiated while a lock is held, we'd probably expect any code running shutdown to freely (recursively) acquire it. But what if one of the other terminated threads held locks when shutdown was initiated? Since these threads were killed in a hostile manner, that is, not unwound carefully as with exceptions, these locks will be left in an acquired state. This is often referred to as an **orphaned lock**, as we'll review a bit later. What's worse, any shared state protected by these locks is apt to be in an inconsistent state, with broken invariants, because the thread executing under the protection of the lock might have been in the middle of some multistep operation when it got interrupted.

If we're running an orderly shutdown and the code that runs during shutdown needs to acquire one of those orphaned critical sections, one of two things might be expected: (1) the shutdown could deadlock when trying to acquire an orphaned lock, leading to hangs during process exits and some very frustrated users; (2) the shutdown could be permitted to freely

acquire those locks even though they are orphaned, possibly exposing it to broken invariants left behind. Depending on the circumstances, either one is possible.

The shutdown process is subtly different for native and managed code, so we will review how this problem is dealt with in both environments. Because all managed code builds on top of native code in the process, it's insightful to understand both sides of the story.

Win32: Weakening (Pre-Vista) and Termination (Vista)

Any application that terminates a process by `ExitProcess` should make a best effort at ensuring all threads have reached safe points before termination occurs. If that can't be guaranteed, it's often safer to resort to `TerminateProcess` instead. Although a rude shutdown won't allow DLLs to clean up after themselves—possibly leading to machine-wide resource leaks and/or some small amount of lost data—the consequences, as outlined soon, are often more dire. It's become increasingly more difficult to orchestrate orderly shutdowns with the addition of more third party in-process add-ins and with the increasing amount of concurrent code in such components. Hosting add-ins out-of-process can often be a more robust and reliable way to ensure you can shut down cleanly. In any case, there are bound to be situations in which you're not in control of process termination, have to make the call yourself to `ExitProcess` in a questionable circumstance, or have to deal with bugs. In all of those cases, it's important to understand the behavior of locks during process exit.

Prior to Windows Vista and Server 2008, the OS reacted very dangerously when shutdown code would acquire `CRITICAL_SECTIONS`. We will describe the Vista behavior later, but first, we'll see why the old approach was in need of a change.

Prior to Vista, calls to `EnterCriticalSection` and `LeaveCriticalSection` are effectively ignored during shutdown. A call to acquire a critical section on the shutdown thread will first check to see if the lock is owned by another thread, and, if it is, the section is automatically reinitialized to "available" before acquiring it. This is sometimes called **weakening the lock**. The result? If one of the threads killed during shutdown, `t1`, held on to critical section `CS1`, for instance, and had partially modified some shared

state protected by it just before being killed for shutdown, the shutdown thread t2 is permitted to freely acquire critical section CS1 too, even though it was found as being officially owned by t1.

This means any code running during shutdown in pre-Vista OSs has to tolerate corrupt state that may have been left behind. This is an open-ended requirement that is difficult to achieve, impossible to verify, and many applications get it wrong. It's especially difficult if you write reusable library code that somebody else calls during shutdown—maybe they are unaware it uses locks internally—but under rare circumstances, crashes the shutdown process. The multithreaded CRT uses locks internally for memory allocation and deletion, for instance, and is actually subject to these issues (because it uses locks to protect the free/used lists). It's not even safe to allocate memory during shutdown. Other services are apt to suffer from similar problems.

Waiting on a mutex that was orphaned during shutdown will give you a `WAIT_ABANDONED` return value. This at least allows you to detect that a mutex was orphaned and react accordingly by validating data, skipping a step in the shutdown cleanup, and so forth. Neither weakening nor abandonment apply to other kernel synchronization objects, such as events and semaphores, so you generally can't rely on state invariants associated with them to hold during shutdown either. Generally speaking, if you use any sort of cross-thread synchronization in your `DllMain` method, you are inviting trouble and long hours of debugging. These callbacks must run under the protection of the OS loader lock, which always demands extreme care and thoughtfulness.

Because of the serious problems this can cause, which often lead to shutdown crashes, behavior has changed in Windows Vista. Instead of weakening the locks and permitting threads to observe corrupt state, Windows Vista will immediately terminate the process (via `TerminateProcess`) when an attempt to acquire an orphaned lock is made on the shutdown thread. Although this can lead to some shutdown logic being skipped (which can itself cause problems), all critical data should have been persisted and machine-wide state cleaned up at the application level before the call to `ExitProcess` ever occurred. Any occurrence of termination during shutdown like this is a bug in some code running in the process. The challenge is figuring out in which code that bug lives.

Slim reader/writer lock (SRWLock) acquisitions are inconsistent with everything said above. They are not shutdown aware and, hence, trying to acquire an orphaned SRWLock on the shutdown thread will cause a hang. This might sound bad, but remember, if a lock can be orphaned leading up to a shutdown, there is a bug in the software somewhere. Instead of data corruption, you at least have the opportunity to get a Windows Error Reporting hang entry.

Let's turn to a sample VC++ program that demonstrates this behavior. You wouldn't write code this way; it's been specifically crafted to illustrate the orphaning problem. First we create a DLL to hold all of the interesting code in its DllMain: we initialize a CRITICAL_SECTION, a mutex, and, on Windows Vista, a SRWLock during DLL_PROCESS_ATTACH, and attempt to acquire them during DLL_PROCESS_DETACH. We define an exported function, GetAndBlock, from our DLL that acquires these synchronization objects and sleeps for a long time with them held. This will be called just before we initiate the shutdown process from a separate thread, causing all of the locks to become orphaned. We also define a function IgnoreCriticalSection, which suppresses critical section acquisition on the shutdown code path (to avoid shutdown in the middle of our test on Vista). This sample code will work on both Windows Vista and older OSs, despite SRWLocks not existing, based on whether _WIN32_WINNT is defined at compile time.

```
#include <stdio.h>

// Uncomment when on Vista (or pass it via /D on the cmd-line):
// #define _WIN32_WINNT 0x0600

#include <windows.h>

CRITICAL_SECTION g_cs;
BOOL g_ignoreCs;
HANDLE g_mutex;

#if _WIN32_WINNT >= 0x0600
SRWLOCK g_rwl;
#endif

// Called during process initialization and shutdown.
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
```

```
DWORD dwThreadId = GetCurrentThreadId();

switch (fdwReason)
{
    case DLL_PROCESS_ATTACH:

        // Initialize all of our objects.

        InitializeCriticalSection(&g_cs);
        g_ignoreCs = FALSE;
        g_mutex = CreateMutex(NULL, FALSE, NULL);
#ifndef _WIN32_WINNT >= 0x0600
        InitializeSRWLock(&g_rwl);
#endif

        break;
    case DLL_PROCESS_DETACH:

        // Try to acquire the objects
        // in addition to printing some diagnostics text.

        if (!g_ignoreCs) {
            wprintf_s(L"%x: Acquiring g_cs during shutdown...", dwThreadId);
            EnterCriticalSection(&g_cs);
            printf("success.\n");
            DeleteCriticalSection(&g_cs);
        }

        wprintf_s(L"%x: Acquiring g_mutex during shutdown...", dwThreadId);
        DWORD result = WaitForSingleObject(g_mutex, INFINITE);
        if (result == WAIT_ABANDONED)
            wprintf_s(L"abandoned.\n");
        else
            wprintf_s(L"success.\n");

        CloseHandle(g_mutex);

#ifndef _WIN32_WINNT >= 0x0600
        wprintf_s("%x: Acquiring g_rwl (X) during shutdown...", dwThreadId);
        AcquireSRWLockExclusive(&g_rwl);
        wprintf_s(L"success.\n");
#endif

        break;
}
return TRUE;
```

```

}

__declspec(dllexport) DWORD WINAPI GetAndBlock(LPVOID lpParameter)
{
    DWORD dwThreadId = GetCurrentThreadId();

    // Acquire the locks.
    EnterCriticalSection(&g_cs);
    wprintf_s(L"%x: g_cs acquired.\n", dwThreadId);

#ifndef _WIN32_WINNT >= 0x0600
    AcquireSRWLockExclusive(&g_rwl);
    wprintf_s(L"%x: g_rwl (X) acquired.\n", dwThreadId);
#endif

    WaitForSingleObject(g_mutex, INFINITE);
    wprintf_s(L"%x: g_mutex acquired.\n", dwThreadId);

    // And just wait for a little while...
    SleepEx(25000, TRUE);

    return 0;
}

__declspec(dllexport) VOID WINAPI IgnoreCriticalSection()
{
    g_ignoreCs = TRUE;
}

```

Next, we define an EXE that invokes `GetAndBlock` and initiates a process shutdown on separate threads. If an argument is supplied, we call `IgnoreCriticalSection`; this allows us to test both critical section and SRWLock acquisition on Vista. Since neither will return successfully, we can only call one or the other. The result is that the shutdown thread acquires the synchronization objects of which the `GetAndBlock` thread currently has ownership.

```

#include <windows.h>
#include <stdio.h>

// Forward-decl the DLL methods we will call.
DWORD WINAPI GetAndBlock(LPVOID lpParameter);
VOID WINAPI IgnoreCriticalSection();

int main(int argc, wchar_t * argv[])
{
    // If any args were supplied, we turn off CRST shutdown acquisition.

```

```

if (argc > 1)
    IgnoreCriticalSection();

// Create a thread to acquire the locks.
HANDLE hT1 = CreateThread(NULL, 0, &GetAndBlock, NULL, 0, NULL);

// Wait for it to run.
SleepEx(100, TRUE);

// Now trigger process exit.
ExitProcess(0);
}

```

The results of running this program depend on whether you are running on Windows Vista or a previous operating system. Pre-Vista, you will see that the critical section is reacquired, that the mutex acquisition reports back WAIT_ABANDONED, and the shutdown process will terminate normally.

```

C:\...>shutdown.exe
664: g_cs acquired.
664: g_mutex acquired.
d18: Acquiring g_cs during shutdown...success.
d18: Acquiring g_mutex during shutdown...abandoned.

```

As expected, no hangs occur. Now on Vista, when run with the critical sections acquisition enabled on shutdown, we see that the process dies and winks out of existence as soon as we try to acquire the critical section.

```

C:\...>shutdown.exe
664: g_cs acquired.
664: g_rwl (X) acquired.
664: g_mutex acquired.
d18: Acquiring g_cs during shutdown...

```

Finally, still on Vista, if we pass an argument when running the program, critical section acquisition is suppressed, and we see that acquiring the SRWLock hangs the process.

```

C:\...>shutdown.exe no_crst
664: g_cs acquired.
664: g_rwl (X) acquired.
664: g_mutex acquired.
d18: Acquiring g_mutex during shutdown...abandoned.
d18: Acquiring g_rwl (X) during shutdown...

```

We never get control back from that last line. We must kill the process.

Managed Code: Shutdown Watchdog

The philosophy for shutdowns in managed code is very different from in native. The CLR exits the process when all primary threads have exited but while background threads may still be actively running code. Thus, unlike `ExitProcess` where all threads are supposed to rendezvous to enable a clean shutdown that doesn't require rude termination of code, the CLR and .NET Framework library developers must regularly deal with the consequences of a shutdown orphaning locks. It's an expected part of the system's architecture. It's also possible to turn around and call `Environment.Exit`, which, in .NET, is acceptable.

Managed DLLs have no equivalent to `DllMain` (although mixed-mode binaries can). So the only managed code that runs during an orderly shutdown is raising the `AppDomain.ProcessExit` event (for each AppDomain) and finalizing the entire heap (which invokes the `Finalize` method for all finalizable objects). The term "orderly shutdown" is used to distinguish a call to `Environment.Exit` from a disorderly P/Invoke to `TerminateProcess`, for instance. The latter case mostly circumvents the CLR's shutdown logic—though it does get notified in its `DllMain`—including these two steps. Unlike native code, threads are first suspended while the CLR is performing managed shutdown; not terminated. Eventually the CLR will call `ExitProcess`, at which point native code in the process gets a chance to run, such as `DLL_PROCESS_DETACH` notifications.

As with the example described for native code, threads can be suspended while they hold arbitrary locks and have partially mutated state to the point where invariants do not hold any longer. Lock acquisitions during managed shutdowns (e.g., via `Monitor.Enter` and `Monitor.Exit`) are treated more like Windows Vista `SRWLocks` rather than critical sections. The CLR does not allow acquisition of orphaned monitors (as with weakening prior to Windows pre-Vista) nor does the CLR terminate the process when one occurs (like Vista's new behavior). Instead, the CLR mitigates the risk of deadlock and hangs by having a watchdog thread monitor shutdown instead of tolerating state corruption and crashes.

If an acquisition of an orphaned lock happens during shutdown, a hang will ensue. (Forget about timeouts for a moment.) To deal with shutdown hangs, one of the first things the CLR does during orderly shutdown is to

create a watchdog thread that monitors the shutdown process. Although changeable by CLR hosts, the CLR will by default allow the AppDomain.ProcessExit and all relevant finalizers to run for 2 seconds before becoming impatient. If this period of time is exceeded, the shutdown thread is suspended, and the CLR shutdown process continues without running any more managed code.

This can be illustrated by the following code example.

```
using System;
using System.Threading;

class Program
{
    private static object s_lock = new object();

    public static void Main()
    {
        // Create 10 new finalizable objects.
        Program[] p = new Program[10];
        for (int i = 0; i < p.Length; i++)
            p[i] = new Program();

        // Obtain the lock and then force a process exit.
        lock (s_lock)
        {
            Environment.Exit(-1);
        }

        // Ensure the objects don't become unreachable before exiting.
        GC.KeepAlive(p);
    }

    ~Program()
    {
        Console.WriteLine("acquiring s_lock...");

        // This lock acquisition will always hang...
        lock (s_lock)
        {
            Console.WriteLine("Got it!? Nope.");
        }
    }
}
```

When this program runs, only one finalizer will run, and it will freeze for about 2 seconds after the shutdown is initiated by the call to `Environment.Exit`. This happens because the attempt to acquire `s_lock` from Program's finalizer deadlocks, and the watchdog eventually kills the thread, skipping the remaining 9 finalizers in the queue. The code in `Main` that initiated the shutdown will have orphaned `s_lock` by calling `Exit` while it was held. The same would have occurred if we attached an event handler to `AppDomain.Current.ProcessExit` that tried to acquire `s_lock`, for example.

This same policy applies to any synchronization objects including managed reader/writer locks, events and condition variables, and any other type of interthread communication. You might expect that mutexes would behave in managed code as they do in Win32 during process exit, given that `Mutex` is a thin wrapper over the OS mutex APIs. In other words, you'd expect a call to `Mutex.WaitOne` on an orphaned mutex to throw a `MutexAbandonedException`. If that happened, the unhandled exception would probably crash the finalizer thread and, hence, the entire process during shutdown. That's not what happens. Because shutdown-oriented managed code runs before `ExitProcess` is called, threads that own abandoned mutexes are just suspended (not killed); thus, the mutexes aren't abandoned, and attempts to acquire them will hang.

The manifestation of these sorts of hangs is often not horrible. Many finalizers are meant to clean up intraprocess state anyway, and because `HANDLE` lifetime is tied to the process lifetime, Windows will close them automatically during process exit. But a hang means that additional library and application logic won't run, like flushing `FileStream` write buffers. And for any cross-process state, you should always have a fail-safe plan in place, such as detecting corrupt machine-wide state and repairing it upon the next program restart. This is similar to what must be done with native code, given that the process will terminate if you try to acquire an orphaned lock. Finally, a 2 second pause doesn't seem like much, but it's long enough that most users will notice it. Avoiding cross-thread coordination during shutdown is considered a best practice, and it can help to (statistically) improve the user experience for shutdowns.

Liveness Hazards

Although liveness hazards don't normally cause programs to compute incorrect results, as correctness hazards do, they can stop programs from producing results at all. Or they can interfere with a program's ability to make forward progress temporarily, yielding hard to diagnose performance problems. In this section, we look at the most pervasive kinds of liveness problems, starting with the one that most people are already familiar with: deadlocks.

Deadlock

Once a thread needs to hold exclusive access to more than one lock at a time, deadlock becomes possible. This is often called a **deadly embrace**, because unless something gives your program will come to a halt (or at least some portion of it will). What's worse is that deadlocks, just like race conditions, depend on the timing of your program and are hard to find.

Examples of Deadlock

Transferring Money Between Two Bank Accounts. As an example of a deadlock, imagine we have a `BankAccount` class. It provides the ability to transfer between two accounts, requiring that more than one lock is held (in case the same accounts are used concurrently). If we don't hold both locks at once, we can cause atomicity problems where it's possible to observe a state in which money has been removed from one account but not yet placed into the other. The obvious approach to transfer funds looks like this.

```
class BankAccount
{
    private int m_id = ...;
    private decimal m_balance = ...;
    private object m_syncLock = new object();

    public static void Transfer(
        BankAccount a, BankAccount b, decimal amount)
    {
        lock (a.m_syncLock)
        {
            if (a.m_balance < amount)
                throw new Exception("Insufficient funds.");

            lock (b.m_syncLock)
```

```

    {
        a.m_balance -= amount;
        b.m_balance += amount;
    }
}

...
}

```

All looks well, and this code will work correctly . . . most of the time. To illustrate the flaw, imagine that we have two `BankAccount` objects—one for account #1234 and another for account #4321—and that one thread tries to transfer \$100 from #1234 to #4321 at the exact same time that some other thread tries to transfer \$500 from #4321 to #1234. The synchronization logic will work correctly, ensuring no money will get lost in the process. But if the following specific interleaving of events were to occur, the program would lock up indefinitely.

T	t1	t2
0	<code>lock (#1234.m_syncLock)</code>	
1		<code>lock (#4321.m_syncLock)</code>
2	<code>lock (#4321.m_syncLock)</code>	
3		<code>lock (#1234.m_syncLock)</code>
	<code>** deadlocked **</code>	<code>** deadlocked **</code>

What happened here? First thread t1 successfully acquires a lock on account #1234. Then t2 runs and successfully acquires a lock on account #4321. The program is doomed at this point. When t1 then tries to acquire a lock on #4321, it is unable, because t2 currently holds the lock, and so it must wait until t2 releases it to proceed. Then t2 goes ahead and tries to acquire #1234, which similarly cannot happen because t1 owns the lock, and waits too. Both t1 and t2 end up waiting for one another. Neither can proceed and both will wait forever.

The Dining Philosophers Problem. Another problem is often used to illustrate deadlock: **the dining philosophers problem**, originally attributed to Edsger Dijkstra (see Further Reading) and later renamed to the Five Dining Philosophers problem by Tony Hoare. It is quite simple. Five philosophers (numbered 0 through 4) sit at a table with five chairs, plates, and forks. Each philosopher has one of each and alternates between thinking and eating.

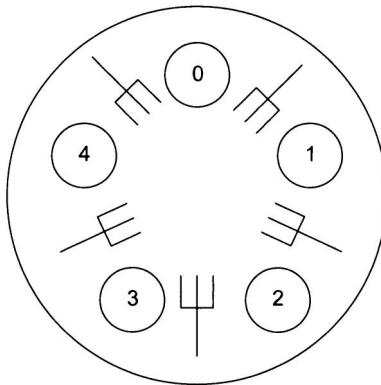


FIGURE 11.1: Five dining Philosophers, each with his own chair, plate, and fork

Unfortunately, the food being eaten is difficult (spaghetti), and requires two forks to be eaten. Thankfully each philosopher can easily access two forks—one to his left and one to his right—but this requires that two adjacent philosophers cannot be eating simultaneously.

If you haven't noticed the deadlock yet, here it is. Imagine that, as a protocol, all philosophers begin eating by grabbing the left fork and then the right. If a neighboring philosopher holds one of the forks, then the philosopher in question must wait for his neighbor to put the fork down. Now, imagine all philosophers decide to grab the left fork at once. Each will succeed. But now no forks are available! When each tries to grab the right fork, each will find it to be held by his neighbor and, hence, each philosopher must wait (indefinitely).

Deadlocks without Locks. Deadlocks have to do with any kind of “shared resource” and are not limited to locks. There are even subtler ways in which a real deadlock might occur.

A single threaded apartment (STA), of the kind we discuss further in Chapter 16, Graphical User Interfaces, is equivalent to an exclusive lock. Only one thread can update a GUI window or run code inside an apartment-threaded COM at once. And this STA lock can only be released by running messages in the queue, either by finishing the actively running callback or pumping the queue. Failure to pump often leads to liveness problems, but not deadlock, such as a delay in processing messages. But if some code

running on the STA thread depends on code that is waiting to run on the STA thread (perhaps because it's been enqueued into the message queue) then a true deadlock could result. The CLR pumps messages automatically during a wait, reducing the likelihood of this but it can show up in native code.

Even more obscure examples exist. Here's a classic example of an STA induced deadlock. A thread running in an STA generates a large quantity of apartment threaded COM component instances and their corresponding runtime callable wrappers (RCWs). These RCWs must be finalized by the CLR when they become unreachable, or they will leak. But the CLR's finalizer thread always joins the process's multithreaded apartment (MTA), meaning it must use a proxy that transitions to the STA in order to release the RCWs (according to COM's strict apartment rules). If the STA doesn't pump and dispatch the finalizer's attempt to finalize the RCW, however—perhaps because it has chosen to block using a nonpumping wait—the finalizer thread will be stuck. It is blocked until the STA unblocks and pumps. If the STA never pumps, the finalizer thread will never make any progress, and a slow, silent buildup of all finalizable resources will occur over time (see Further Reading, Brumme). This can, in turn, lead to a subsequent out-of-memory crash or a process recycle in ASP.NET.

Different types of deadlocks require different techniques to combat. Most of this section focuses on lock based deadlocks exclusively because they are most common. It is worth mentioning that CLR 2.0 introduced a managed debugging assistant (MDA), `ContextSwitchDeadlock`, which monitors for deadlocks induced by cross-apartment proxies and failure to pump. If a cross-apartment call takes longer than 60 seconds to complete, the CLR assumes the receiving STA is not pumping and fires this MDA.

Avoiding and Detecting Deadlocks

Generally speaking, there are four conditions necessary for deadlock.

1. **Mutual exclusion.** Using a resource prevents all other threads from accessing it.
2. **Waiting.** After acquiring some resource, a thread may wait for another resource, which itself could be, at that moment, held exclusively by another thread.

3. **Lack of preemption.** A resource held by one thread cannot be forcibly taken away by another thread. The owning thread will relinquish ownership of a resource only after it has finished using it.
4. **Circular wait.** A chain of threads exists in which each thread owns one or more resources being requested by the next thread in the chain.

These are known as the **Coffman conditions** (see Further Reading, Coffman, Elphick, Shoshani) and are readily described in any OS course book. In this definition a resource can mean many things: a critical region, kernel object, I/O resource, and so on. Most deadlocks in modern concurrent programs are due to critical regions, such as Win32 CRITICAL_SECTIONS and CLR Monitors, although variants on the idea are also common, which lead to deadlock-like symptoms (such as missed events).

While circular waits involving two threads are fairly obvious, piecing together deadlocks consisting of more than one thread are more difficult (though no less possible). As an illustration, imagine that three threads hold separate locks: thread 1 holds lock A, thread 2 holds B, and thread 3 holds C. If thread 3 suddenly tries to acquire lock A, a deadlock will occur.

Aside from eliminating concurrency altogether, one of the Coffman conditions must be mitigated in order to avoid or react to deadlocks. Here are some examples of how.

1. **Mutual exclusion.** Some resources can be shared, for instance by using a lock with a shared-mode (e.g., a reader/writer lock). If this is possible, mutual exclusion is not present and, therefore, won't create indefinite waiting. But with common locks like CRITICAL_SECTIONS and Monitors, this is a nonnegotiable aspect to the lock itself. You can't change it.
2. **Waiting.** If a program never had to hold more than one lock at a time, this wouldn't be an issue. The very basic bank account example earlier should convince you this isn't always feasible. Most locking primitives offer a "try enter" method of acquisition that uses a timeout to avoid waiting indefinitely. It is possible, within some fairly closed-world scenarios, to use a timeout as an opportunity to voluntarily back off, releasing some resources to allow others to proceed, and then restarting the whole operation. This isn't always possible.

3. **Lack of preemption.** Transactional systems often deal with deadlock by preempting one of the participants in a wait chain. This transaction is then forced to relinquish its resources and retry the transaction again. Though this feature isn't available in general programming environments, it is certainly one reasonable (and reasonably successful) approach to dealing with deadlocks. Using thread interruption and termination is not an appropriate way to do this.
4. **Circular wait.** By enforcing an ordering on locks and mandating that threads always acquire locks in that certain order, circular acquires can be made impossible and, hence, so too are deadlocks. This is perhaps the most promising of the four conditions to eliminate, as we will focus on below.

The Banker's Algorithm and Simultaneous Lock Acquisition. The first famous, but seldom used in practice, technique for avoiding deadlocks is called **The Banker's Algorithm** and was also invented by Edsger Dijkstra (see Further Reading). (If it's not obvious, Dijkstra was quite fascinated by synchronization.) For The Banker's Algorithm to work, the complete set of resources that a thread will hold at once must be known. Armed with this information, the system will know that any particular acquisition won't put the system in a deadlock prone state. If the acquisition would indeed compromise the system, the acquiring thread must wait for other conflicting threads to finish the conflicting operations before even starting its own operation. No step is permitted that could eventually lead to deadlock, therefore eliminating the possibility.

While interesting from a theoretical perspective, The Banker's Algorithm is seldom applied in general purpose programming environments. Knowing, for any arbitrary thread, the complete set of locks it will ever hold at once is impossible in today's world of dynamically composed software, without some fairly extravagant changes to the programming model. With that said, we can borrow and use the core idea in closed settings.

If we carefully structure software into subsystems in which dependencies are always unidirectional and where there are no circular dependencies—a generally accepted practice in software design—then we can use a variant of The Banker's Algorithm to avoid deadlocks. We call

this **simultaneous multilock acquisition**. Here's how it works. When a call enters the subsystem, the full set of needed locks is acquired at once. This solves our earlier BankAccount example, because all locks needed to transfer between two accounts is known.

```
class BankAccount
{
    private decimal m_balance = ...;
    private object m_syncLock = new object();

    public static void Transfer(
        BankAccount a, BankAccount b, decimal amount)
    {
        MultiLockHelper.Enter(a, b);
        try
        {
            if (a.m_balance < amount)
                throw new Exception("Insufficient funds.");

            a.m_balance -= amount;
            b.m_balance += amount;
        }
        finally
        {
            MultiLockHelper.Exit(a, b);
        }
    }

    ...
}
```

The idea is that `MultiLockHelper.Enter` acquires the full set of locks provided, or it acquires none of them. The region executed afterwards is brief and does not acquire any additional locks. Of course, locks aren't really acquired "at once." Win32 critical sections and CLR monitors don't support that. But because all of the lock acquisitions happen in the same location, we can simulate this by implementing some clever logic that avoids deadlock.

That last bit is the interesting part: How do we implement such "clever logic"? One possible solution is to detect contention dynamically and to back off using some spinning and, possibly, waiting. But this can be quite wasteful and could trade deadlock for livelock (more on that soon). An alternative strategy is to sort the locks first and then acquire them: so long

as all multiacquisitions of these particular locks use the same ordering, we are guaranteed deadlock freedom. The ordering idea will be taken further in the next section:

To sort the locks we need a key. Recall that `BankAccount` objects have unique identifiers (their `m_id` fields), so we can use that as a sort key for our specific scenario.

```
using System;
using System.Threading;

internal static class MultiLockHelper
{
    internal static void Enter(BankAccount a, BankAccount b)
    {
        if (a.m_id < b.m_id)
        {
            // Acquire a first, and then b:
            Monitor.Enter(a.m_syncLock);
            try
            {
                Monitor.Enter(b.m_syncLock);
            }
            catch
            {
                Monitor.Exit(a.m_syncLock); // b failed
                throw;
            }
        }
        else
        {
            // Reverse order. Acquire b first, and then a:
            Monitor.Enter(b.m_syncLock);
            try
            {
                Monitor.Enter(a.m_syncLock);
            }
            catch
            {
                Monitor.Exit(b.m_syncLock); // a failed
                throw;
            }
        }
    }

    internal static void Exit(BankAccount a, BankAccount b)
    {
        if (a.m_id < b.m_id)
```

```
        // Reverse order of acquire: b then a.  
        Monitor.Exit(b.m_syncLock);  
        Monitor.Exit(a.m_syncLock);  
    }  
    else  
    {  
        // Reverse order of acquire: a then b.  
        Monitor.Exit(a.m_syncLock);  
        Monitor.Exit(b.m_syncLock);  
    }  
}
```

This approach ensures deadlock free Transfer operations. And it doesn't really add any additional overhead, although the reason why it's correct is somewhat subtle. It works for our specific example of exactly two Bank-Account objects, but doesn't scale to all possible cases.

To support a broader range of scenarios, we can resort to doing a general sort instead.

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Runtime.CompilerServices;

internal static class MultiLockHelper<T> where T : IComparable<T>
{
    internal static void Enter(params T[] locks)
    {
        Array.Sort(locks);

        // Now perform the waits in sorted order.
        int i = 0;
        try
        {
            for ( ; i < locks.Length; i++)
                Monitor.Enter(locks[i]);
        }
        catch
        {
            // Undo the successful acquisitions.
            for (int j = i - 1; j >= 0; j--)
                Monitor.Exit(locks[j]);
            throw;
        }
    }
}
```

```
internal static void Exit<T>(params T[] locks)
{
    Array.Sort(locks);

    // Exit the locks in reverse sorted order.
    for (int i = locks.Length - 1; i >= 0; i--)
        Monitor.Exit(locks[i]);
}
```

This code has some disadvantages. One clear disadvantage is the performance overhead for doing a sort. We also have to do it twice, once for Enter and once for Exit, although this could be avoided. If the caller passed the same locks array to both methods, we could sort it in place in Enter and then skip the sort entirely inside of Exit assuming the same array is supplied.

Another disadvantage is that locks themselves don't always have unique keys associated with them. When coding in C++ with CRITICAL_SECTIONS, you can sort on the memory address; and with kernel objects, you can use the HANDLE value. Both are guaranteed unique and stable. But CLR monitors can be any kind of CLR object, so you need to implement ordering at some higher level (hence the restriction in MultiLockHelper<T> above that the generic argument T implements IComparable<T>). We could do this in our BankAccount example by combining the `m_id` and `m_syncLock` fields into a single comparable object.

Lock Leveling. All of this talk about ordering locks during acquisition brings us to our next technique for avoiding deadlocks: **lock leveling**. This technique is commonly known under several other guises: **lock ranking**, **lock hierarchies**, and **lock ordering**, among others. We already stated that if threads always acquire locks in a consistent order, there will be no deadlocks, but it may not be obvious why this is true. Cycles are required to produce a deadlock, and consistent ordering (with no exceptions) eliminates the possibility of cycles.

Imagine we assign a unique level to each lock in the system. (This is stronger than the previous example, where only like locks needed to be disambiguated.) Then, if a thread only waits for locks with a level "less" than the lowest level already currently held, it is enough to guarantee deadlock freedom by construction. Strict adherence to the leveling scheme can be statically verified in the best case, and dynamically verified in the worst.

All of this sounds great. But if it's so great, you might ask, why isn't lock leveling already used pervasively as a deadlock prevention technique? Lock leveling is actually a tad onerous and constraining for a few reasons.

- Assigning levels to your locks requires careful planning and a bit more engineering discipline. It is hard to come up with levels in the first place—demanding careful thought about the global layering of the system's components—and forethought into specifically where locks will be necessary.
- After that, maintaining the levels that you have assigned can be a chore.
- Once you have come up with levels, the restrictions can sometimes be too great: lock leveling effectively requires static knowledge of call graphs around critical regions. With late bound method calls (virtuals, function pointers, delegates), this is difficult. Simultaneous lock acquisition (shown earlier) can be used to disambiguate certain cases where the relative ordering of a fixed number of locks isn't known statically, but can't handle all cases. Making a late bound call from inside of a critical region is a very bad practice anyhow, so one could argue that this is indicative of deeper problems.
- The last reason lock leveling isn't used heavily is that neither C++ nor .NET offer out-of-the-box support for it; the result is that most people aren't even aware that it exists.

All that said, most arguments against lock leveling boil down to the inconvenience they pose to the development process. It is ultimately up to you to decide whether or not that inconvenience is worth the added safety it brings. I know which choice I would make.

Let's take an example of using leveled locks. Imagine we have two subsystems, A and B, protected by a lock apiece. We could assign system A level 10 and system B level 5. The rationale behind doing so could be that A represents a higher-level subsystem (like a business logic layer) and B represents a lower-level subsystem (like a data persistence engine). Notice how the assignment of levels closely maps to the way a system is factored: upward dependencies from B to A are probably prohibited, so the lock leveling requirements should pose no problems.

If we had a `LeveledLock` class, we might construct instances of these as follows:

```
LeveledLock lockA = new LeveledLock(10);
LeveledLock lockB = new LeveledLock(5);
```

If any thread needs to hold both `lockA` and `lockB` simultaneously, it must first acquire `lockA` and then `lockB`, in that order. Acquiring in the opposite order is an error by construction. Ideally this would be a compile time error, but that requires some kind of static analysis; instead, we will explore making this a runtime error.

There are some corner cases. Intralevel lock acquisitions are typically illegal. If you hold lock A at level 10 and attempt to acquire some other lock C at level 10, the attempt should fail. If this were legal, the two threads could deadlock: if one acquires A and then C, and another acquires C and then A, deadlock occurs. It's usually best to decide which order is legal and to codify it in the levels assigned by ensuring no two locks can share the same level. Because recursive lock acquires never wait and are confined within a single thread, they can be safely allowed without risking deadlock. But unless a recursive acquire immediately follows the prior acquires of that lock, recursion can be an indication of a poor layering that may become deadlock prone in the future. Be on the lookout for this.

Ensuring that coarse-grained locks are acquired in the correct order by construction is often straightforward. But fine-grained locks pose more challenge because many locks logically end up at the same "layer" in a program. The original illustration of transferring funds between two `BankAccount` objects requires more thought. One could assign levels to the locks based on an account's unique identifier and continue using some kind of multilock acquisition technique to take more than one at a time. With lock leveling, sorting the locks is matter of comparing each lock's level with respect to one another. But if the multiple locks aren't acquired all at once, we run up against the limits of lock ordering.

If we assign levels based on account identifiers, it becomes hard to place them relative to other locks in the system, especially if account identifiers can take on any value in the range of 32-bit integers. This reflects a basic flaw in the use of absolute numbers to express levels. Some lock leveling systems instead allow relative orderings to be expressed. This is helpful,

but it can be difficult to eliminate the possibility of cycles in the relative relationships expressed. If identifiers are within a well-defined range—say, 1 through 200,000—then you can set aside some range—such as 2,000,000 through 2,200,000—and order all other locks around it.

Similarly, lock orderings are often only applicable to code within a single assembly. It's unlikely that a lock at level 100 in an official .NET binary such as `System.Core.dll` would carry any relationship at all to a lock given level 101 in some application specific `FooCompany.dll`. In fact, the levels themselves are quite arbitrary; instead, it's better to assume the levels represent two entirely separate systems, or to even level the assemblies among each other, for example, saying `System.Core.dll` can't call `FooCompany.dll` when a lock is held.

Let's look at a sample implementation in .NET of a `LeveledLock` class. Based on the description before, I'm sure you get the gist of the idea. But seeing it written out can be useful. The following is a fully functional implementation of a simple lock leveling scheme. Feel free to use it in your own code. It is very straightforward to follow.

```
#define LOCK_TRACING

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Reflection;
using System.Threading;

namespace LockLeveling
{
    public sealed class LeveledLock
    {
        // Static fields
        [ThreadStatic]
        private static Dictionary<Assembly, Stack<LeveledLock>> s_currLevels;

        // Fields
        private object m_lock = new object();
        private int m_level;
        private bool m_allowRecursion;
        private string m_name;
```

```
// Constructors
public LeveledLock(int level, bool allowRecursion, string name)
{
    m_level = level;
    m_allowRecursion = allowRecursion;
    m_name = name;
}

// Properties
public int Level
{
    get { return m_level; }
}

public bool AllowRecursion
{
    get { return m_allowRecursion; }
}

public string Name
{
    get { return m_name; }
}

// Methods
public void Enter()
{
    TryEnterCore(
        Assembly.GetCallingAssembly(), false, Timeout.Infinite);
}

public void Enter(bool permitIntraLevel)
{
    TryEnterCore(
        Assembly.GetCallingAssembly(), permitIntraLevel,
        Timeout.Infinite);
}

public bool TryEnter(int millisecondsTimeout)
{
    return TryEnterCore(
        Assembly.GetCallingAssembly(), false,
        millisecondsTimeout);
}

public bool TryEnter(
    bool permitIntraLevel, int millisecondsTimeout)
{
```

```
        return TryEnterCore(
            Assembly.GetCallingAssembly(), permitIntraLevel,
            millisecondsTimeout);
    }

    private bool TryEnterCore(
        Assembly caller, bool permitIntraLevel,
        int millisecondsTimeout)
    {
        bool taken = false;

        Thread.BeginCriticalSection();
        try
        {
            PushLevel(caller, permitIntraLevel);
            taken = Monitor.TryEnter(m_lock, millisecondsTimeout);
        }
        finally
        {
            if (!taken)
                Thread.EndCriticalSection();
        }

        return taken;
    }

    public void Exit()
    {
        Monitor.Exit(m_lock);
        try
        {
            PopLevel(Assembly.GetCallingAssembly());
        }
        finally
        {
            Thread.EndCriticalSection();
        }
    }

    [Conditional("LOCK_TRACING")]
    private void PushLevel(Assembly caller, bool permitIntraLevel)
    {
        Stack<LeveledLock> levelStack = null;

        // Find the current stack of levels, if any.
        if (s_currLevels == null)
            s_currLevels = new
                Dictionary<Assembly, Stack<LeveledLock>>();
        else
            s_currLevels.TryGetValue(caller, out levelStack);
    }
}
```

```
if (levelStack == null)
{
    levelStack = new Stack<LeveledLock>();
    s_currLevels.Add(caller, levelStack);
}
else if (levelStack.Count > 0)
{
    // If locks are held, validate acquiring this one is OK.
    LeveledLock current = levelStack.Peek();
    int currentLevel = current.m_level;

    if (m_level > currentLevel ||
        (current == this && !m_allowRecursion) ||
        (m_level == currentLevel && !permitIntraLevel))
        throw new LockLevelException(current, this);
}

// OK to proceed with locking. Put the new lock in TLS.
levelStack.Push(this);
}

[Conditional("LOCK_TRACING")]
private void PopLevel(Assembly caller)
{
    if (s_currLevels == null)
        throw new InvalidOperationException(
            "No locks acquired");

    Stack<LeveledLock> levelStack;
    if (!s_currLevels.TryGetValue(caller, out levelStack))
        throw new InvalidOperationException(
            "No locks acquired in this assembly");

    // Just pop the latest level placed into TLS.
    if (levelStack.Count == 0 || levelStack.Peek() != this)
        throw new InvalidOperationException(
            "Out of order release detected");

    levelStack.Pop();

    // Clean up garbage.
    if (levelStack.Count == 0)
    {
        s_currLevels.Remove(caller);
        if (s_currLevels.Count == 0)
            s_currLevels = null;
    }
}
```

```

        public override string ToString()
    {
        return string.Format(
            "<level={0}, allowRecursion={1}, name={2}>",
            m_level, m_allowRecursion, m_name
        );
    }

    public class LockLevelException : Exception
    {
        public LockLevelException(
            LeveledLock currentLock, LeveledLock newLock) :
            base(string.Format(
                "You attempted to violate the locking protocol" +
                "by acquiring lock {0} while the thread already" +
                "owns lock {1}.", currentLock, newLock)) { }
    }
}

```

At construction time, we provide the lock's level, whether we support recursive acquires, and a name for the lock (just for diagnostics purposes). Then we proceed to use it as we would any other lock: acquisitions use the `Enter` method, of which there are a few overloads (to support timeouts), and releases use the `Exit` method. The implementation uses a CLR monitor underneath to achieve mutual exclusion, perform waiting, and so on.

The lock leveling aspects are simple to follow. A single `ThreadStatic` field is used to keep the levels of locks held by the current thread. This is kept in a dictionary so we can track separate lists of levels per unique `Assembly`, which we retrieve by calling the static `Assembly.GetCallingAssembly` from our `Enter` and `Exit` methods. The list of levels is held in a `Stack`, which enforces that they are also released in the reverse order in which they were acquired. When `Enter` or `TryEnter` is called, we defer to the private `PushLevel` method; similarly, when `Exit` is called, we defer to `PopLevel`. Both of these methods do simple bookkeeping on the dictionary and stack for the calling thread. During acquisition, the `PushLevel` method throws a `LockLevelException` (which has a nice diagnostics message) if one of a set of conditions holds: (1) if the target level is higher than the most recent acquisition; (2) if the target lock is the same lock as the most recently acquired one, and we've disabled recursive acquisitions; or, (3) the target

lock is a different lock, but the same level, and we have specified `false` for the `permitIntraLevel` argument (the default).

Many lock leveling systems are turned off in nondebug builds to avoid the performance penalty of maintaining and inspecting lock levels at run time. This is the purpose of the `LOCK_TRACING` conditional symbol. Turning it off and recompiling the implementation makes `LeveledLock` work the same as a standard CLR monitor by statically removing the calls to `PushLevel` and `PopLevel`. Some kind of runtime configuration could have been used instead, for example, if `LeveledLock` was in a separately compiled assembly. Turning this off requires thorough testing to uncover all violations of the locking protocol because turning it off will possibly lead to deadlocks instead of level violation exceptions. Dynamic composition of the kind we discussed earlier makes this level of test coverage hard to achieve in practice.

Deadlock Detection

Wholesale deadlock prevention is not always possible. Often we can instead detect when one has occurred. To determine whether deadlock has happened requires construction of a **wait graph**, which simply exposes the dependencies between those waiting for locks and those that already hold locks of interest. Wait graphs are great debugging aids for tracking down how deadlocks have occurred, and some real systems can use them to break deadlocks.

Relational databases, for example, allow developers to query and update tables, requiring locks of various kinds. But a single query can require multiple locks: SQL uses a hierarchy of locks (tables, pages, rows), and a query may span multiple of any of those units. Calculating the whole lock set is not always possible, and asking the programmer to do so is more burdensome than is warranted. Instead, most databases detect deadlocks when they occur and respond by choosing a victim, killing the victim's transaction (undoing any uncommitted actions) and permitting other transactions in the system to proceed. An application must code for this circumstance, the most common response of which is to retry the operation.

Similar approaches clearly won't work well for general purpose programming environments. Threads that have accumulated locks are not

transactional and, therefore, can't be aborted in the middle of execution without the risk of corrupting state. Closed systems could be developed with an awareness of deadlock detection, but this technique is not broadly useful.

Although deadlock detection isn't a great way to respond at runtime to deadlocks, it is a very useful diagnostics tool. It's relatively straightforward to write a wrapper on top of your favorite locking primitive that, when a deadlock is suspected, performs a complete deadlock detection algorithm for tracing purposes. The algorithm for detecting such a deadlock is basic and can be used in many settings. The trick is figuring out how to plumb your favorite synchronization primitives so that a wait graph can be constructed when necessary. A wrapper type can be used (as shown by Stephen Toub in his *MSDN Magazine .NET Matters* column, [see Further Reading]), the CLR hosting APIs can be used to hook blocking events (as I did in a previous *MSDN Magazine* article [see Further Reading, Duffy, April 2006]), and the new Windows Vista Wait Chain Traversal (WCT) APIs can be used (for native locks only—they don't currently support managed code).

In this section we will take a look at a sample deadlock detection algorithm in addition to the WCT APIs, but won't build a fully capable deadlock detecting lock. For this, please refer to one of the aforementioned *MSDN Magazine* articles.

Deadlock Detection Wait Graph Algorithm. To build a wait graph, we need two pieces of information.

1. A mapping of all locks held by all threads.
2. A list of which locks certain threads are currently waiting to acquire.

So, the first step in enabling creation of a wait graph is to track this information.

Once a deadlock is suspected, we can use these two things to build a graph. Building a graph is not cheap, as it requires tracking the aforementioned information, inspecting many shared data structures (depending on the specific mechanisms you've used to track the information), and involves a loop that is $O(N)$ where N is the size of the longest possible wait chain in your system. Common approaches include doing this on demand

when a debugger is attached, for debug builds only, or to run the algorithm in response to an acquisition timeout.

Here is some C# code that implements the general algorithm.

```
void DetectDeadlock(object targetLock)
{
    Dictionary<object, Thread> lockOwners = /*get shared list*/;
    Dictionary<Thread, object> waitingFors = /*get shared list*/;

    // Create a queue to contain threads waiting for locks:
    Queue<WaitPair> waitGraph = new Queue<WaitPair>();

    // Add the current thread to the list of threads already seen.
    WaitPair current = new WaitPair(Thread.CurrentThread, targetLock);
    waitGraph.Enqueue(current);

    while (true)
    {
        Thread owner;

        // If the lock is available, there is no cycle. Exit.
        if (!lockOwners.TryGetValue(current.Lock, out owner))
            return;

        // If the owner is in our wait-graph, there is a cycle.
        // The wait graph starts at the owner.
        foreach (WaitPair pair in waitGraph)
        {
            if (pair.Owner == owner)
            {
                // Deadlock found! The wait graph starts at the first
                // occurrence of 'owner' in the 'waitGraph' queue. We
                // can print diagnostics, throw an exception, etc.
                throw new Exception(...);
            }
        }

        // If the owner isn't, there is no cycle. Exit.
        object ownerWaitingOn;
        if (!waitingFors.TryGetValue(owner, out ownerWaitingOn))
            return;

        // Otherwise, add the entry to the graph, and proceed.
        current = new WaitPair(owner, ownerWaitingOn);
        waitGraph.Enqueue(current);
    }
}
```

```

struct WaitPair
{
    internal Thread Owner;
    internal object Lock;

    internal WaitPair(Thread owner, object slock) {
        Owner = owner;
        Lock = slock;
    }
}

```

We begin by creating a queue containing a single `WaitPair` entry. This first pair tracks the current thread whose attempted acquisition of `targetLock` is triggering detection to kick in. (Alternative algorithms involve starting with *all* threads that hold locks and attempting to find *any* cycle. The one shown only finds cycles that are rooted with a specific acquire. This is slightly more efficient.) We then enter a `while` loop. We omit a slight optimization for code brevity: if `targetLock` has no owner, there is no need to allocate any lists. The initial pair is stored inside a variable `current`, which will always hold the most recent pair in the wait graph.

Once inside the `while` loop, we first see whether the current pair's lock has an owner. If the lock is not held by another thread, there is no cycle and we return out of the method. Otherwise, we check whether the owner is inside the wait graph. If we've seen the thread previously, we have found a cycle and, therefore, can report a deadlock. What we do is very specific to the scenario: we may print some diagnostics and wait anyway, communicate the information through a debugger, throw an exception, and so on.

Next, if we have not found a cycle, we continue. We check what lock the owner is waiting to acquire. If the owner isn't waiting, it's making forward progress under the lock, and we can safely exit knowing there are no deadlocks. Otherwise, we produce a new pair, set it as the `current`, and add it to the wait graph. We then go back around the loop and continue until we find a deadlock or are convinced there aren't any.

In effect, we're building a graph like the one shown in Figure 11.2. The boxes indicate threads and the circles indicate locks; a line from a box to a circle means the thread is waiting for that lock, and a line from a circle to a box means that lock is owned by that particular thread.

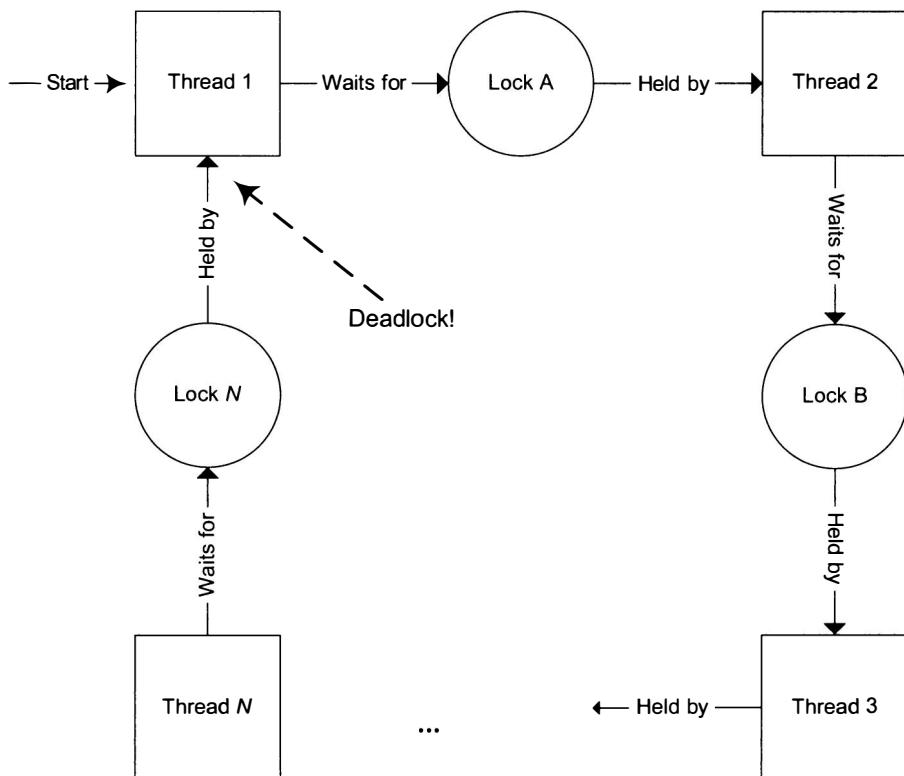


FIGURE 11.2: A wait-graph illustrating a deadlock

This is a relatively straightforward algorithm. Even if you don't have a wait graph creation algorithm in play, you can use information available from a debugger and/or from crash dumps to create a graph to aid in debugging. Even just reconstructing one on a whiteboard can be a helpful exercise for understanding difficult deadlocks.

Careful readers might have noticed a couple limitations with our algorithm. We don't ever create a true "graph," so why the name? For simplicity's sake, we have limited our algorithm so that it handles threads waiting on a single lock only; that's acceptable because common lock kinds—such as Win32 CRITICAL_SECTIONS and CLR Monitors—only support single waits. But for wait-any and wait-all style waits, the algorithm would need to be revised slightly. Finally, creating a wait graph for reader/writer locks is not

shown and somewhat more complicated because a wait graph must include both shared-mode and exclusive-mode locks.

Using Timeouts to Detect Deadlocks. A less sophisticated technique for “detecting” deadlocks is to use timeouts when acquiring locks and waiting on events. Be forewarned: this technique is often misused. It should be obvious that a deadlock is an error in the program that must be treated as a bug. By the time a piece of code is labeled “done” it should be deadlock free, even if fancy techniques such as lock leveling haven’t been used. Therefore, the use of timeouts to detect a deadlock is appropriate for debugging and testing, but not for inclusion in a production system.

By using a timeout during a synchronization wait—typically something ridiculously long like 5 seconds—you will be able to do any number of helpful things in response to a timeout. This typically involves tracing some information to a log, raising a debugger event (via `Debugger.Break` in managed code), and/or even calling the `Environment.FailFast` or `TerminateProcess` API to bring down the process and allow a dump to be captured. Which one is appropriate depends on your particular program, but this kind of checking is most often useful in debug builds and is best turned off in shipping bits.

Windows Vista Wait Chain Traversal (WCT). Windows Vista ships with a new set of Win32 APIs that fall under a single common feature, wait chain traversal, or WCT for short. WCT is meant to enable debuggers to capture wait graphs, much like what was shown earlier, in a nonintrusive way. Nonintrusive means that the debugged program need not be rewritten to support constructing an on demand wait graph: the WCT APIs gather and work with information already available in user-mode and the Windows kernel to produce a wait graph when requested to do so.

The WCT APIs also support a surprisingly rich set of wait kinds: ALPCs used for remote procedure calls, Win32 critical section acquisitions, mutex acquisitions, synchronous `SendMessage` calls for message queues, COM calls, and waits on process and thread handles. WCT does not, however, support wait-any or wait-all style waits. And it doesn’t support managed code either, because it doesn’t know about CLR monitors and other lock types. This is because these locks are built out of custom synchronization mechanisms such as interlocked operations and events. These practical limitations also mean

that WCT hasn't been integrated into popular debuggers such as Visual Studio yet.

The algorithm WCT uses is very much like the one shown before. It looks at a particular thread and, if it is blocked, figures out on what object the thread is blocked, what thread owns that object, and so on.

WCT is declared in the `Wct.h` platform header file and exposed from the `Advapi32` library. To use WCT, you'll need to first register some obscure callbacks. This is so WCT can retrieve state as needed to do with COM callbacks.

```
VOID WINAPI RegisterWaitChainCOMCallback(
    PCOGETCALLTYPESTATE CallStateCallback,
    PCOGETACTIVATIONSTATE ActivationStateCallback
);
```

Pass the addresses of `ole32.dll`'s `CoGetCallState` and `CoGetActivationState` functions, respectively. These are undocumented COM APIs, so doing this feels hacky. But it's all boilerplate and necessary for WCT to work.

Next, you must "open" a WCT session, which must be closed once you are done.

```
HWCT WINAPI OpenThreadWaitChainSession(
    DWORD Flags, PWAITCHAINCALLBACK callback
);
VOID WINAPI CloseThreadWaitChainSession(HWCT WctHandle);
```

The `OpenThreadWaitChainSession` returns a handle to the WCT session that can be used to close it later and to retrieve wait chain information from particular threads. Retrieving a wait chain for a particular thread is done with `GetThreadWaitChain`.

```
BOOL WINAPI GetThreadWaitChain(
    HWCT WctHandle,
    DWORD_PTR Context,
    DWORD Flags,
    DWORD ThreadId,
    LPDWORD NodeCount,
    PWAITCHAIN_NODE_INFO NodeInfoArray,
    LPBOOL IsCycle
);
```

The `ThreadId` parameter indicates which thread's chain is to be computed. `Context` is only used for asynchronous retrieval and is an opaque value that

is passed to the callback (shown soon). By default, WCT only computes wait chains within a single process; you can pass specific values in Flags to indicate out of process information is desired too. Specifying `WCTP_GETINFO_ALL_FLAGS` is the easiest way to do this, although you can specify three independent flags if you want to select only some: `WCT_OUT_OF_PROC_COM_FLAG`, `WCT_OUT_OF_PROC_CS_FLAG`, and `WCT_OUT_OF_PROC_FLAG`. Finally, the three pointers, `NodeCount`, `NodeInfoArray`, and `IsCycle` are used to communicate the wait chain. `NodeCount` is used for input too: it specifies the maximum chain to retrieve, and `NodeInfoArray` must be sized to receive a chain of at least that size. `WCT_MAX_NODE_COUNT` is the maximum chain length supported by WCT.

WCT supports both retrieving wait chains synchronously or asynchronously. Most people will use the former, where all wait chain information is computed and returned from calls to `GetThreadWaitChain`. For this, pass `0` for `Flags` and `NULL` for `callback` to the `OpenThreadWaitChainSession` function. This causes WCT to return the wait chain information in the aforementioned pointer arguments to `GetThreadWaitChain`. The support for asynchronous retrieval delivers the wait chain information in a callback instead of in these arguments. To use this style instead, pass `WCT_ASYNC_OPEN_FLAG` for `Flags` and a function to receive the wait chain as `callback` to `OpenThreadWaitChainSession`.

```
VOID Callback WaitChainCallback(
    HWCT WctHandle,
    DWORD_PTR Context,
    DWORD CallbackStatus,
    LPDWORD NodeCount,
    PWAITCHAIN_NODE_INFO NodeInfoArray,
    LPBOOL IsCycle
);
```

In summary, the `NodeInfoArray` is an array of `WAITCHAIN_NODE_INFO`s passed in to `GetThreadWaitChain` that is to retrieve the full wait chain. The length of the computed chain is provided as `NodeCount`. The `IsCycle` `BOOL` is set to `TRUE` if a deadlock was found and `FALSE` otherwise. The `WAITCHAIN_NODE_INFO` structure is defined as follows.

```
typedef struct _WAITCHAIN_NODE_INFO
{
    WCT_OBJECT_TYPE ObjectType;
```

```
WCT_OBJECT_STATUS ObjectStatus;
union {
    struct
    {
        WCHAR ObjectName[WCT_OBJNAME_LENGTH];
        LARGE_INTEGER Timeout;
        BOOL Alertable;
    } LockObject;
    struct
    {
        DWORD ProcessId;
        DWORD ThreadId;
        DWORD WaitTime;
        DWORD ContextSwitches;
    } ThreadObject;
};
} WAITCHAIN_NODE_INFO, *PWAITCHAIN_NODE_INFO;
```

We won't spend much time on the details here. Please consult the platform header files and SDK documentation for the finer points. But it's worth pointing out that `ObjectType` captures the kind of object a node represents: it will be either one of the kinds of wait constructs mentioned above or a thread object. This might be initially surprising. But the wait chain is a sequence of alternating thread and wait object pairs: a thread is followed by the object for which it waits, which is followed by the thread that owns that object, and so on. This is exactly like the wait graphs we looked at earlier. All the other fields are meant to provide additional diagnostics information about the kind of wait being performed.

Missed Wake-Ups (a.k.a. Missed Pulses)

We looked at condition variable and freeform event mechanisms back in Chapter 6, Data and Control Synchronization. The most common form of misuse when it comes to such facilities is the so-called **missed wake-up**, where some thread signals that a certain condition has arisen with the intent of waking up one or more other threads, and for some reason the signal does not correctly reach all of the intended recipients. Those intended recipients often end up waiting for the signal, which, because it was missed, leads to an indefinite wait. The result is the same as a deadlock: your program—or at least some of the threads within it—hang.

There are myriad ways that this can happen. Let's look at two of them.

The proper use of a condition variable is to test some condition having to do with program state from inside of a lock before waiting. Due to the possibility of **spurious wake-ups**, checking the condition typically utilizes a while loop instead of a simple if statement. The common structure of such regions of code looks like this (using CLR monitors as an exemplar).

```
object someLock = ...;
...
lock (someLock)
{
    while (!p)
        Monitor.Wait(someLock);
    ... p holds true ...
}
```

The `p` part in this code stands in for any predicate that involves reading state protected by `someLock`. If it is found to be `false`, we issue a wait. Some other thread in the system will subsequently cause `p` to become `true` and, in doing so, issues a `Pulse` or `PulseAll` to wake up any threads waiting for the condition.

Let's imagine for a moment that `p` is a one time thing. That is, once it becomes `true`, it remains `true` forever. This is called a **latch**. It is wasteful to continuously acquire the lock around the evaluation of the condition—imagine `p` is a single `bool`, and so it's safe to read atomically outside of a lock—so we might decide to do something like this:

```
object someLock = ...;
...
if (!p)
{
    lock (someLock)
        Monitor.Wait(someLock); // bad! deadlock-prone.
}
```

This should raise red flags. The first problem is that we're not abiding by the protection against spurious wake-ups, as shown earlier. Even if we fix that and change our `if` statement to a `while` loop, this code can hang forever. Thread `t1` might read `p` as `false` and immediately afterwards thread `t2` might make it `true` and call `PulseAll`. Next, `t1` acquires `someLock` and calls `Wait`. Because condition variables are not “sticky” as is a windows

kernel manual-reset event, this leads to a missed wake-up. Thread t1 is doomed to wait forever. And figuring out what has happened by looking at the state of the system after the fact is likely to lead to nothing but confusion. The condition p will be true, and other threads may have been awoken properly. Only by carefully inspecting the code will you determine the root cause of the problem.

The solution is simple. In this kind of circumstance, we can check the condition outside of the lock. But once we enter, we must recheck it via double-checked locking.

```
object someLock = ...;
...
if (!p)
{
    lock (someLock)
    {
        while (!p)
            Monitor.Wait(someLock);
    }
}
```

Another common and similar problem often leads to missed wake-ups. Windows kernel events may only be in one of two states: signaled or nonsignaled. If you set one multiple times, there is no notion of “multiple signals” as with a semaphore. Particularly when it comes to auto-reset events, it’s a common mistake to signal an auto-reset event more than once, expecting each signal to result in a single thread awakening. We will see a real occurrence of this lost signal problem in Chapter 12, Parallel Containers.

Say we had a naïve algorithm to synchronize access to a buffer with a finite number of elements within it. When empty, a consumer should wait until a producer has placed an item of interest into the buffer. It’s unlikely that real code would be written this way (one would hope), but imagine somebody coded up the synchronization like this.

```
AutoResetEvent m_itemAvailable = new AutoResetEvent(false);
Queue<T> m_items = new Queue<T>();
...
void Add(T item)
{
    lock (m_items)
```

```

    {
        m_items.Enqueue(item);
        m_itemAvailable.Set();
    }
}
T Remove()
{
    while (true)
    {
        lock (m_items)
        {
            if (m_items.Count > 0)
                return m_items.Dequeue();
        }
        m_itemAvailable.WaitOne(); // Bad! Deadlock prone!
    }
}

```

What is the intended behavior of this code? When adding an item, we use the `Enqueue` method on `Queue<T>` inside of a lock region, and call `Set` on the `AutoResetEvent`, ensuring it is signaled and that a single thread waiting for an element is awakened. When removing an item, we check the `Count` of the `Queue<T>` inside of a lock and, if empty, exit the lock and call `WaitOne` on the event. Once an element becomes available, we will wake up and loop around to remove it. There are obvious races here that lead to unfairness, so if we're awakened and lose the race, you'd think we will just rewait for the next element.

However, imagine two threads `t1` and `t2` call `Remove`, and both end up context switched out right after releasing the lock but before getting to calling `WaitOne`. Now some thread `t3` calls `Add` twice, placing two elements in the queue and calling `Set` on the event twice. Recall that the second call to `Set` is effectively ignored since the event was already signaled. Now when `t1` resumes and calls `WaitOne`, it wakes up right away and transitions the auto-reset event back into the unsignaled state. It loops around and snags one of the two items out of the queue. Now `t2` resumes and also calls `WaitOne`. It blocks even though an item is in the queue for it. If no other threads add elements to the queue or come back for the last remaining item, the system is locked up, items may be dropped, and threads may hang.

Other problems can lead to event signals being missed. Even if both threads had called `WaitOne` by the time `t3` added its two items, event signals

could get missed. This is because, as was explained back in Chapter 5, Windows Kernel Synchronization, operations such as interrupts and APCs can cause a thread to temporarily remove and re-add itself from and to the wait queue.

This particular issue is tricky because we must exit the lock before waiting. The coding pattern becomes simpler with condition variables because they address this very situation.

Livelocks

A **livelock**, as its name implies, is a condition in which threads get “locked up.” Livelocks are a lot like a deadlock, hence the similarity in name, but lead to “busy” waits rather than stalls and are more often finite in duration (at least statistically speaking). Everybody has probably encountered a situation akin to a livelock in real life: just think of the last time you were walking down a hallway in the opposite of another individual; as they approach, you realize you must step to the right or left to avoid collision; they also realize the same; they first choose right, and you choose left; both of you realize this won’t work, and reverse your direction, to no avail; this pattern is apt to repeat a few times until something gives. This is a lot like livelock, where multiple threads collide but politely try to get out of each other’s way.

Livelock commonly happens in low-level concurrency algorithms that involve optimistic concurrency and/or spin-waiting. A loop is usually involved. And often they can manifest as a single thread being livelocked versus a whole set of threads being livelocked simultaneously, although both situations are possible. Nonblocking code such as the lock free algorithms we took a look at in the last chapter trade off deadlock for livelock.

As an example of a livelock prone piece of code, say that many threads are trying to increment a shared counter using `Interlocked.CompareExchange`:

```
static volatile int s_counter = ...;
...
int c;
do
{
    c = s_counter;
}
while (Interlocked.CompareExchange(ref s_counter, c + 1, c) != c);
```

Under extreme circumstances, one or more threads could be locked out (i.e., livelocked).

T	t1	t2
0	c = s_counter (0)	
1		c = s_counter (1)
2		CompareExchange(0, 1) (success)
3	CompareExchange(0, 1) (fail!)	
4	c = s_counter (1)	
5		c = s_counter (1)
6		CompareExchange(1, 2) (success)
7	CompareExchange(1, 2) (fail!)	
8	c = s_counter (2)	
	...	

In this example, t1 keeps getting beat out by t2, leading to it retrying over and over again. While it's unlikely such extreme examples would arise, the example does illustrate the point.

This is an example that only results in a single thread being livelocked. One can easily imagine situations where two threads are cooperating and both end up backing off voluntarily to retry some operation. Imagine if we implemented the simultaneous lock acquisition code earlier by trying to acquire locks in the order supplied. If one thread tried to acquire lock A and then B, while another tried to acquire lock B and then A, deadlock could occur. To cope, we might use timeouts and "roll back" successful acquisitions upon contention; we then spin briefly and try again. If all threads participate in this scheme, they may interfere with one another, back off, retry, interfere yet again, and so on, indefinitely.

In both cases, threads use up a lot of processor time without making any true forward progress. This can result in hard to explain delays in processing and drops in throughput.

Livelock is just a fact of life. Algorithms deep down in the Windows OS and in the CLR suffer from these kinds of issues. They rely on the fact that, probabilistic speaking, indefinite livelock will not happen. There are too many subtle timing issues involved in order to produce most indefinite livelocks: cache misses, context switches, background services, foreground applications, disk and memory access latencies, and the like.

That said, **randomized backoff** is a popular technique that decreases the chances even further of a thread being indefinitely delayed. This is a

technique we explore in Chapter 14, Performance and Scalability, when looking at spin wait algorithms. The idea is that, upon failure and before retrying an operation, a thread spins for a random amount of time. Moreover, for each failed attempt at an operation, the amount of spin delay used will be increased. Provided that all threads in the system cooperate by using the same backoff logic, the chance of having many threads enter a true livelock situation is rare.

Lock Convoys

Lock convoys are situations where the arrival rate for a lock is high compared to the release rate. Convoys can have a dramatic impact on scalability, leading to threads being backed up waiting for a lock (or event) and, in many cases, a substantial drop in throughput. A convoy is most often due to a fundamental architectural problem in a system, but can also be exacerbated by the implementation of synchronization primitives as well as runtime and OS features.

Two conditions are typically involved when a convoy occurs.

- The arrival rate for some lock is high. In other words, a nontrivial amount of the program's execution happens under the protection of a particular lock.
- The hold time for that same lock is also high. In other words, once a thread acquires the lock, it doesn't release it for some a lengthy period of time.

Some simple mathematics can be used to describe the problem. Imagine the arrival rate for a lock is 1 thread/10,000 cycles. If the average lock hold time is any higher than 10,000 cycles, a convoy will ensue, and threads will arrive more frequently than locks are granted. Imagine the average hold time is also exactly 10,000 cycles. The system will be perfectly balanced in a sense and in theory, but in practice, random delays due to cache misses and page faults can throw this balance out of whack without notice. One thread holding the lock for 15,000 cycles is enough to cause the wait queue to grow. Unless a subsequent thread holds the lock for 5,000 or less cycles to offset this balance (or the arrival rate slows), we will not recover

the time lost. Once a convoy occurs, and the wait queue for a lock grows in length, the effects tend to snowball quickly. Convoys are known for bringing servers to their knees.

Fair locks often worsen convoys. This was mentioned in Chapter 5, Windows Kernel Synchronization. A fair lock guarantees that threads are given access to the lock in FIFO order, even when contention occurs. The reason fairness exacerbates convoys is subtle. As before, imagine some lock's arrival rate is 1 thread/10,000 cycles. Imagine that each thread holds the lock for 2,000 cycles. Because the arrival rate is far lower than the lock hold time, we expect that threads usually don't have to wait for the lock. Occasionally a thread will block—this is, after all, just an average—but we expect the throughput of the system to be quite good and the occurrence of convoys to be low.

Unfortunately, a fair lock can destroy this assumption. Say we get into a situation where two threads, t1 and t2, arrive at the lock simultaneously. Then t1 acquires the lock, and subsequent threads trying to acquire the lock must wait, including t2. To ensure fairness, we must ensure that when t1 releases the lock thread t2 gets it next. Unfortunately, this takes time. Because t2 has blocked, there is a delay between the time t1 releases the lock and t2 may actually enter its critical region and do useful work. How long is that delay? It's at least the cost of a context switch (more if t2 hadn't finished waiting, there are more threads in the runnable queue, and so forth); and recall that context switches can cost around 10,000 cycles on modern processors. This makes it look as though a thread holds the lock for 12,000 cycles instead of 2,000 when contention is involved. If the arrival time is 1 thread/10,000 cycles, our system will scale very poorly. All it takes is a single thread blocking to trash the entire system.

Windows has historically used fair locks almost exclusively. That includes deep in kernel and also in user-mode synchronization primitives, such as critical sections, mutexes, and events. This is the most main reason Windows uses **priority boosting** on the recipients of a signaled event: to try to minimize the amount of time between a lock becoming available and when the thread waiting on it actually wakes up, lessening the likelihood of convoys.

Much of this has changed in Windows Vista (and Windows Server 2003 R2). The bulk of the synchronization primitives are now unfair, including critical sections, mutexes, internal pushlocks, and SRWLocks. What does this mean? When released, a single waiting thread will be awakened (still in a FIFO fashion due to events maintaining wait list in FIFO order) as before, but any thread that attempts to acquire the lock before that awakened thread has successfully acquired will be granted. The wakened thread has to contend for the lock. If it fails to acquire, it must rewait and go back to the tail of the wait list. It will get another shot at the lock eventually.

Stampeding

The choice between wake-one (which wakes at most a single waiting thread) vs. wake-all (which wakes all currently waiting threads) arises when using any of the control synchronization primitives we've reviewed in previous chapters. Table 11.1 provides a refresher on this.

Often the decision to use wake-one is motivated by scalability. By choosing a wake-one style operation, however, you need to be certain of a few conditions. Specifically, you must be in a situation where the possibility that some portion of the waiting threads definitely needn't be alerted to the change in circumstance. Not being sure of this can lead to missed wake-ups.

Since we've already established that fairness can lead to convoys, most synchronization primitives provided are unfair. That unfairness has some negative effects: the most obvious one is that it can lead to starvation; the less obvious one is that it leads to wasted work. Threads awakened that fail to acquire the resource for which they have been awakened will have to

TABLE 11.1: Wake-one vs. wake-all with common synchronization primitives

Primitive	Wake-One	Wake-All
Kernel event objects	Auto-reset Set/SetEvent	Manual-reset Set/SetEvent
Monitors	Pulse	PulseAll
Win32 condition variables	WakeConditionVariable	WakeAllCondition-Variable

rewait and do it over again at some point. That incurs at least two context switches, each of which is roughly 10,000 cycles. And priority boosting can increase the chances of those threads actively preempting another. On a single processor machine, the priority boost typically has the intended effect: since there's only one processor, it's very likely that allowing the thread access to the sole processor will ensure it acquires the resource. But on multi-processor machines, there are plenty of other processors to run code in the 10,000 cycles or so that it takes for the awakened thread to context switch back in, in which time other threads may fend for the resource.

A **stampede** is the extreme case of this problem. This occurs when many threads fight for a shared resource, and when only some of them can actually win. As an example, imagine that critical regions used a manual-reset event internally (unlike the auto-reset event that they actually use); whenever the lock became available, all of the waiting threads would be awakened. All but one of them will immediately find that they cannot acquire the lock and must instead go back and wait. Ignoring the fundamentally bleak outlook of the scenario to begin with, if we have 100 threads waiting for a single lock, this approach is going to wreak havoc. One hundred threads will be awakened, preempt other (useful) threads, drag a data into the caches, fight for cache lines, and waste thousands upon thousands of cycles of processor time that could have been used to make forward progress. And yet only one of them will ultimately acquire the lock; the rest will have to rewait.

Stampedes are often a sign of a wake-all being used when wake-one would have been a better choice. Often this is done because there is no other reasonable way to implement an algorithm. For example, an interviewing question I often use is “implement a counting semaphore.” Those unlucky interviewees who first choose to use interlocked operations and Windows events run into a tradeoff between the possibility of missed pulses and the possibility of stampedes. This tradeoff is not uncommon.

Two-Step Dance

This section could have been called the N-Step Dance, but the most common value for N is 2, hence the name I've chosen for this section. This problem occurs when an event that indicates a resource is available is set prematurely, possibly waking a thread before the resource is available. The

practical outcome of this is that the awakened thread must go back to sleep for a small amount of time only to be awakened again later.

The most common example of this involves a critical region and an event.

```
object syncLock = ...;
AutoResetEvent are = ...;
...

void Producer()
{
    lock (syncLock)
    {
        // Produce some data of interest
        are.Set();
    }
}

void Consumer()
{
    are.WaitOne();
    lock (syncLock)
    {
        // Consume the data
    }
}
```

In this simplistic example, the producer sets an event while it still holds the lock on `syncLock`. The first thing the consumer does when it wakes up from waiting on the event is to attempt to acquire `syncLock`. Since the producer still holds `syncLock` at this time, its attempt will fail and it will have to wait again. When the producer finally releases `syncLock`, the lock will internally signal the consumer thread to wake up and acquire the lock.

There's a lot of wasted work going on here. In the worst case, the consumer incurs four context switches: one to wait on the event, one to wake up from the wait, another to wait on the lock, and the last one to wake up from waiting on the lock. And it gets worse. On a single processor system, due to priority boosting, you're just about guaranteed that the consumer thread will preempt the producer thread when it wakes up the first time. This adds to the delay.

Most two-step dance problems are due to fundamental race conditions that are hard to avoid and lead to setting events with locks still held. Sometimes they are caused by holding multiple locks at once. And the problem

is fairly widespread too: CLR Monitor's `Wait/Pulse/PulseAll` inherently suffer from this, as do Windows Vista's condition variables. For example, when `Monitor.Pulse` is called, an internal CLR-managed event is set, and a waiting thread is allowed to wake up immediately. The first thing the thread that called `Wait` must do is reacquire the lock; and yet it's still held by the thread calling `Pulse`. This is fundamentally a problem with the API since `Pulse` may only be called with the lock held.

Priority Inversion and Starvation

A phenomenon called **priority inversion** can lead to a thread's priority being artificially increased because the lower priority thread holds on to a shared resource—normally a lock—that a higher priority thread needs to access. This can lead to a lower priority thread getting more than its fair share of processor time, compared to what the thread scheduling logic would have ordinarily allotted. In effect, the priorities have been inverted, hence the name.

Priority inversion can be worsened by having a third middle priority thread, leading to a related problem called **starvation**. If this middle priority thread preempts the lower priority one, then the lower priority thread may not get a chance to run to completion and release the lock. Imagine there's a continuous stream of middle priority work; the Windows thread scheduler by default will continue to give the highest runnable threads access to the processors, and so the high priority thread could be starved of processor time indefinitely.

Priority inversion and starvation are possible without needing the standard definition of a shared resource: imagine some higher priority thread is waiting for an event to be set by a lower priority thread. That higher priority thread might decide to spin-wait for a bit of time, to avoid needing to context switch. This is foolish, since spinning takes processor time and the Windows thread scheduler will view the higher priority thread's spinning as real work. Even if the higher priority thread calls `Sleep(0)` to let another thread run, the problem may persist. Calling `Sleep` with an argument of `0` only considers other threads of equal priority, so the lower priority thread will be skipped. A combination of `SwitchToThread` and `Sleep(1)` must be used instead (see Further Reading, Duffy, August 2006). This is a common problem with custom

spin locks. We'll look at how to properly write spin-waits in Chapter 14, Performance and Scalability.

Starving high priority work is a real problem, especially in real time or mission critical systems, where some background processing interferes with a more important time sensitive operation. This is one reason that changing thread priorities should be (mostly) avoided, unless you have a very compelling reason to do so.

Windows has a system thread called the **balance set manager**, whose job mainly centers around management of virtual memory tables. But another one of its responsibilities includes rudimentary starvation management. It wakes up once a second, and, if a particular thread has not run for 4 seconds, it temporarily boosts that thread's priority to "time critical" (priority level 15—the highest dynamic thread priority without entering real time) and the thread also enjoys a quantum boost so that it runs for twice the ordinary quantum length on client SKUs and four times on server SKUs. Priority decays at each quantum, until the thread reaches its original priority again. This virtually guarantees that the thread will get a chance to run soon and, in the case of priority inversion, long enough to release its lock. But then again, 4 seconds is a long time to wait for the starvation to kick in, so even with this support, priority inversion and starvation are problems.

Many alternative solutions to starvation are possible. The kernel uses IRQLs to prevent interrupts, including context switches, during some critical regions. This technique isn't available to user-mode code. Other solutions are known in the literature but aren't currently used by the Windows kernel; one such technique is **priority inheritance**, where the priority of a thread holding a shared resource is temporarily boosted to equal that of another thread that needs access to the shared resource (until it has been relinquished) (see Further Reading, Sha, Rajkumar, Lehoczky). You could build such a scheme in user-mode, but lack of support for priority inheritance is one of several often cited reasons why NT is generally insufficient as a real-time or embedded OS.

Where Are We?

In this chapter, we switched our focus from the mechanics and techniques useful for building concurrent programs to the kinds of hazards that plague

them. We've looked at two broad categories of hazards: correctness and liveness. The presence of such a hazard is usually best treated like a bug that should be found and fixed—along with other ordinary bugs—before shipping your software. Along the way, we've seen some ways to avoid or mitigate these errors.

The term "hazard" is certainly appropriate. Some of the most famous bugs that slipped into production software have been due to concurrency. A few examples.

- In 1985 through 1987, six massive overdoses of radiation were administered to therapy patients via the Therac-25 machine. The dosage was about 100 times the expected amount. This incident lead to three of the affected patients dying and the others were left with serious injuries. Many root causes have been identified, but a major cause was the presence of a race condition between the operator's input and the processing of that input (see Further Reading, Leveson, Turner).
- On August 14th, 2003, a massive power outage plagued the north-eastern and Midwestern U.S., in addition to Ontario, Canada. This was the largest blackout in U.S. history, affected 50 million people, and resulted in approximately \$6 billion USD in financial losses. The root cause as to why the software system did not respond correctly was also race condition (see Further Reading, Poulsen).
- In 1997, the Mars Pathfinder mission launched a rover to Mars with the aim of collecting meteorological data. It did this, but not without a large number of software hiccups within the first few days after landing. Due to a software bug that eluded testing, the rover encountered a situation that caused it to continuously experience total system resets, losing data in the process. These problems made the news and were eventually attributed to priority inversion (see Further Reading, Reeves).

Any software bug that goes unnoticed can be just as deadly as any of these. But as has been noted several times already, concurrency bugs more easily slip through the cracks due to the difficulty of testing for them.

In subsequent chapters we will look at some common data structures and patterns for using concurrency. We'll look at Parallel Containers in Chapter 12,

which are useful for any concurrent program manipulating data (nearly all of them) and Data and Task Parallelism in Chapter 13, which illustrates common uses of parallelism. In addition to careful testing, following common practices can help reduce the occurrence of concurrency errors.

FURTHER READING

- M. Abadi, C. Flanagan, S. N. Freund. Types for Safe Locking: Static Race Detection for Java. In *ACM Transactions on Programming Languages and Systems*, Vol. 28, No. 2 (2006).
- M. Barnett, K. R. M. Leino, W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*, LNCS, Vol. 3362 (Springer, 2004).
- C. Brumme. Apartments and Pumping in the CLR. Weblog article, <http://blogs.msdn.com/cbrumme/archive/2004/02/02/66219.aspx> (February 2004).
- L. T. Chen. The Challenge of Race Conditions in Parallel Programming (Sun Developer Network, 2006).
- E. G. Coffman, Jr., M. L. Elphick, A. Shoshani. System Deadlocks. In *Computing Surveys*, Vol. 3, No. 2 (1971).
- E. W. Dijkstra. EWD310: Hierarchical Ordering of Sequential Processes. In *Acta Informatica*, 1(2) (1971).
- E. W. Dijkstra. EWD 623: The Mathematics Behind the Banker's Algorithm. In *Selected Writings on Computing: A Personal Perspective* (Springer-Verlag, 1982).
- J. Duffy. No More Hangs: Advanced Techniques to Avoid and Detect Deadlocks in .NET Apps. *MSDN Magazine* (2006).
- J. Duffy. Priority-Induced Starvation: Why Sleep(1) is Better than Sleep(0); and the Windows Balance Set Manager. Weblog article, <http://www.bluebytesoftware.com/blog/2006/08/23/PriorityinducedStarvationWhySleep1IsBetterThanSleep0AndTheWindowsBalanceSetManager.aspx> (2006).
- N. Leveson, C. S. Turner. An Investigation of the Therac-25 Accidents. In *IEEE Computer*, Vol. 26, No. 7 (1993).
- B. Meyer. An Eiffel Tutorial: Interactive Software Engineering. <http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial-00.html>.
- M. Pietrek and R. Osterlund. Threading: Break Free of Code Deadlocks in Critical Sections Under Windows. *MSDN Magazine* (2003).

- K. Poulsen. Tracking the Blackout Bug. *SecurityFocus*, <http://www.securityfocus.com/news/8412> (2004).
- G. Reeves. What Really Happened on Mars? http://research.microsoft.com/~mbj/Mars_Pathfinder/Authoritative_Account.html (1997).
- J. Robbins. Buslayer: Wait Chain Traversal. *MSDN Magazine* (2007).
- S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In *ACM Transactions on Computer Systems*, Vol. 15, No. 4 (1997).
- L. Sha, R. Rajkumar, J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. In *IEEE Transactions on Computers*, Vol. 39 (1980).
- S. Toub. .NET Matters: Deadlock Monitor. *MSDN Magazine* (2007).
- Y. Yu, T. Rodeheffer, W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the ACM Symposium on Operating System Principles*, SOSP'05 (2005).

12

Parallel Containers

EVERY PROGRAM NEEDS containers to hold interesting data. And while it's not necessarily always true that all parallel programs need parallel containers, frequently they do. A parallel container usually differs from ordinary sequential ones—such as those available in the C++ Standard Template Library (STL) or .NET's `System.Collections.Generic` namespace—in several ways:

- The container provides **scalable access**. Ordinary containers are usually not safe for concurrent access. And even if they are, most general purpose libraries that offer containers safe for concurrent access favor single threaded performance over scalability. This is true of the .NET 1.0 nongeneric collection types that provided “synchronized” wrappers over the same underlying sequential container. While this ensures correctness and is simple, the result does not exploit the **natural scalability** of many kinds of containers.
- The container may offer efficient **parallel traversal**. Many algorithms achieve parallelism by partitioning some data source so that many threads can do something with it at once. (This is a primary focus of the next chapter.) And that data source is often a parallel container of some sort, so having the ability to access it in a scalable way enables efficient parallel traversal.

- Some, but not all, containers provide **concurrent orchestration**. This is most common in one broad class of parallel containers: **producer/consumer containers**. These enable multiple threads to coordinate with one another using structured patterns that hide tricky synchronization behind a simple and familiar container oriented interface, such as a blocking or bounded queue.

In order to provide these properties, many of the techniques from past chapters must be used. That includes synchronization primitives (Chapters 5 and 6), lock free programming (Chapter 10), and, an awareness of concurrency hazards (Chapter 11). Not only is this fairly extensive background necessary, but there are multiple approaches from which to choose.

1. **Coarse-grained locking** is the easiest scheme to implement. A single lock per container is used, and all read/write operations acquire this single lock. This guarantees contention any time more than one thread accesses the same container. This is what sequential oriented libraries typically provide because scalability is a distant concern. Scalability can be improved by using coarse-grained reader/writer locks instead of mutually exclusive locks—especially when reads outnumber writes, which is often the case—but often not satisfactorily.
2. **Fine-grained locking** is advantageous when the data structure can be broken into distinct pieces. Only threads that access the same piece at the same time will experience contention. Such a scheme can take two forms: associating locks with actual parts of the data structure, such as individual nodes in a linked list, or by having some kind of mapping from an arbitrary part to a set of collection-wide locks. How you'd do the first is probably obvious—although having low overhead locks, such as single word spin locks, becomes more important—but the second approach may be less obvious. **Striping** is the most commonly used technique, enabling you to have fewer locks than pieces.

To illustrate striping, a structure with P pieces will have L locks, and when a thread needs to access a particular piece of the structure, p_n , it just acquires lock number $p_n \% L$. (“Piece” has different meanings for

different kinds of containers: a node in a linked list, element in an array, a bucket in a hashtable, and so forth; how fine to go is a design choice.) L can be sized based on expected concurrency levels, eliminating the single bottleneck and reducing contention. To make this idea more concrete, imagine we have an array of 2,048 elements protected by 16 locks. Accessing the 1,077th element means we have to acquire lock number 5 (i.e., $(1077 \% 16) == 5$). Alternative schemes for assigning locks can be used to reduce **false contention**; this happens when two threads access logically disjointed parts of the structure but share a lock by coincidence because of the specific piece-to-lock mapping scheme chosen.

While fine-grained locking provides better scalability, having multiple locks for a single container can introduce complexities. It increases the storage and management of OS resources required for a single instance. And it also complicates the implementation because we must be careful to acquire locks in the right order so as not to deadlock. Globally impactful operations such as resizing and clearing the container will often require acquiring more than one lock before proceeding, and enumeration is tricky. If these are common operations, the resulting cost can be dramatically higher than the corresponding implementation using coarse-grained locking.

3. **Nonblocking**, a.k.a. **lock free**, techniques can be used to avoid locks altogether. This approach usually carries many of the same benefits of fine-grained locking without some of the aforementioned challenges. But it often means changing the layout of a container's storage, such as using a linked list for storage instead of an array, as we saw with the lock free stack shown in Chapter 10, Memory Models and Lock Freedom. This is sometimes not optimal for sequential code, although it can improve high-end scalability. Such lockless data structures also require extreme care to implement and sometimes must resort to trickery and spinning in corner cases (particularly for global operations such as resizing).

The choice between these three must be made based on the performance and scalability requirements of your code. And the choice is often not

obvious until you've put a fair bit of engineering work into making a decision. A wise decision, however, is to start at the top and move your way down to the bottom: coarse-grained locking first. If your container is not a bottleneck in the program—or most access is read-only and can be protected by a reader/writer lock—you will save a lot of time by choosing the simplest approach first. Next, try fine-grained locking. For simple containers, this approach usually reduces a sizeable amount of contention. Only after exhausting those approaches should you go down the lock free data structure route.

With all these generalities, let's review some real parallel collection implementations. Most of them will be written in C# and .NET for consistency's sake. We'll skip the coarse-grained implementations—since they are obvious and can be built by wrapping access to ordinary STL or .NET containers with locks—and focus on fine-grained and, sometimes, lock free approaches. This includes linked lists, queues, and dictionaries. A few specialty containers are also dissected along the way: work stealing queues used for concurrency scheduling and a few producer/consumer containers.

Fine-Grained Locking

We will begin by looking at some containers that use fine-grained locking.

Arrays

A program can safely read from or write to an array that contains word sized elements (i.e., the size of a pointer) that have been perfectly aligned (i.e., no two elements span a contiguous pointer sized chunk of memory) without any additional synchronization. This is because the hardware ensures such memory operations are atomic. If the elements are larger than this or not properly aligned, locking will be needed. Adding fine-grained locking to an array is somewhat trivial. We just divide the array up into chunks and assign a unique lock to each unique chunk, or alternatively use striping. The design looks a lot like arrays that are partitioned for purposes of data parallelism, as we will see in Chapter 13, Data and Task Parallelism.

FIFO Queue

Using fine-grained locking for a LIFO stack makes little sense. Stacks typically don't support random access, so concurrency is inherently limited by the single head of the stack that must be manipulated in order to push or pop. FIFO queues, on the other hand, have two ends: enqueues go to one, and dequeues go to another. There is a natural way to achieve better concurrency with fine-grained locks: use two locks, one for each end.

This approach is correct but can be deadlock prone. There are plenty of ways to build a queue, but a common way is to use a linked list. In such cases, there would be two fields, one referring to the head and the other the tail. Most of the time operations are completely independent. But when the queue becomes small, it may be necessary to acquire both locks. And, in fact, the logic (which appears simple at first) quickly becomes complicated. For instance, when the first node is enqueued, both head and tail must point to it; and similarly, when the last node is dequeued, both head and tail must be changed to null. Ensuring both threads notice each other's progress around empty/nonempty is difficult. Here is where the logic can become deadlock prone: for example, the enqueueer acquires its lock first, then sees it must acquire the other; similarly, the dequeuer acquires its lock first, then sees it must acquire the other; neither will proceed from here. We can work around this by having one of the threads first back off and then acquire the opposite lock, so that all threads acquire locks in the same order if both must be held. But there is a simpler way.

The simpler solution to this problem is to use a sentinel node to represent an empty queue. Thus we never have to worry about two threads operating on separate shared locations. It is true that a dequeuing thread will read an enqueueing thread's writes (e.g., the next pointer), but this can be done in a safe way as long as the write of the node's value is done first. For example:

```
public class FineGrainedLinkedListQueue<T>
{
    class Node
    {
        internal T m_val;
        internal Node m_next;
    }
}
```

```

private Node m_head;
private Node m_tail;
private object m_enqLock = new object();
private object m_deqLock = new object();

public FineGrainedLinkedQueue()
{
    m_head = m_tail = new Node();
}

public void Enqueue(T obj)
{
    Node n = new Node();
    n.m_val = obj;

    lock (m_enqLock)
    {
        m_tail.m_next = n;
        m_tail = n;
    }
}

public T Dequeue()
{
    T val;

    lock (m_deqLock)
    {
        Node next = m_head.m_next;

        if (next == null)
            throw new Exception("empty");

        val = next.m_val;
        m_head = next;
    }

    return val;
}
}

```

The implementation here is fairly simplistic. We have two nodes, `m_head` and `m_tail`, and two locks, `m_enqLock` for enqueueing and `m_deqLock` for dequeuing. The queue is initialized with `m_head` and `m_tail` pointing at the same sentinel node. As elements are enqueued, we acquire `m_enqLock` and change `m_tail.m_next` and `m_tail` itself to refer to the new node. As elements are dequeued, we acquire `m_deqLock` and swap the `m_head` reference

with its `m_next` pointer. When its `m_next` field is `null`, this indicates the queue is empty, ensuring that we never actually change `m_head` itself to `null`. A thread dequeuing a node that is in the middle of being enqueued serializes correctly because the `m_val` field will have been made visible (due to the fence implied by the acquisition of `m_enqLock`) in time.

Using a linked list is simpler, but has some disadvantages. The biggest one is that enqueueing creates new heap allocated objects and dequeuing creates garbage. It is less straightforward to create a fine-grained locking queue that has an array instead for storage, but certainly possible. It looks similar to the linked list version, but requires that we properly resize the queue when it becomes full.

```
public class FineGrainedQueue<T>
{
    private const int INITIAL_SIZE = 32;
    private T[] m_array = new T[INITIAL_SIZE];
    private int m_head = 0;
    private int m_tail = 0;
    private object m_enqLock = new object();
    private object m_deqLock = new object();

    public void Enqueue(T obj)
    {
        lock (m_enqLock)
        {
            int newTail = m_tail + 1;
            if (newTail == m_array.Length) newTail = 0;

            // If full, resize.
            if (newTail == m_head)
            {
                Resize();
                newTail = m_tail + 1;
                // assert: newTail != m_array.Length
                // assert: newTail != m_head
            }

            m_array[m_tail] = obj;
            m_tail = newTail;
        }
    }

    private void Resize()
    {
        // assert: m_enqLock is held.
```

```

        lock (m_deqLock)
        {
            T[] newArray = new T[m_array.Length * 2];
            Array.Copy(m_array, m_head, newArray, 0, m_array.Length - m_head);
            Array.Copy(m_array, 0, newArray, m_array.Length - m_head, m_head);
            m_array = newArray;

            if (m_tail < m_head)
                m_tail += m_array.Length - m_head;
            else
                m_tail -= m_head;

            m_head = 0;
        }
    }

    public T Dequeue()
    {
        lock (m_deqLock)
        {
            if (m_head == m_tail)
                throw new Exception("empty");

            T value = m_array[m_head];

            if (default(T) == null)
                m_array[m_head] = default(T); // mark eligible for GC

            int newHead = m_head + 1;
            if (newHead == m_array.Length) newHead = 0;
            m_head = newHead;

            return value;
        }
    }
}

```

This implementation is a standard array based queue, such as the one found in .NET. We start with an initially sized array, and whenever it becomes full we grow the array by doubling it. Most of the complicated logic is surrounding the management of `m_head` and `m_tail` (since they can wrap around) and the resizing: synchronization is actually fairly straightforward. Threads that enqueue must only acquire `m_enqLock` (unless resizing is necessary) and threads that dequeue must only acquire `m_deqLock`. We detect a full queue when the enqueueing thread would update `m_tail` such that it equals `m_head` in order to make room in the queue. In this case,

the `Enqueue` method calls `Resize` while still holding `m_enqLock`. That method then acquires `m_deqLock` and performs the resizing while holding both. When it unlocks, the queue is back in a consistent state.

There is a small benign race here that could lead to resizing when not strictly necessary: after seeing that the queue was full, any number of threads could dequeue elements before the enqueuer gets around to actually calling `Resize`. In such a case, the array would grow although there is technically now space available. To avoid this, we could recheck the full condition again after acquiring `m_deqLock`. But this is a minor optimization and adds complexity to the code base, so its value is questionable. This was brought up because it's an interesting example of the kinds of tradeoffs you will encounter in the real world, particularly for low-level data structures.

Linked Lists

We've already seen a linked list used in a context with fine-grained locking. But what if we want to provide access to arbitrary elements within such a list? This could be useful for adding and removing elements at particular locations. To do these kinds of things using fine-grained locks, we'll need to somehow lock individual nodes. For simplicity's sake, our example linked list will be a singly linked list and has a very simplistic surface area. Adds and removes from the head are allowed, and adds to the tail are allowed, all of which are $O(1)$ operations; inserts and removes are also permitted, typically requiring the use of $O(N)$ find operations, as is standard with linked lists. This can be used to create a simple dequeue, among other things.

Access to non-head and non-tail nodes works by searching for a particular value in the list. We have three relevant methods: `TryInsertAfter`, `TryInsertBefore`, and `TryRemove`, all implemented using a standard `TryFindAndPerform` method that encapsulates the tricky race free traversal logic and invokes a delegate when the sought after value has been found. (More useful interfaces are conceivable and necessary for more complicated use cases, such as maintaining a list in sorted order. This could be accommodated with a variant of `TryFindAndPerform` that used a predicate delegate that found an arbitrary position in the list, but may also require exposing the internal list nodes publicly for efficiency reasons.) In order to implement searching, we will use so-called **hand over hand locking**.

Here is the sample implementation.

```
public class FineGrainedLinkedList<T>
{
    class Node
    {
        internal T m_val;
        internal Node m_next;
    }

    private Node m_head;
    private Node m_tail;

    public FineGrainedLinkedList()
    {
        m_head = m_tail = new Node();
    }

    public void AddHead(T obj)
    {
        Node n = new Node();
        n.m_val = obj;

        while (true)
        {
            Node h = m_head;
            lock (h)
            {
                if (m_head != h) continue;
                n.m_next = h.m_next;
                h.m_next = n;
                break;
            }
        }
    }

    public T RemoveHead()
    {
        T val;

        while (true)
        {
            Node h = m_head;
            lock (h)
            {
                if (m_head != h) continue;

                if (h.m_next == null)
                    throw new Exception("empty");
            }
        }
    }
}
```

```
        Node next = h.m_next;
        val = next.m_val;
        m_head = next;
        break;
    }
}

return val;
}

public void AddTail(T obj)
{
    Node n = new Node();
    n.m_val = obj;

    while (true)
    {
        Node t = m_tail;
        lock (t)
        {
            if (m_tail != t) continue;
            t.m_next = n;
            m_tail = n;
            break;
        }
    }
}

// RemoveTail difficult w/out doubly linking. Left as an exercise.

private delegate void FindAction(Node pred, Node curr);

private bool TryFindAndPerform(T obj, FindAction action)
{
    Node pred = m_head;
    Node curr;

    Monitor.Enter(pred);
    while ((curr = pred.m_next) != null)
    {
        Monitor.Enter(curr);
        if (EqualityComparer<T>.Default.Equals(curr.m_val, obj))
        {
            action(pred, curr);
            Monitor.Exit(pred);
            Monitor.Exit(curr);
            return true;
        }
    }
}
```

```

        Monitor.Exit(pred);
        pred = curr;
    }
    Monitor.Exit(pred);

    return false;
}

public bool TryInsertAfter(T search, T toAdd)
{
    return TryFindAndPerform(search, delegate(Node pred, Node curr)
    {
        Node n = new Node();
        n.m_val = toAdd;
        n.m_next = curr.m_next;
        curr.m_next = n;
    });
}

public bool TryInsertBefore(T search, T toAdd)
{
    return TryFindAndPerform(search, delegate(Node pred, Node curr)
    {
        Node n = new Node();
        n.m_val = toAdd;
        n.m_next = curr;
        pred.m_next = n;
    });
}

public bool TryRemove(T obj)
{
    return TryFindAndPerform(obj, delegate(Node pred, Node curr)
    {
        pred.m_next = curr.m_next;
        if (m_tail == curr)
            m_tail = pred;
    });
}
}

```

AddHead, RemoveHead, and AddTail are somewhat similar in concept to the FineGrainedLinkedList<T> type's methods we saw earlier. In each case, we need to be careful when locking `m_head` or `m_tail` to ensure the fields don't change; this requires that we use while loops. The tricky method is TryFindAndPerform, used by the other Try methods. It walks the list and maintains a predecessor and current node, starting at `m_head`. The predecessor is locked,

which freezes its `m_next` reference. The `m_next` reference then becomes the current node and is locked. At this point, both the predecessor and next node are frozen, allowing us to insert before or after the current node or remove the current node. By using `EqualityComparer<T>.Default.Equals`, we determine whether we have found the element we're searching for and, if so, we invoke the action delegate, exit the locks, and return true. Otherwise, we continue the search. This entails releasing the lock on the predecessor, setting predecessor to the current, and continuing. Eventually, if we fail to find a matching element, we must remember to exit the predecessor lock.

The drawback to this approach of course is that it requires O(N) lock acquisitions to find an element. We could perform an optimization by using **optimistic concurrency**. If we avoided taking locks until we found an element of interest, we would substantially reduce the number of locks acquired during the search. This requires that we restart our search, however, if we find that something has gone awry in the meantime.

```
private bool TryFindAndPerformOptimistic(T obj, FindAction action)
{
    while (true) {
        Node pred = m_head;
        Node curr;

        while ((curr = pred.m_next) != null)
        {
            if (EqualityComparer<T>.Default.Equals(curr.m_val, obj))
            {
                lock (pred)
                {
                    lock (curr)
                    {
                        // If next pointer changed, curr was deleted.
                        if (pred.m_next != curr)
                            break;
                        // If random access updates are allowed, we must
                        // revalidate that equals still holds.
                        if (!EqualityComparer<T>.Default.Equals(
                            curr.m_val, obj))
                            break;

                        action(pred, curr);
                        return true;
                    }
                }
            }
        }
    }
}
```

```
        return true;
    }
    pred = curr;
}

if (curr == null)
    return false;
}
}
```

Notice that we defer locking until we've found a matching element. Once this happens, we acquire locks on both the predecessor and the current element, and, before invoking the action, verify that `pred.m_next` still points at `curr`. If not, we break out and continue around the outer loop; this restarts the search back at the beginning of the list. A reasonable implementation might be to fall back to the pessimistic routine (shown earlier) if one failure was reached; this prevents too many restarted attempts and wasted work. For lengthy lists this will save time spent retraversing nodes and will ensure the worst case is still $O(N)$. This is the already the best case for the pessimistic approach.

Dictionary (Hashtable)

Building an efficient hash table based dictionary is no easy task. STL offers `hash_map` and .NET offers its old `System.Collections.Hashtable` and new `System.Collections.Generic.Dictionary<TKey, TValue>` types for this purpose. When it comes to building a concurrent one, there are several algorithms from the research community that build on top of lock free sets and linked lists. Most of them tend to be very expensive in terms of the number of CAS operations incurred for simple operations such as adding, searching, and deleting. For modern Intel and AMD architectures, such algorithms tend not to perform too greatly; and, moreover, the implementations are incredibly complex. That said, they are worth understanding from a pure educational standpoint: refer to one of the papers referenced at the end of the chapter (see Further Reading, Michael, Scott, Purcell, Harris) if you are interested.

It's relatively straightforward to build a hashtable that provides two properties.

- Fine-grained locking can be implemented by striping a fixed number of locks L across a fixed number of buckets. When modifying a particular bucket b 's contents, we ensure that the thread holds the associated lock $b \% L$. This is similar to how we might create an array with fine-grained locks.
- Lock free reading can be performed when inquiring about the presence of an element in the hashtable. This is possible because the addition of an element to the hashtable is performed with a single atomic write, but does require that the node's next field is marked `volatile` (in .NET) to prevent load reordering.

It turns out the `.NET Hashtable` type actually implements thread safe reading without locks. Many .NET developers still take advantage of this (though writes still require custom synchronization). `Dictionary< TKey, TValue >`, on the other hand, does not offer any such guarantees.

We will vastly simplify our example hashtable implementation by using a naïve closed addressing based algorithm. This allows us to focus on the basic locking aspects of the data structure. That said, this choice—particularly the choice to have a fixed number of buckets—is very limiting. It also avoids needing to address some definitely interesting problems, such as how to implement resizing safely. This is left as an exercise for the motivated reader.

Before moving on, you may have wondered why we didn't populate our hashtable's buckets with `FineGrainedLinkedList< T >` objects, as defined above. We could have done so, but this may or may not be worthwhile. There is an overhead to each element incurred and we expect (for a well performing hashtable) that collisions will be rare: so having fine-grained locks within the individual buckets will probably not gain anything. It would also complicate one of our stated goals: to enable lock free reading from the contents of the buckets.

One such problem is reading lock free concurrently with a resizing operation. This can be done by optimistically reading a bucket's contents and checking afterward that a resize has not happened in the meantime. In the event that a concurrent resize occurs, we must fall back to acquiring a lock. This is easier to do in .NET because the GC prevents reclamation of memory

while outstanding references exist. It would be substantially harder to do in native C++.

Here is our very basic fixed size hashtable algorithm, in C#.

```
public class FineGrainedHashtable<K, V>
{
    class Node
    {
        internal K m_key;
        internal V m_value;
        internal volatile Node m_next;
    }

    private Node[] m_buckets;
    private object[] m_locks;
    private const int BUCKET_COUNT = 1024;

    // Constructs a new hashtable w/ concurrency level == #procs.
    public FineGrainedHashtable() : this(Environment.ProcessorCount) { }

    // Constructs a new hashtable with a particular concurrency level.
    public FineGrainedHashtable(int concurrencyLevel)
    {
        m_locks = new object[Math.Min(concurrencyLevel, BUCKET_COUNT)];
        for (int i = 0; i < m_locks.Length; i++)
            m_locks[i] = new object();
        m_buckets = new Node[BUCKET_COUNT];
    }

    // Computes the bucket and lock number for a particular key.
    private void GetBucketAndLockNo(
        K k, out int bucketNo, out int lockNo)
    {
        if (k == null)
            throw new ArgumentNullException();

        bucketNo = (k.GetHashCode() & 0xffffffff) % m_buckets.Length;
        lockNo = bucketNo % m_locks.Length;
    }

    // Adds an element.
    public void Add(K k, V v)
    {
        int bucketNo;
        int lockNo;
        GetBucketAndLockNo(k, out bucketNo, out lockNo);

        Node n = new Node();
        n.m_key = k;
```

```
n.m_value = v;

lock (m_locks[lockNo])
{
    n.m_next = m_buckets[bucketNo];
    m_buckets[bucketNo] = n;
}

// Retrieves an element (without locks), returning false not found.
public bool TryGet(K k, out V v)
{
    int bucketNo;
    int lockNoUnused;
    GetBucketAndLockNo(k, out bucketNo, out lockNoUnused);

    // We can get away w/out a lock here.
    Node n = m_buckets[bucketNo];
    Thread.MemoryBarrier();
    while (n != null)
    {
        if (n.m_key.Equals(k))
        {
            v = n.m_value;
            return true;
        }
        n = n.m_next;
    }

    v = default(V);
    return false;
}

// Retrieves an element (without locks), and throws if not found.
public V this[K k]
{
    get
    {
        V v;
        if (!TryGet(k, out v))
            throw new Exception();
        return v;
    }
}

// Removes an element under the specified key.
public bool Remove(K k, out V v)
{
    int bucketNo;
    int lockNo;
```

```

GetBucketAndLockNo(k, out bucketNo, out lockNo);

// Quick check.
if (_buckets[bucketNo] == null)
{
    v = default(V);
    return false;
}

lock (_locks[lockNo])
{
    Node nprev = null;
    Node ncurr = _buckets[bucketNo];
    while (ncurr != null)
    {
        if (ncurr.m_key.Equals(k))
        {
            if (nprev == null)
                _buckets[bucketNo] = ncurr.m_next;
            else
                nprev.m_next = ncurr.m_next;

            v = ncurr.m_value;
            return true;
        }

        nprev = ncurr;
        ncurr = ncurr.m_next;
    }
}

v = default(V);
return false;
}
}

```

Most of the implementation of `FineGrainedHashtable<K, V>` is straightforward. When the container is constructed, we create two arrays: `m_buckets`, which is fixed in size to `BUCKET_COUNT` and holds elements of type `Node` forming a linked list, and `m_locks`, which is sized based on the expected concurrency level (or `BUCKET_COUNT` if smaller). The sizing of buckets is extremely naïve; please refer to your favorite data structures book (see Further Reading, Cormen, Leiserson, Rivest, Stein) for more clever and appropriate techniques. It's generally a good practice to ensure the number of buckets is a prime number, for example, to help reduce collisions for degenerate inputs.

The `GetBucketAndLockNo` is then used in various places when the appropriate indices into `m_buckets` and `m_locks` are needed. It is implemented simply with modulus: the hash code is taken from the key, and we modulus it with the bucket count, giving us `bucketNo`; then we modulus the `bucketNo` with the lock count, giving us `lockNo`. This method also validates that the key provided is not `null`: supporting `null` keys could be done by treating them like `0s`.

When `Add` is called, it computes these indices and then allocates a new node. It takes the lock using its `lockNo` index as late as possible and pushes the new node on the front of the linked list in the appropriate bucket. We could have reasonably added it to the tail (LIFO order versus FIFO), but this could incur an $O(N)$ traversal of the bucket list. It's also worth pointing out that we might have considered a lock free stack for the buckets but that doing so would cause some issues when it comes to removing elements (since the lock free stack doesn't support random access). Some lock free hashtable algorithms use a lock free linked list to support the random access requirements.

The `Remove` method works similar to `Add`, with one interesting caveat: it checks the bucket for a `null` value (meaning it is empty) before even acquiring a lock. This is a minor optimization—and a questionable one—but is shown for illustration purposes only.

Finally, the `TryGet` and indexer methods do not acquire locks at all. The reason this works is subtle. The linearization point for adding a new element is the write to the appropriate bucket that links on a new node; and the point for removing an element is the write to the appropriate bucket or node's next pointer. Notice that the linearization point is not when the lock is released inside `Add` or `Remove`; this is an important distinction to make, because if the hashtable ever required more complicated invariants that could not be captured in a single atomic write, then the lock free reading would not work. For this to function properly, writes must also retire in order (which is guaranteed by the .NET memory model) so that a node cannot be seen with an empty key or value. Additionally, the lock free reads must occur in order too: this is accomplished by issuing an explicit `MemoryBarrier` after reading the bucket's value, and by making the subsequent reads of `m_next` fields on the nodes volatile reads.

Lock Free

We'll only review a few lock free data structures. There is a wealth of literature on building lock free linked lists, sets, hashtables, and the like—this is an area of increasingly active and ongoing research—and the aim of this book is not to present a comprehensive overview of all of them. Rather, we will see a couple illustrative examples that, coupled with the contents of Chapter 10, Memory Models and Lock Freedom, will enable you to learn more about and experiment with the current state of the art.

General-Purpose Lock Free FIFO Queue

There is a straightforward lock free queue algorithm that was popularized by Michael and Scott (see Further Reading, 1996) about a decade ago. It is somewhat similar to the fine-grained queue we saw earlier, and is effectively an extension of the lock free stack algorithm we already looked at in Chapter 10, Memory Models and Lock Freedom: nodes are the same structure, but in addition to a head reference, we also maintain a tail reference too. Enqueuing a new node places it at the tail end, and dequeuing removes from the head end. There is some subtlety around how we ensure both the head and tail pointers, plus all the next pointers in the linked chain, stay in sync. This will be explained in more detail after seeing the code.

Here is an implementation of a `LockFreeQueue<T>` class.

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

#pragma warning disable 0420

public class LockFreeQueue<T> : IEnumerable<T>
{
    class Node
    {
        internal T m_val;
        internal volatile Node m_next;
    }

    private volatile Node m_head;
    private volatile Node m_tail;

    public ConcurrentQueue()
```

```
{  
    m_head = m_tail = new Node();  
}  
  
public int Count  
{  
    get  
    {  
        int count = 0;  
        for (Node curr = m_head.m_next;  
             curr != null; curr = curr.m_next) count++;  
        return count;  
    }  
}  
  
public bool IsEmpty  
{  
    get { return m_head.m_next == null; }  
}  
  
private Node GetTailAndCatchUp()  
{  
    Node tail = m_tail;  
    Node next = tail.m_next;  
  
    // Update the tail until it really points to the end.  
    while (next != null)  
    {  
        Interlocked.CompareExchange(ref m_tail, next, tail);  
        tail = m_tail;  
        next = tail.m_next;  
    }  
  
    return tail;  
}  
  
public void Enqueue(T obj)  
{  
    // Create a new node.  
    Node newNode = new Node();  
    newNode.m_val = obj;  
  
    // Add to the tail end.  
    Node tail;  
    do  
    {  
        tail = GetTailAndCatchUp();  
        newNode.m_next = tail.m_next;  
    }  
    while (Interlocked.CompareExchange(  
        ref tail.m_next, newNode, null) != null);  
}
```

```
// Try to swing the tail.  If it fails, we'll do it later.
Interlocked.CompareExchange(ref m_tail, newNode, tail);
}

public bool TryDequeue(out T val)
{
    while (true)
    {
        Node head = m_head;
        Node next = head.m_next;

        if (next == null)
        {
            val = default(T);
            return false;
        }
        else
        {
            if (Interlocked.CompareExchange(
                ref m_head, next, head) == head)
            {
                // Note: this read would be unsafe with a C++
                // implementation. Another thread may have dequeued
                // and freed 'next' by the time we get here, at
                // which point we would try to dereference a bad
                // pointer. Because we're in a GC-based system,
                // we're OK doing this -- GC keeps it alive.
                val = next.m_val;
                return true;
            }
        }
    }
}

public bool TryPeek(out T val)
{
    Node curr = m_head.m_next;

    if (curr == null)
    {
        val = default(T);
        return false;
    }
    else
    {
        val = curr.m_val;
        return true;
    }
}
```

```
public IEnumerator<T> GetEnumerator()
{
    Node curr = m_head.m_next;
    Node tail = GetTailAndCatchUp();

    while (curr != null)
    {
        yield return curr.m_val;

        if (curr == tail)
            break;

        curr = curr.m_next;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return ((IEnumerable<T>)this).GetEnumerator();
}
```

One obvious difference when compared to the stack is that `m_head` can never be `null`. We initialize the queue with a sentinel dummy node, and both `m_head` and `m_tail` initially refer to it. When `m_head` is equal to `m_tail`, which means that `m_head.m_next` is `null`, the queue is considered empty. The reason we do this is the same as why we did for the fine-grained locking case: we need to avoid cases that would call for updating both `m_head` and `m_tail` atomically (i.e., when the first element was added or last element removed).

The algorithm uses a subtle trick. When enqueueing a new node, we must update the tail node's next reference to the new node. In order to quickly find the new tail node for enqueues, we will use the `m_tail` field. Once the tail has been found, we then attempt to CAS the new node as its `m_next` field, using `null` as the comparison value. After this CAS succeeds, however, `m_tail` is actually out of sync and subsequent enqueues may notice it as such. To resolve the issue, a thread enqueueing a new node must CAS `m_tail` to point at the newly enqueued node as quickly as possible. The trick is that this second CAS may fail, although the first one succeeded. The algorithm works by having all threads “catch up” the tail in the event that they see that it is out of date, otherwise they would have

to wait indefinitely for the enqueueing thread to complete; this would effectively form a lock during enqueue. It is easy to detect when a tail is inaccurate: `m_tail` will have a non-null next field. The `GetTailAndCatchUp` method encapsulates this logic. Before enqueueing anything new, a thread ensures the tail is caught up. The tail can only be a single node behind the real tail because in order to enqueue another, it must be up to date. But one thread can get stuck continuously updating the tail for many other successfully enqueueing threads.

Most of the remainder of the algorithm is straightforward and should be familiar due to the similarities to `LockFreeStack<T>`. The `GetEnumerator` method is worth examining in more detail because it is a design point that is apt to come up in practice when developing new containers. The implementation effectively provides a “snapshot” of the state of the queue at a particular time. A thread enumerating the contents will not observe subsequent updates. But there is actually no copying involved. It does this by remembering the tail at the time `GetEnumerator` was called; it then subsequently walks the linked list during enumeration and stops when it reaches the tail. Because we never modify the `m_next` fields of nodes in the queue after they have been enqueued, we can safely rely on them remaining valid.

Work Stealing Queue

Most schedulers—such as the CLR thread pool—operate by having a single global work queue. This queue is protected by a lock, and all enqueues and dequeues must serialize with respect to one another. Each worker thread in the pool goes back to this central queue and grabs a new work item when it finishes running its current task. While simple, this can lead to a large amount of contention on the central queue. For fine-grained tasks with short execution times, and as processor counts grow, the threads will spend an increasing amount of time in contention.

An alternative data structure called a **work stealing queue** can be used to substantially reduce this contention and improve scalability. This queue makes it incredibly cheap to push and pop from the so-called thread private end, but allows for “steals” (pops) by foreign threads to occur from the

opposite end (although foreign pushes are not allowed). The way this can be applied to a thread pool is to keep a global queue for work that comes from threads outside of the pool's purview, but to queue all recursively queued work into a per thread work stealing queue. When the thread is looking for work, it first consults its local queue. For divide and conquer algorithms or others where tasks are generated from within other tasks, this can lead to sizeable improvements. Moreover, it encourages finer-grained decomposition due to reduced costs.

Before diving into the implementation (in C#) of our `WorkStealingQueue<T>`, a brief introduction is in order. The queue is array based and is a basic circular queue with a head and tail index. The `LocalPush` and `LocalPop` methods are meant for the single thread that owns the queue, and so long as the queue is small, they can add and remove without locks. The `TrySteal` method is meant for a foreign thread to pop from the opposite end and is thread safe so that multiple foreign threads can try to perform this operation simultaneously. When the queue is small, the local methods must acquire locks to be safe with respect to concurrent steals.

Here's the code.

```
public class WorkStealingQueue<T>
{
    private const int INITIAL_SIZE = 32;
    private T[] m_array = new T[INITIAL_SIZE];
    private int m_mask = INITIAL_SIZE - 1;
    private volatile int m_headIndex = 0;
    private volatile int m_tailIndex = 0;
    private object m_foreignLock = new object();

    public bool IsEmpty
    {
        get { return m_headIndex >= m_tailIndex; }
    }

    public int Count
    {
        get { return m_tailIndex - m_headIndex; }
    }

    public void LocalPush(T obj)
    {
        int tail = m_tailIndex;
```

```
// When there is space, we can take the fast path.
if (tail < (m_headIndex + m_mask))
{
    m_array[tail & m_mask] = obj;
    m_tailIndex = tail + 1;
}
else
{
    // We need to contend with foreign pops, so we lock.
    lock (m_foreignLock)
    {
        int head = m_headIndex;

        // If there is still space (one left), add the element.
        if (tail < (head + m_mask))
        {
            m_array[tail & m_mask] = obj;
            m_tailIndex = tail + 1;
        }
        else
        {
            // Otherwise, we're full; expand the queue by
            // doubling its size (ignoring overflow).
            T[] newArray = new T[m_array.Length << 1];
            for (int i = 0; i < m_array.Length; i++)
                newArray[i] = m_array[(i + head) & m_mask];

            // Reset the field values, incl. the mask.
            m_array = newArray;
            m_headIndex = 0;
            m_tailIndex = tail - m_mask;
            m_mask = (m_mask << 1) | 1;

            // Now place the new value.
            m_array[tail & m_mask] = obj;
            m_tailIndex = tail + 1;
        }
    }
}

public bool LocalPop(out T obj)
{
    // Decrement the tail using a fence to ensure the subsequent
    // read doesn't come before.
    int tail = m_tailIndex - 1;
    Interlocked.Exchange(ref m_tailIndex, tail);

    // If there is no interaction with a take, do the fast path.
```

```
if (_headIndex <= tail)
{
    obj = _array[tail & _mask];
    return true;
}
else
{
    // Interaction with takes: 0 or 1 elements left.
    lock (_foreignLock)
    {
        if (_headIndex <= tail)
        {
            // Element still available. Take it.
            obj = _array[tail & _mask];
            return true;
        }
        else
        {
            // We lost the race, element was stolen, restore.
            _tailIndex = tail + 1;
            obj = default(T);
            return false;
        }
    }
}
}

public bool TrySteal(out T obj)
{
    return TrySteal(out obj, 0); // no blocking by default.
}

private bool TrySteal(out T obj, int millisecondsTimeout)
{
    if (Monitor.TryEnter(_foreignLock, millisecondsTimeout))
    {
        try
        {
            // Increment head, and ensure read of tail doesn't
            // move before it (fence).
            int head = _headIndex;
            Interlocked.Exchange(ref _headIndex, head + 1);

            if (head < _tailIndex)
            {
                obj = _array[head & _mask];
                return true;
            }
        }
    }
}
```

```

        {
            // Failed, restore head.
            m_headIndex = head;
        }
    }
finally
{
    Monitor.Exit(m_foreignLock);
}
}

obj = default(T);
return false;
}
}

```

Let's look briefly at some highlights. `LocalPush` has two paths: the fast path, which it can take if it will not contend with concurrent foreign pops, and the slow path, which runs under a lock. The fast path increments the tail and stores the element into the array without any added synchronization overhead. This is ultra cheap. Note that instead of ensuring the `m_tailIndex` and `m_headIndex` values stay within bounds, we keep `m_mask` up to date and use it whenever an index is used to access array elements. The slow path does the same thing, except it also checks for resizing the array. If resizing is necessary, it doubles the size (without checking for overflow) and copies the elements.

The `LocalPop` method is similar: it operates on the tail end, just like `LocalPush`, and can also take a fast path if there is sufficient room in the queue. Unfortunately this is a little more expensive than `LocalPush` because we need a fence to prevent the initial write of `m_tailIndex` from passing the subsequent read of `m_headIndex`. Recall from Chapter 10 that this is a legal movement in the .NET memory model.

The `TrySteal` method operates similar to `LocalPop`, except that it executes under the protection of a lock. And it takes elements from the opposite end, using `m_headIndex` instead of `m_tailIndex`. This is the only method that is safe to call from foreign threads.

Coordination Containers

Let's take a look at a few coordination oriented containers.

Producer/Consumer Data Structures

A common relationship formed among two or more tasks is referred to as a **producer/consumer relationship**. In this situation, one or more producers are linked to one or more consumers through some communication mechanism. Producers are responsible for generating items of interest, and consumers process the items in some interesting way. The items generated can be anything: blocks of data read off the disk, received via the network, an infinite stream of information, simulation data, and so on. Concurrency is inherent in this situation because producing and consuming are typically completely independent activities.

The ratio of producers to consumers can vary dramatically. The ratio that leads to optimal throughput depends on the costs involved to produce and consume elements: if consuming an item is 10 times the cost of producing that item, it's likely a producer to consumer ratio of 1:10 would be best to balance out the producers and consumers. We can extend this situation to have multiple stages, which forms a **pipeline**. A pipeline is the composition of many producer/consumer relationships into a larger dataflow; we will look more closely at them in the next Chapter 13, Data and Task Parallelism.

A common way to implement the communication for producer/consumer situations is with a container type. None of the above containers had any kind of coordination built in except for simple mutual exclusion, so they are inadequate. When the `LockFreeQueue<T>` becomes empty, for instance, `TryDequeue` simply returns `false`. What the caller does in response is not a concern for the container itself. But what if a caller just wanted to wait for an element to arrive? It's fairly simple to build a so-called **blocking queue** that provides this behavior intrinsically by wrapping an existing queue with some additional synchronization. As another related example, what if we expect producers to sometimes get ahead of the consumers? We may want to throttle the rate at which new elements are enqueued to limit memory consumption. To do this, we may also have some logic to block producers, something called a **bounded buffer**.

We will now take a look at several alternative approaches to building both kinds of containers. It's often useful to have a single type that has

both blocking and bounding, but we will start simple. The three basic implementation considerations we must make are:

- The containers must be safe to access concurrently. We will demonstrate fairly simple approaches with coarse grain, but when scalability is important, any of the techniques shown earlier can be used.
- When a consumer attempts to take an element from an empty queue, it must be blocked until the next producer makes an element available, a.k.a. **blocking**.
- When a producer attempts to place an element into a full queue, it must be blocked until the next consumer takes an element and makes space, a.k.a. **bounding**.

Also note that we will use existing containers (such as .NET's `Queue<T>` and C++ STL's `queue<T>`) rather than rolling our own. This is done for brevity, but you may instead choose to look at custom data structures that might enable fine-grained locking. The choice of a queue is purely an implementation detail, but ensures elements are given to consumers in roughly the same order they are produced (with all of the standard timing related concurrency caveats).

A Simple C# Blocking Queue with Monitors

For the simplest example, we will use .NET's `Monitor` class for the C# example and then the nearly equivalent code in VC++ with Win32 critical sections and condition variables. The condition variable capabilities of these give us an easy way to both ensure thread safety and to also wait and signal threads when some event of interest occurs.

There are certainly alternative approaches. For instance, we could use a semaphore to track the count of elements remaining in the queue. In fact, you saw an example implementation of such a data structure back in Chapter 5, Windows Kernel Synchronization. It was a way to illustrate the use of mutexes and semaphores, and a more efficient implementation was promised. You likely wouldn't want to use that approach in practice because it involves kernel transitions on each enqueue and dequeue operation. Another alternative is to use a kernel event instead—such as a manual-reset event that gets set when transitioning from empty to non-empty and reset when moving from nonempty to empty—but this can be more complicated and has no immediately obvious benefit.

Here's an initial cut at a very simple BlockingQueue<T> in C#.

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

public class BlockingQueue<T>
{
    private Queue<T> m_queue = new Queue<T>();
    private int m_waitingConsumers = 0;

    public int Count
    {
        get
        {
            lock (m_queue)
                return m_queue.Count;
        }
    }

    public void Clear()
    {
        lock (m_queue)
            m_queue.Clear();
    }

    public bool Contains(T item)
    {
        lock (m_queue)
            return m_queue.Contains(item);
    }

    public void Enqueue(T item)
    {
        lock (m_queue)
        {
            m_queue.Enqueue(item);

            // Wake consumers waiting for a new element.
            if (m_waitingConsumers > 0)
                Monitor.Pulse(m_queue);
        }
    }

    public T Dequeue()
    {
        lock (m_queue)
        {
            while (m_queue.Count == 0)
            {
                // Queue is empty, wait until an element arrives.

```

```

        m_waitingConsumers++;
        try
        {
            Monitor.Wait(m_queue);
        }
        finally
        {
            m_waitingConsumers--;
        }
    }

    return m_queue.Dequeue();
}
}

public T Peek()
{
    lock (m_queue)
        return m_queue.Peek();
}
}
}

```

The container has two fields: a queue to hold elements and a count of consumers that are blocked waiting for elements to arrive. (Note that this particular example would also work without the `m_waitingConsumers` field. It turns out that this has some slight performance advantages because we avoid superfluous calls to `Monitor.Pulse` when no threads are waiting.) Many methods add some locking but are otherwise just simple wrappers on top of the queue: `Count`, `Clear`, `Contains`, and `Peek`, for example. `Enqueue` and `Dequeue` are the interesting bits. A consumer in `Dequeue` checks the count of the queue and, if it empty, must wait. First it increments `m_waitingConsumers` and then calls `Monitor.Wait`. When a producer enqueues a new element, it checks `m_waitingConsumers` and will call `Monitor.Pulse` to wake a single waiting thread if it is non-0. A consumer that wakes up in this manner decrements the `m_waitingConsumers` field and proceeds to remove and return the element from the underlying queue.

A Simple C++ Blocking Queue with Critical Sections and Condition Variables

Here is an example much like the one shown in C#, but instead using the new Windows Vista condition variable support for waiting and signaling. Very little must change.

```
template <class T>
class BlockingQueue
{
private:
    queue<T> * m_pQueue;
    CRITICAL_SECTION m_exclusiveLock;
    CONDITION_VARIABLE m_consumerEvent;

public:
    BlockingQueue()
    {
        m_pQueue = new queue<T>();
        InitializeCriticalSection(&m_exclusiveLock);
        InitializeConditionVariable(&m_consumerEvent);
    }

    ~BlockingQueue()
    {
        DeleteConditionVariable(&m_consumerEvent);
        DeleteCriticalSection(&m_exclusiveLock);
        delete m_pQueue;
        m_pQueue = NULL;
    }

    void Enqueue(T item)
    {
        EnterCriticalSection(&m_exclusiveLock);
        m_pQueue->push(item);
        LeaveCriticalSection(&m_exclusiveLock);

        // Wake consumers who are waiting for a new item.
        WakeConditionVariable(&m_consumerEvent);
    }

    T Dequeue()
    {
        T item;

        EnterCriticalSection(&m_exclusiveLock);
        // If the queue is empty, wait until a new item arrives.
        while (m_pQueue->empty())
            SleepConditionVariableCS(
                &m_consumerEvent, &m_exclusiveLock, INFINITE);

        item = m_pQueue->pop();
        LeaveCriticalSection(&m_exclusiveLock);

        return item;
    }
};
```

The structure of this code is nearly identical to the managed implementation: there's a little more state management minutia and the optimization to avoid unnecessary pulses has been omitted for brevity. Prior to Windows Vista, this would have been far more difficult to implement, requiring you to use heavyweight semaphores, mutexes, and/or events instead.

C# Blocking/Bounded Queue with Multiple Monitors

An unbounded queue has one major disadvantage in producer/consumer scenarios: producers and consumers may become imbalanced over time.

Say that you predicted your average producer's throughput would be 500 items/second and that your average consumer's throughput would be 1,000 items/second. Based on this, you might reasonably decide to (statically) assign two producers for every consumer in order to offset the imbalance. But what happens if the dynamic execution of your program results in actual throughputs of 750 items/second for both? Instead of the predicted cost ratio of 1:2, the ratio is 1:1. Producers are creating items at a rate twice what the consumers can keep up with, resulting in 750 items/second surplus production for each producer. Some simple math: if we have 16 producers, after 10 seconds the buffer will have grown to hold 120,000 items; after 60 seconds, 720,000 items; and so on. Unless we do something about it, this could be disastrous, especially in long running programs such as server applications. If each item is 1KB bytes in size, that's approaching 1GB of memory just to hold them all after 60 seconds, and an out of memory condition shortly after that.

A bounded buffer throttles producers so that this problem is avoided. This is very similar to the blocking queue described above, only the reverse: instead of a consumer blocking when the queue has become empty, the producer blocks when the queue has become full. It is then the responsibility of consumers to notify waiting producers that a slot has become available in the queue, much like producers in the blocking queue do when a new item is added. We can simply extend our previous `BlockingQueue<T>` implementation to accommodate this coordination. It's certainly reasonable to have a bounded buffer in which consumers do not block on empty, but it's also common to want both simultaneously.

To get started, we add a `m_capacity` field to hold the upper bound of the queue's size, and will use two objects (instead of one) as condition

variables for producers and consumers that observe full and empty queues, respectively: `m_fullEvent` and `m_emptyEvent`. We still use the queue itself as a way to synchronize access to the data:

```
public class BlockingBoundedQueue<T>
{
    private Queue<T> m_queue = new Queue<T>();
    private int m_capacity;
    private object m_fullEvent = new object();
    private int m_fullWaiters = 0;
    private object m_emptyEvent = new object();
    private int m_emptyWaiters = 0;

    public BlockingBoundedQueue(int capacity)
    {
        m_capacity = capacity;
    }

    public int Count
    {
        get
        {
            lock (m_queue)
                return m_queue.Count;
        }
    }

    public void Clear()
    {
        lock (m_queue)
            m_queue.Clear();
    }

    public bool Contains(T item)
    {
        lock (m_queue)
            return m_queue.Contains(item);
    }

    public void Enqueue(T item)
    {
        lock (m_queue)
        {
            // If full, wait until an item is consumed.
            while (m_queue.Count == m_capacity)
            {
                m_fullWaiters++;
                try
                    {
                        m_fullEvent.WaitOne();
                    }
                catch (InterruptedException)
                    {
                        m_fullWaiters--;
                    }
            }
            m_queue.Enqueue(item);
            m_emptyEvent.Set();
        }
    }

    public T Dequeue()
    {
        lock (m_queue)
        {
            if (m_queue.Count > 0)
            {
                T item = m_queue.Dequeue();
                m_emptyEvent.WaitOne();
                return item;
            }
            else
                throw new InvalidOperationException("Queue is empty.");
        }
    }
}
```

```
        {
            lock (m_fullEvent)
            {
                Monitor.Exit(m_queue);
                Monitor.Wait(m_fullEvent);
                Monitor.Enter(m_queue);
            }
        }
        finally
        {
            m_fullWaiters--;
        }
    }

    m_queue.Enqueue(item);
}

// Wake consumers who are waiting for a new item.
if (m_emptyWaiters > 0)
    lock (m_emptyEvent)
        Monitor.Pulse(m_emptyEvent);
}

public T Dequeue()
{
    T item;

    lock (m_queue)
    {
        while (m_queue.Count == 0)
        {
            // Queue is empty, wait for a new item to arrive.
            m_emptyWaiters++;
            try
            {
                lock (m_emptyEvent)
                {
                    Monitor.Exit(m_queue);
                    Monitor.Wait(m_emptyEvent);
                    Monitor.Enter(m_queue);
                }
            }
            finally
            {
                m_emptyWaiters--;
            }
        }
        item = m_queue.Dequeue();
    }
}
```

```
// Wake producers who are waiting to produce.  
if (_fullWaiters > 0)  
    lock (_fullEvent)  
        Monitor.Pulse(_fullEvent);  
  
    return item;  
}  
  
public T Peek()  
{  
    lock (_queue)  
        return _queue.Peek();  
}  
}
```

This code is a little more complicated than the `BlockingQueue<T>` example we saw previously, but not by much. The most complicated aspect is caused by our use of separate condition variables to represent the producer and consumer wait conditions. We could have legitimately used the `_queue` object for both events so long as we started using `PulseAll` instead of `Pulse` for notifications, ensuring any producer or consumer waiting would be awakened. But this would cause threads to wake up superfluously (in stampede fashion) only to find out they must go back to sleep. We also use a similar optimization to `BlockingQueue<T>` to avoid calling `Pulse` when no thread of the particular kind is waiting on the condition variable.

Before calling `Wait` on either event, we have to manually exit the mutual exclusive lock on `_queue` taken by the `lock (_queue) { ... }` statement (but only after entering the appropriate lock). Invoking `Wait(x)` on some object `x` releases the lock on `x` and then waits, in that order. Because we use a separate object for locking and event orchestration, we have to do this manually, otherwise another thread couldn't acquire the lock and make the condition we're waiting for become true. The result would be deadlock. This is safe in this specific code because of the waiting flags; we increment them inside of the `_queue` lock, guaranteeing subsequent threads will notice a value greater than 0 and contend for the lock used for signaling. This is subtle and certainly isn't always the case, so be careful if you ever do this.

Another subtlety is that we call `Pulse` on the events *after* we've released the lock on `_queue`. This is a slight performance optimization: we could have just as correctly signaled while the lock was held. But the first thing all waiting threads do when they wake up—producers and consumers alike—is try

to reacquire the lock on `m_queue`, so if we still held it when we signaled the event, we could create two-step dance scalability problems such as those we saw in Chapter 11, Concurrency Hazards.

Phased Computations with Barriers

Another kind of orchestration that is somewhat common but that isn't strictly a container, is called a **barrier**. Computations that use barriers are typically called **phased computations**. The kinds of algorithms that use barriers are split into separate phases and are sometimes cyclic such that all threads in a group wait for each participant to reach the end of the current phase before moving on to the next. The CLR's GC, for example, uses this approach to synchronize threads in the server GC when moving between its various phases: marking, relocating, and compacting. It is common to have some data being produced by threads participating in a given phase, stored in some shared location (such as having thread `n` store data into an array `a` at slot `a[n]`), which can be safely accessed by all participants during the next phase.

The basic data structure's task is simple: it must block all threads that arrive at the barrier until a certain number have arrived; at that point, all threads are released atomically. There are several alternative algorithms to choose from. One that performs well on reasonable numbers of processors (i.e., machines you're apt to program today) and that doesn't require any kind of locking, is called a **sense-reversing barrier** (see Further Reading, Mellor-Crummey, Scott). The barrier tracks whether the current phase is odd or even and uses a separate event internally based on this. The separate senses are needed to avoid races that would result (e.g., setting and then resetting the event). This trick also makes it simple to transition the barrier's current count using only interlocked operations.

```
#pragma warning disable 0420

using System;
using System.Threading;

public class Barrier : IDisposable
{
    private readonly int m_initialCount; // Initial count.
```

```
// High order bit 0==even, 1==odd; other bits are count.
private volatile int m_currentCountAndSense;
private const int MASK_CURR_SENSE = unchecked((int)0x80000000);
private const int MASK_CURR_COUNT = ~MASK_CURR_SENSE;

private ManualResetEvent m_oddEvent; // Event for odd phases.
private ManualResetEvent m_evenEvent; // Event for even phases.

public Barrier(int initialCount)
{
    if (initialCount < 1)
        throw new ArgumentOutOfRangeException("initialCount");

    m_initialCount = initialCount;
    m_currentCountAndSense = initialCount; // Start at even sense.
    m_oddEvent = new ManualResetEvent(false);
    m_evenEvent = new ManualResetEvent(false);
}

public int InitialCount
{
    get { return m_initialCount; }
}

public int CurrentCount
{
    get { return m_currentCountAndSense&MASK_CURR_COUNT; }
}

internal void SignalAndWait()
{
    TrySignalAndWait(Timeout.Infinite);
}

internal bool TrySignalAndWait(int timeoutMilliseconds)
{
    // Read the sense so we can reverse it later if needed.
    int sense = (m_currentCountAndSense & MASK_CURR_SENSE);

    // We may have to retry in the case of timeouts, hence the loop.
    while (true)
    {
        int currentCountAndSense = m_currentCountAndSense;
        if ((currentCountAndSense & MASK_CURR_COUNT) == 1)
        {
            // Last thread, try to reset the barrier state.
            if (Interlocked.CompareExchange(
                ref m_currentCountAndSense,
                m_initialCount|(~(m_currentCountAndSense)&MASK_CURR_SENSE),
                currentCountAndSense) == 1)
                break;
        }
    }
}
```

```
        currentCountAndSense) != currentCountAndSense)
    continue; // CAS failed, retry.

    // Reset old event 1st, ensuring threads that wake up
    // don't race and satisfy the next phase.
    if (sense == 0)
    {
        // Even.
        m_oddEvent.Reset();
        m_evenEvent.Set();
    }
    else
    {
        // Odd.
        m_evenEvent.Reset();
        m_oddEvent.Set();
    }
}
else
{
    // Not last thread, decrement the count and wait.
    int newCount = (currentCountAndSense & MASK_CURR_SENSE) |
        ((currentCountAndSense & MASK_CURR_COUNT) - 1);
    if (Interlocked.CompareExchange(
        ref m_currentCountAndSense, newCount,
        currentCountAndSense) != currentCountAndSense)
        continue; // CAS failed, retry.

    // Wait on the event.
    bool waitSuccess;
    if (sense == 0)
        waitSuccess = m_evenEvent.WaitOne(
            timeoutMilliseconds, false);
    else
        waitSuccess = m_oddEvent.WaitOne(
            timeoutMilliseconds, false);

    // Timeouts are tricky since we already told other
    // threads we reached the barrier. Need to consider
    // that they may have already noticed our state updates
    // and hence moved to the next phase. If they did move
    // to the next phase, we will have to return true rather
    // than timing out. We know this by checking the sense.
    while (!waitSuccess)
    {
        currentCountAndSense = m_currentCountAndSense;
        if ((currentCountAndSense & MASK_CURR_SENSE) !=
            sense)
            // Sense changed. We are past the point of
```

```
// timing out: return true.  
break;  
  
int resetCount =  
    (currentCountAndSense & MASK_CURR_SENSE) |  
    ((currentCountAndSense & MASK_CURR_COUNT) + 1);  
if (Interlocked.CompareExchange(  
    ref m_currentCountAndSense, resetCount,  
    currentCountAndSense) != currentCountAndSense)  
    continue; // CAS failed, retry.  
  
// Timed out and patched up our state changes.  
return false;  
}  
}  
  
return true;  
}  
}  
  
public void Dispose()  
{  
    m_oddEvent.Close();  
    m_evenEvent.Close();  
}  
}
```

This implementation is fairly dense. First notice that we bit pack the current count and the phase (even or odd) into a single field: a high bit of 0 means we're in an even phase, while a high bit of 1 means we're in an odd phase. This complicates life slightly when we're updating or reading the `m_currentCountAndSense` field, but provides some performance gain and enables a lock free implementation because we can update both with a single compare-and-swap.

Let's walk through the primary steps in the `TrySignalAndWait` method.

- We read the current sense (with appropriate masks) and check whether there is a count of 1 remaining. If yes, the calling thread is the last one and must transition the barrier to the next phase, including signaling other threads waiting at the barrier. If no, we can update the count and wait.
- If the caller is the final thread in the phase, the `m_currentCountAndSense` field is updated: the phase is reversed (if it was odd, it becomes

even, and vice versa), and the count is reset back to `m_initialCount`. Once we set the event, threads will awaken to find the barrier in the valid state for the next phase.

- If the phase was even (bit was 0), we reset `m_oddEvent` and then signal `m_evenEvent`. If the phase was odd, we reset `m_evenEvent` and set `m_oddEvent`. Notice that it's crucial we do the reset first. If we woke threads and then reset the event, threads would move on to the next phase and any waiting would be satisfied immediately. This kind of **overtaking race** would completely break the validity of our implementation.
- Waiting threads initially have an easier time. They decrement the current count keeping the sense identical by using a `CompareExchange`. They then wait on the appropriate event based on the sense, supplying a timeout (if any). If the wait succeeds (no timeout), the method can return right away.
- Here is where things get tricky. If a thread awakens due to a timeout, we need to undo the update to the current count, because the last thread may arrive in the meantime and transition to the next phase, thinking that the timed out thread successfully woke up. We want to catch this. So we attempt to revert the initial change by incrementing the count and keeping the phase identical. But if, in this process, the barrier notices that the sense has changed in the meantime, we will instead act as though the wait didn't timeout and return successfully.
- There's also a lot of looping to handle failed interlocked operations. In fact, for every interlocked operation we must handle the possibility of failure.

Lastly, `Barrier` also implements `IDisposable` because it owns two kernel events.

Where Are We?

In this chapter, we surveyed several different approaches to building scalable parallel containers. This included solutions ranging from coarse-grained to fine-grained locking and even those that didn't require locking at all.

(i.e., lock free). We concluded with a look at some common coordination oriented data structures. This chapter applied many of the concepts seen in all the previous chapters. In the next chapter, we will begin looking at some of the data and task parallel patterns and algorithms that are common and that might benefit from using the containers we just explored.

FURTHER READING

- C. Click. A Lock-Free Hashtable. *JavaOne* (2007).
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. (The MIT Press, 2001).
- S. Heller, M. Herlihy, V. Luchangco, M. Moir, B. Scherer, N. Shavit. A Lazy Concurrent List-based Set Algorithm. In *Principles of Distributed Systems* (2005).
- M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming*. (Morgan Kaufmann, 2008).
- J. Mellor-Crummey, M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM TOCS* (1991).
- M. M. Michael, M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *15th Annual ACM Symposium on Principles of Distributed Computing* (1996).
- M. M. Michael, M. L. Scott. Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. In *Journal of Parallel and Distributed Computing*, 51(1) (1998).
- M. M. Michael. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *14th Annual ACM Symposium on Parallel Algorithms and Architectures* (2002).
- M. M. Michael. CAS-based Lock-Free Algorithm for Shared Deques. In *9th Euro-Par Conference on Parallel Processing*, LNCS, Vol. 2790 (2003).
- C. Purcell, T. Harris. Non-blocking Hashtables with Open Addressing, Technical Report UCAM-CL-TR-639. (University of Cambridge, 2005).

13

Data and Task Parallelism

MOST OF THIS BOOK has been dedicated to specific mechanisms and best practices used when building concurrent programs. Algorithms that use these mechanisms are important to understand too but, until this point, we've only touched on this topic in passing. That's what this chapter is about. We'll look at many algorithms that are common to concurrent programs and will see various ways that sequential algorithms can be decomposed into subproblems suitable for parallel execution.

Whenever writing an algorithm to use concurrency, the first and most important design choice that needs to be made is how to partition the original problem into individual sub-parts. There are three broad approaches that we will look at in this chapter: data, task, and message based parallelism. These classifications can help to frame your thoughts.

- **Data parallelism** uses the input data to some operation as the means to partition into smaller pieces, either because there is a large amount of data to process, the processing operation is costly, or a combination of both. Data is divvied up among the available hardware processors in order to achieve parallelism. This partitioning step is often followed by replicating and executing some mostly independent program operation across these partitions. Typically it's the same operation that is applied concurrently to the elements in

the dataset. Optionally, a final aggregation step is used to combine the multiple independent results into a single result. All of this synchronization and coordination is packaged into simple constructs, such as parallel for loops and declarative statements. This often takes the form of the now popular map/reduce paradigm (see Further Reading, Dean, Ghemawat).

- **Task parallelism** takes a different approach. Programs are already decomposed into individual parts—statements, methods, and so forth—that can often be run in parallel, particularly in object oriented systems. Task parallelism takes and extends the preexisting functional partitioning that already exists, and runs independent pieces in parallel with respect to one another. Two major approaches are commonplace: **structured** and **unstructured task parallelism**. Structured parallelism encapsulates all synchronization in simple to use abstractions with clear begin and end points, much like data parallelism. Unstructured parallelism, on the other hand, often demands explicit synchronization, making it more difficult to use without encountering the kinds of concurrency hazards we looked at in Chapter 11, Concurrency Hazards. Structured parallelism should be preferred when possible.
- **Message based parallelism** is yet a different approach. Partitioning is often achieved via events and workflow and is a byproduct of orchestrated dependencies rather than performance. Problems are decomposed into independent units of work whose execution is self-contained and keyed off of the completion of some previous event(s) of interest. As with data parallelism and structured task parallelism, synchronization and coordination are usually hidden behind some set of abstractions for representing events and dependencies.

While the three groupings are not strictly orthogonal, and there are alternative ways of grouping and categorizing parallel programming models, this taxonomy tends to be a useful and is driven mostly by the coordination and data access patterns employed by parallel workers. Deciding which technique to employ depends a lot on the design forces present in the overall program. For example, when using concurrency for performance, the major design considerations are typically partitioning the input

problem so as to optimize memory access patterns, that is, to improve cache locality, in addition to trying to reduce the amount of communication and synchronization, and achieve good load balance between the processors. Conversely, when using concurrency for responsiveness or to hide latencies, these factors matter less, and ease of programming, robustness, and maintainability tend to be more important.

Data Parallelism

As summarized already, task decomposition is a common way to achieve parallelism. Breaking larger problems apart into smaller subproblems is something developers are used to doing on a regular basis when writing sequential software, so it's often a natural first approach to consider when adding parallelism to a program. It's also more cognitively familiar. In sequential software, the decomposition into methods is done to support APIs and architecture, to improve the code's maintainability, and/or to ease the mental burden on the developers of the program. The exercise has little to do with performance, and in fact overdecomposing a problem into too many individual pieces leads to worse performance due to the overhead of indirections.

While task parallelism works for many classes of problem, it is not always appropriate. Many new concerns must be considered: performance, load balance between different subproblems, data sharing, control and data dependencies among the subproblems, and so on. Breaking apart a function into smaller bits of work for parallelism is a very different beast. Moreover, the number of individual methods in a program is rarely dynamic, and so an approach that uses task parallelism is typically inherently limited in terms of scalability.

Data parallelism takes a different approach that side steps many of these issues. (That's why we're covering it first.) Most programs spend a large amount of their execution time running loops: for example, for loops over an iteration range, C# foreach loops (or VB ForEach loops, or loops which use C++ STL iterators) over the contents of a collection of data, or while loops to execute so long as some predicate evaluates to true. If we were looking for opportunities to find the "biggest bang for the buck" when it comes to parallelism, it would seem that somehow parallelizing these loops

might be fruitful. In doing so, it often becomes evident that many loops in programs are comprised of iterations that are entirely independent of one another, that is, the execution of iteration i does not depend on the outcome of some separate predecessor or successor iteration j , or at least could be written that way.

This is great for parallelism, because, in the extreme, it means all loop iterations could run in parallel at once. Given enough processors, of course.

The data parallelism approach is also nice for scalability. The upper limit on parallelism is typically much larger, because loop iteration counts are often quite large and dependent on the dynamic size of data that must be operated upon. The amount of data on which programs must operate normally grows over time, and while processor clock speeds have begun to slow, the growth in disk space usage has not. GBs are now giving way to TBs, and there is no end in sight (aside from physical limitations on how fast humans can create the data). Growth in data sizes in a data parallel program translates into the exposure of more parallelism opportunities that can scale to use many processors as they become available. Because of this, many industry experts believe that data parallelism is the most scalable and future-proof way of building parallel programs—programs that will not be inherently limited by their construction.

Data parallelism is not a panacea. Every part of every program is not comprised of a loop. Some things can be expressed that way, but not all. This is why the recommended architecture for concurrent applications, outlined back Chapter 1, Introduction, encourages higher level isolation and architectural separation of independent parts, mixing diverse kinds of parallelism together in the same program. But for parts of the program that can use it, data parallelism should be the first choice.

Loops and Iteration

Let's begin with simple loop parallelism. When data parallelism is used, the first thing to consider is how to break the iteration space into independent units of work. In the case of an ordinary `for` loop, the iteration space is typically a range of integers, while `foreach` loops iterate over individual elements in some collection. What is the best way to divvy these things up among the processors?

For example, if we were to parallelize the following loop, how would we decide how many threads to use, how best to schedule them, how to assign iteration ranges to threads, and so on?

```
void For(int lo, int hi, Action<int> body)
{
    for (int i = lo; i < hi; i++)
    {
        body(i);
    }
}
```

The same questions are equally interesting for parallelizing code that iterates over collections of data, for example, an array or any other data structure with an indexer (such as `IList<T>` in the .NET Framework and `std::vector` in C++'s STL).

```
void For<T>(T[] arr, Action<T> body)
{
    for (int i = 0; i < arr.Length; i++)
    {
        body(arr[i]);
    }
}
```

Notice that the second loop can be trivially written in terms of the first one.

```
void For<T>(T[] arr, Action<T> body)
{
    For(0, arr.Length, i => body(arr[i]));
}
```

Because of this simple translation, we will not discuss the second style. The only advantage to writing it longhand is to avoid the double delegate invocation per iteration. But it is implied that the same parallelization techniques apply.

Different techniques are typically needed for loops that aren't based on indices (such as `while` loops) and for code that iterates over collection data structures that do not offer random access indexers. We'll encounter such a situation later when we deal with .NET `IEnumerable<T>` inputs where the size of the input isn't even known.

Prerequisites for Parallelizing Loops

Before discussing how to run these loops in parallel, it should be made clear that a necessary prerequisite to parallelizing is that the loop's body is thread safe. If it isn't, running it in parallel is sure to cause trouble. In our previous example, that means that all code run inside of the body delegate must be thread safe.

Being thread safe isn't enough for our purposes, however. Thread safety means that it's correct to run separate iterations in parallel (which is important); but thread safety might just involve body acquiring a lock for the duration of its entire function body. If we're running a loop in parallel in an attempt to attain better performance, we'd have done nothing but add a lot of concurrency related overhead to our program—with forking, joining, waiting, context switches, cache effects, and so on—and will likely see negative performance effects rather than gains, not to mention code complexity. Part of the data parallelism process, therefore, must also involve an analysis of the code that will be run inside of the loop bodies and possibly a restructuring of it so that it doesn't depend on shared state, uses more efficient fine-grained synchronization, and so forth.

Additionally, the fact that synchronization is involved may not be sufficient either. If the loop itself isn't associative—that is, order of execution doesn't matter—or it is performing nonassociative operations on data read and written by the loop bodies—then the loop may produce incorrect answers.

Static Decomposition

Once we've done the work to ensure that body is safe to run in parallel, the simplest approach to parallelizing the loop is to divide the size of the loop (i.e., $hi - lo$, assuming the iterations of the loop are in ascending order, that is, that $lo \leq hi$) by the number of processors, to get a per thread iteration count and to have each thread process a series of contiguous iterations.

This approach, called **static decomposition**, while simple, is not ideal for a few reasons, but mainly because it can lead to inefficient use of the available processors. An alternative to static decomposition is to spawn a certain number of threads, or to somehow arrange for the number of threads to scale based on available processors and to have each of those

threads calculate iterations on demand. In this approach, which we call **dynamic decomposition**, threads do not know a priori which iterations they will be executing. Instead, they find out as they execute and as they become available to run extra iterations. Both approaches will be examined.

Contiguous Iterations. To begin, let's take the loop example seen before and see what happens when we use the straightforward static decomposition already outlined above: dividing the iteration space into contiguous chunks of indices. Applying this technique to the sequential For method seen earlier, we might end up with code that looks like the following ParallelFor method

```
static void ParallelFor(int lo, int hi, Action<int> body, int p)
{
    int chunk = (hi - lo) / p; // Iterations per thread
    CountdownEvent latch = new CountdownEvent(p);

    // Schedule the threads to run in parallel
    for (int i = 0; i < p; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object obj)
        {
            int pid = (int)obj;
            int start = lo + pid * chunk;
            int end = pid == p - 1 ? hi : start + chunk;

            for (int j = start; j < end; j++)
            {
                body(j);
            }

            latch.Signal();
        }, i);
    }

    latch.Wait(); // Wait for them to finish
}
```

We let the caller choose a value for p , which represents the degree of parallelism we'll use for the loop, that is, the number of threads used to concurrently run iterations. A reasonable choice to begin with would be `Environment.ProcessorCount`, and we might want to provide an overload

that uses it by default. (In native code, you can access the number of processors with the Win32 `GetSystemInfo` API.)

Next in this function we calculate the number of elements each thread will process, `chunk`, by dividing the iteration count by the number of processors. As an illustration, say we had 100,000 iterations to perform (i.e., $(\text{hi} - \text{low}) = 100,000$) and a degree of parallelism of 16 (i.e., $p = 16$); each thread would then execute 6,250 iterations (i.e., `chunk = 6,250`). It's not a requirement that the iteration count is evenly divisible by p , so we have to take care of some edge conditions. With our partitioning strategy, the last partition could end up with more iterations to run than others.

We immediately create a `CountdownEvent` of count p : this is an event abstraction that becomes signaled once p threads have called `Signal` on it. We then queue up p work items in the CLR thread pool (each of which signals the latch upon completion) and wait on the latch. Each work item queued to the pool iterates over its iteration space: the `pid` is just the loop counter `i` passed as the second argument to `QueueUserWorkItem`. This is used for a subtle reason: if we used `i` directly from the C# anonymous delegate passed to the thread pool, it would be hoisted into a closure and shared by all iterations; the result is that the wrong value of `i` would be used by any given iteration, and, in fact, most threads would probably observe `i` as p (depending on various race conditions), which is outside of its legal range.

Each thread iterates from `lo + pid * chunk` to `lo + (pid + 1) * chunk` or `hi`, whichever is larger, and calls the `body` function, passing the iteration index as the argument. We check for `hi` because, if the task is the last of the group, it must iterate until `hi` in case the iteration count was not evenly divisible by p . Notice the indices that any given thread processes are adjacent and contiguous; this usually (but not always) helps improve cache locality, particularly when the indices will be used to index into an array. After executing the part of the loop for which the thread is responsible, it calls `latch.Signal` to indicate that it has finished. Finally, the thread that ran the parallel loop waits for all iterations to finish by calling `latch.Wait`. This call unblocks once all iterations are done.

There are a few noteworthy comments. First, we could make a slight optimization and initialize the latch with one fewer signal and run one of

the iterations on the calling thread itself. This would avoid the overhead with queuing one work item. Second, we do not handle cases where the size of the loop is smaller than the size of p . For loops where this is expected to be true, we'd want to avoid parallelizing or change the division used because our current algorithm leads to the last partition running all loop iterations. It might even be possible that we'd want to use just the calling thread to execute the whole loop serially, for example, if we inspect the size of the loop and decide it's too small to be worthwhile. We also do not handle failures in the loop body at all. If an exception is thrown from body, it will go unhandled on a thread pool thread and will terminate the process; we'd probably prefer to rethrow the exception on the original thread to preserve the sequential loop semantics. This is trickier than it first appears, so we will return to this in its own section later in this chapter.

Our one line loop has suddenly become more than a dozen lines. Most of it is cluttered with the code to calculate various ranges of indices. This isn't difficult, but is easy to get wrong. A lot of it is boilerplate and can be reused from one loop to the next, which is why we've hoisted it all into a reusable function that accepts the body as an `Action<int>` delegate.

Why Simple Isn't Always Best. There are several reasons this approach is far from perfect.

One is that, if there's any possibility that the function `a` will block, we will waste a processor. Blocking calls are often not evident in the source—due to internal synchronization, in APIs and the Windows kernel itself, hard page faulting, among other things. As an illustration, say we have a 4-CPU machine, create 4 threads, and 1 of them blocks while running the loop; at some points during execution we would only be using 3 of the 4 available CPUs. It could even be that our loop would be using no CPUs at some point if all iterations block at once. In this case, we'd probably have liked to create more threads than the number of processors, or to have used a non blocking design.

Conversely, creating too many threads is not ideal because our program may not be eligible to run on all of the processors: if they are busy running other code, or if the process has been hard affinitized to use only a subset of the CPUs, we may incur unnecessary overheads due to the context

switches to use precisely 4 threads to run the loop. In such situations, we might prefer to create fewer threads than the number of processors, the reverse of the earlier situation. Worse, this situation is completely dynamic and unpredictable.

The approach of dividing iterations also has flaws. If every invocation of f costs the same (in terms of execution time), then having each thread execute an equivalent number of iterations seems ideal. But there's nothing that guarantees this balance. For example, imagine the implementation of the loop body we supply does something like this:

```
ParallelFor(..., delegate(int i)
{
    for (int j = 0; j < i; j++)
        /*... do something O(1) ...*/;
}, ...);
```

In this illustration, iterations become successively more expensive as the iteration number increases. Statically decomposing work as we did above would be a bad idea resulting in those threads running later iterations having to do substantially more work than threads running earlier iterations. Some threads would finish sooner than others. When we discuss critical paths in Chapter 14, Performance and Scalability, the gravity of this will become much clearer. But, in summary: the scalability of any given parallel algorithm is always limited by the piece of concurrent work that takes the longest to complete. While we would still possibly see a performance improvement due to the parallelism in such an unbalanced situation, it will not be the most impressive improvement we could have achieved. Soon we'll look at striping, which can balance the load of loop work more evenly, though it's still imperfect.

While there are some drawbacks to the contiguous partitioning approach, it is perhaps the simplest to comprehend and implement. The biggest drawback is the inherent inability to respond to information that may not become available until the code is running. This includes whether iterations block and/or the distribution of work among iterations, which

itself is usually not determinably statically. A decent compromise is to overdecompose the work. For instance, rather than choosing a value for p that is equal to the number of processors, choose twice the number of processors (or some other constant multiplier). While this is less efficient than the simple static partitioning shown earlier, when work never blocks and all iterations are equal, this perfect scenario seldom arises in practice. Experiment with different strategies for your particular workload and make decisions based on measurements.

Striped Iterations. Breaking the iteration space into contiguous iterations is not always the best solution. For instance, we saw a case above where the cost of loop iterations increases as the iteration number increases. But sometimes threads will terminate the iteration early (something we will discuss shortly when we look at cooperative algorithms), and it may make sense to have all threads iterating on lower (or higher) indices to minimize the possibility of wasted work.

As a real world illustration, imagine we want to find the first occurrence of an element in a list that satisfies some criteria. When a thread finds a candidate, we still cannot break out of the loop until all other threads have iterated up to the candidate element because it's possible they will find one earlier than the candidate. With the aforementioned partitioning approach, there is virtually no benefit to a thread finding a later element quickly. One solution is to use striping rather than contiguous iterations.

With striping, the input data is divided into many smaller chunks. As any given thread moves from one chunk to the next, it must "skip over" all other threads' chunks. Contiguous partitioning is a special case of striping where the chunk size is chosen carefully so that each thread has only a single chunk. The choice of chunk size is something that you will also have to decide. It often makes sense to choose a number that will result in aligned accesses, for example, if we're indexing into an array, we may choose a chunk size that, when multiplied by the size of the elements in the array, yields a size that is 128- or 64-byte aligned.

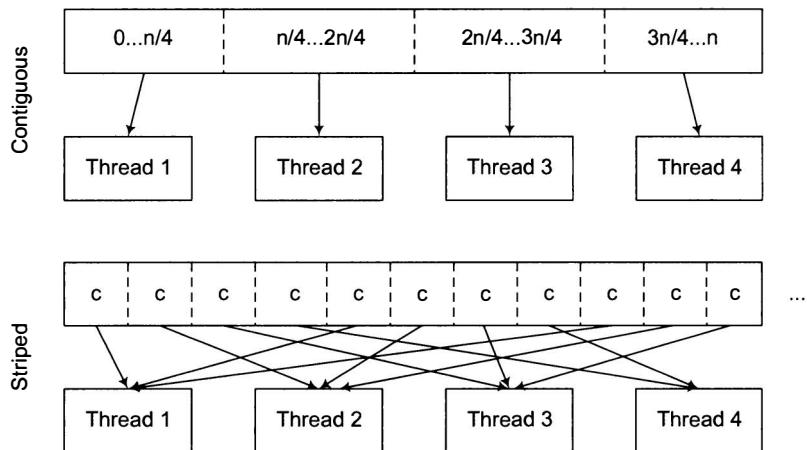


FIGURE 13.1: Contiguous and striped partitioning compared

The overall structure of the `ParallelFor` algorithm remains the same when striping is used, but a couple details, such as the calculation of indices during the per thread loop, change.

```

static void ParallelFor(int lo, int hi, Action<int> body, int p)
{
    const int chunk = 16; // Chunk size (constant)
    CountdownEvent latch = new CountdownEvent(p);

    // Schedule the threads to run in parallel
    for (int i = 0; i < p; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object procId)
        {
            int start = lo + (int)procId * chunk;
            for (int j = start; j < hi; j += chunk * (p - 1))
            {
                for (int k = 0; k < chunk && j + k < hi; k++)
                {
                    body(j + k);
                }
            }
            latch.Signal();
        }, i);
    }

    latch.Wait(); // Wait for them to finish
}

```

The only difference between this and the earlier chunking example is that we use two loops to enumerate the indices in a given chunk. The outer loop (with induction variable j) begins at a starting index of $lo + procId * chunk$ and continues until we reach hi . It increments j by $chunk * (p - 1)$ on each iteration, having the effect of skipping over all other threads' chunks each time that thread finishes with one of its own, as explained earlier. Then, beginning at that index, we enumerate the indices in the current chunk by using another inner loop (with induction variable k). We must make sure we also stay within the bounds of the loop by checking that $j + k$ is less than hi each iteration. All of the other details, such as how we initialize and signal the latch, call the function, and so forth, remain the same. And many of the same limitations explained above in the context of contiguous partitions also hold here.

Dynamic (On Demand) Decomposition

The previous approaches relied on an up front partitioning of the iteration space. As we noted, this can lead to imperfect utilization in cases where work blocks or is uneven. Overdecomposition was a suggested method for dealing with this. But there are other approaches too. One good approach for dealing with the uneven work problem is to dynamically decompose the iteration space by handing out chunks of work "on demand." This looks a lot like the striped iteration case seen earlier, with one difference: we need to use synchronization to communicate the current index among workers. It also handles loops that are not index based.

For Known Size Iteration Spaces. The first case we will look at is when the iteration space is of a known size, such as with a traditional `for` loop.

```
static void ParallelFor(int lo, int hi, Action<int> body, int p)
{
    const int chunk = 16; // Chunk size (constant)
    CountdownEvent latch = new CountdownEvent(p);
    int current = lo;

    // Schedule the threads to run in parallel
    for (int i = 0; i < p; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object procId)
```

```

    {
        int j;
        while (((j = (Interlocked.Add(
            ref current, chunk) - chunk)) < hi)
    {
        for (int k = 0; k < chunk && j + k < hi; k++)
        {
            body(j + k);
        }
    }
    latch.Signal();
}, i);
}

latch.Wait(); // Wait for them to finish
}

```

We have introduced a shared variable, `current`, that all threads use as a way of communicating the next chunk on which to begin working. Each thread calls `Interlocked.Add` on this shared location, incrementing it by `chunk` and ensuring that the current iteration still falls below the loop's upper bound, `hi`. (Notice that we subtract `chunk` from `Add`'s return value because `Add` returns the new value after the addition; we want to use the current value because that's what we'll use to start our iteration, that is, we want to start iterating at `lo` not `chunk`.) The inner loop looks identical to the striped iteration case shown before. (Also, for those unfamiliar with C# closures, the `current` variable is not a local variable; it is hoisted into a heap allocated closure object, and that is what gets shared among the threads.)

In this case, the size of `chunk` is not solely dependent on factors such as achieving good locality, although that is important here too. The `chunk` size also controls the frequency with which threads will attempt to write to a common memory location using an interlocked `Add` operation, which causes additional traffic in the memory system. Increasing the size can also be seen as a way of amortizing this communication. In summary, though, you should choose a size that is as small as needed to achieve your load balance goals, but no smaller.

You can also consider overdecomposition techniques in terms of how many threads to create, as mentioned above, due to the possibility of blocking and imbalance. With this approach, there is a high likelihood that future work items may become scheduled only to find that the `current` counter

has already reached `hi` because predecessor threads have finished all necessary iterations. It may be worth adding a check at the front of the work item for this condition.

Note also that a chunk size of more than 1 could perform poorly on loops with small sizes. If we have a 16-element array and a 16-processor system, it could be that invoking `body` on each element takes sufficiently long that parallelizing the loop by giving 1 element to each processor is worthwhile. The above example prohibits this because all 16 elements would be taken by the first processor to call `Add`. One solution to this problem that was suggested by a colleague of mine, is to have each thread start by taking 1 element, then 2, then 4, and so on, until it reaches its maximum chunk size. The code stays mostly the same, but the work queued to the thread pool differs ever so slightly.

```
static void ParallelFor(int lo, int hi, Action<int> body, int p)
{
    const int chunk = 16; // Chunk size (constant)
    ...
    ThreadPool.QueueUserWorkItem(delegate(object procId)
    {
        int j;
        int currChunk = 1;
        while ((j = (Interlocked.Add(
            ref current, currChunk) - currChunk)) < hi)
        {
            for (int k = 0; k < currChunk && j + k < hi; k++)
            {
                body(j + k);
            }
            if (currChunk < chunk) currChunk *= 2;
        }
        latch.Signal();
    }, i);
    ...
}
```

For dramatic overdecomposition and/or very large chunk sizes, the code written above suffers from possible integer overflow (because we call `Add` regardless of the value of `current`). The symptom—if checked arithmetic is not used—would be a loop that wraps back around to a negative number, causing unpredictable behavior. It is easy to rewrite this code to use `CompareExchange` and/or a range validation check to avoid overflow.

It would be less efficient but might be important for certain situations that demand high reliability.

For Unknown Size Iteration Spaces. Under some circumstances we can't deal in terms of indices. This makes things more difficult. For instance, imagine we have a .NET `IEnumerable<T>` and want to partition its contents so we can perform a data parallel computation on it. Instead of a `for` loop as shown earlier, the sequential code for this might take the form of a `foreach` loop in C#.

```
void For<T>(IEnumerable<T> e, Action<T> body)
{
    foreach (T e in enumerable)
    {
        body(e);
    }
}
```

The C# compiler expands this into a `while` loop that explicitly uses `IEnumerable<T>`.

```
void For<T>(IEnumerable<T> e, Action<T> body)
{
    using (IEnumerator<T> enum = e.GetEnumerator())
    {
        while (enum.MoveNext())
        {
            body(enum.Current);
        }
    }
}
```

Note that the C++ equivalent of this case is parallelizing some loop that uses a STL `std::iterator` object to perform its iteration.

```
template class<T> ...
void For(
    std::vector<T>::iterator it,
    std::vector<T>::iterator end,
    void (*body)(T))
{
    for (; it != end; it++)
    {
        *body(*it);
    }
}
```

We'll focus only on the .NET example below, but the point of showing the C++ code is to show that it's a similar problem.

How might we go ahead and parallelize this, given that we can't use indices to partition data? First, most enumerators are not thread safe, so it would be illegal for many threads to attempt to pull items from it at once. So it's not going to be as simple as letting all threads loose and racing to call `MoveNext` and `Current`. This implies we'll need to use some form of synchronization to protect concurrent access to the enumerator. In fact, the solution can be made to look a lot like the dynamic partitioning for loop indices shown previously, by allowing threads to accumulate "chunks" of data inside of a lock.

```
static void ParallelFor<T>(IEnumerable<T> e, Action<T> body, int p)
{
    const int chunk = 16; // Chunk size (constant)
    CountdownEvent latch = new CountdownEvent(p);
    IEnumerator<T> en = e.GetEnumerator();

    // Schedule the threads to run in parallel
    for (int i = 0; i < p; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object procId)
        {
            T[] elems = new T[chunk];
            int elemsCount = 0;

            do
            {
                // Under the lock, accumulate items in our buffer:
                lock (en)
                {
                    for (elemsCount = 0;
                        elemsCount < chunk;
                        elemsCount++)
                    {
                        if (!en.MoveNext())
                            break;
                        elems[elemsCount] = en.Current;
                    }
                }

                // Process the elements:
                for (int j = 0; j < elemsCount; j++)
                {
                    body(elems[j]);
                }
            }
        });
    }
}
```

```
        }
    }
    while (elemsCount == chunk);

    latch.Signal();
}, i);
}

latch.Wait(); // Wait for them to finish
}
```

Each thread allocates its own private array `elems` that can hold up to `chunk` elements at a given time. Then each one sits inside of a do-while loop, which is exited once the enumerator is found to be empty. Threads acquire a lock (using `en` as the lock) and, inside of the critical region, accumulate up to `chunk` items from the enumerator by calling `MoveNext` and remembering the `Current` element in its private array. Afterwards, `elemsCount` will be the number of elements taken, and it will invoke `body` on each element it took (if any). Notice that the loop termination condition occurs when the number of elements taken from the enumerator is fewer than the maximum that could have been taken; the only way this would arise is if a call to the enumerator's `MoveNext` function returned `false`.

Note that this technique generalizes easily to other kinds of loops that use predicates to determine when to exit a loop. For example, by replacing the call to `MoveNext` with the invocation of a `Func<bool>` and the call to `Current` with an invocation of a `Func<T>`, we could parallelize a `while` loop. There is one thing we must ensure, however: once the predicate evaluates to `false`, it will always subsequently evaluate to `false`. If this weren't the case, the loop may not terminate appropriately when expected.

Scalability of this algorithm is going to be far less attractive than the index approaches shown earlier, unless the work done per element is huge. The reason is that locking the enumerator is likely a significant scaling bottleneck. As the size of `chunk` increases, the amount of time each thread spends inside the critical region also increases (because the loop complexity depends directly on it). If `MoveNext` is simple—as would be the case with any .NET collection enumerators—then the cost per element can be expected to be fairly small; but if `MoveNext` is referencing a LINQ query that is streaming results from a database, for example, this code performs I/O

inside of a critical region. Also, larger chunk sizes mean that threads need to acquire the lock less frequently, which can aid in performance, but detracts from load balancing. Yet another factor that impacts the frequency of lock acquisitions is the cost of the function body, which is invoked for each element. As the number of threads increases, the contention at the lock also increases, meaning that for larger number of threads, bigger chunks may be better (assuming the cost of body outweighs that of `MoveNext`). In the end, there is no perfect answer other than to experiment for your particular scenarios.

If a data structure only offers an iterator based interface, it's often a better idea to take one of two approaches. One is to crack open its internals and devise your own data structure specific partitioning scheme. For instance, a binary tree may not offer an indexer, but it's almost certainly a better idea to partition it by handing out independent subtrees in a divide and conquer style approach than to rely on the generic enumerator based partitioning. Another alternative is to create your own data structure that allows for efficient partitioning.

Parallel Loops Applied: Mapping (or Projecting) Over Input Data

A common operation in functional programs is to map some operator over a source list to transform it into another list of the same size.

```
static U[] Map<T,U>(T[] input, Func<T, U> map)
{
    U[] output = new U[input.Length];
    for (int i = 0; i < input.Length; i++)
    {
        output[i] = map(input[i]);
    }
    return output;
}
```

This is functionally equivalent to LINQ's `Select` operator. Now that we have the tools above to perform parallel loops, it's simple to implement a `ParallelMap`.

```
static U[] ParallelMap<T,U>(T[] input, Func<T, U> map, int p)
{
    U[] output = new U[input.Length];
```

```

    ParallelFor(0, input.Length, i => output[i] = map(input[i]), p);
    return output;
}

```

This was simple because all iterations are inherently independent in a `map` operation.

One downside to this approach is that we must perform two delegate invocations for each element in `input`, rather than the original sequential implementation's one. One invocation occurs for the `map` delegate itself, while the other occurs for the `body` delegate passed to `ParallelFor`. For cases where work per element is small enough for this to matter, two particular optimizations can be considered. First, a handwritten parallel for loop that is specific to the `map` operation can be written. This avoids the extra invocation of the `body` delegate but at the cost of having to maintain a separate parallel for implementation. Second, the size of the `ParallelFor` iteration space can be divided by a certain constant, and each body can invoke `map` for a certain range of elements, amortizing invocations of the loop body delegate, again at the cost of implementation complexity.

```

static U[] ParallelMap<T,U>(T[] input, Func<T, U> map, int p)
{
    U[] output = new U[input.Length];
    const int stride = 16;
    ParallelFor(0, input.Length / stride,
        delegate(int i)
    {
        for (int j = 0; j < stride && (i + j) < input.Length; j++)
        {
            output[i+j] = map(input[i+j]);
        }
    }, p);
    return output;
}

```

This approach suffers from reducing the amount of latent parallelism available, which will possibly impact the speedup observed in practice. For situations where the input data size is very large, all individual invocations of `map` cost roughly the same, however, this approach should not tangibly impact the parallel efficiency (and should improve things).

Nesting Loops and Data Access Patterns

When loops are nested, there is an interesting decision to make. Considering a two-loop case, should we parallelize the outer loop, the inner loop, or both?

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        f(i, j);
```

As with most things, there isn't a simple one size fits all answer. In many cases, parallelizing the outer loop will yield the most benefit. This assumes that, in the above example, N is sufficiently large to expose enough parallelism to achieve a speedup. If N is less than the number of processors, for instance, then it is worth considering an alternative such as parallelizing the inner loop instead. Again, this assumes M is sufficiently large. If it isn't, then it may be worth at considering parallelizing both. (When it comes to the parallelization process, we can use the techniques we have already reviewed.) A word of caution: a naïve implementation of nested invocations of the above parallel loop examples will lead to terrible performance because the growth for units of work will be quadratic (i.e., $O(NM)$), and recursion and blocking will become a problem for many implementations (such as the thread pool, where such a scheme could easily lead to deadlock). There are alternative approaches.

One can “fuse” the inner with the outer loop, and then parallelize the single remaining loop. This exposes more information to the parallel loops implementation, so that it can more accurately partition the entire space of the iteration at once, rather than dynamically.

```
for (int i = 0; i < N*M; i++)
    f(i / M, i % M);
```

This is typically the best approach for such blatant nesting. It also leads to roughly the same cache access patterns as if the inner loop remained sequential.

It is also worth considering whether to rearrange the loop's structure. If the data access pattern of the body is such that parallelizing on the inner loop but executing the outer loop inside each thread will lead to better

cache efficiency, it may be desirable to first restructure the above loop into the following code before parallelizing the outer loop (or even applying the fusion technique).

```
for (int j = 0; j < M; j++)
    for (int i = 0; i < N; i++)
        f(i, j);
```

As an example of why you might care, imagine we were indexing into a matrix in the body of our loop. If the original inner loop (with *j* and *M*) controlled the row accessed and the original outer loop (with *i* and *N*) controlled the column, then partitioning on the row indices instead of the column would lead to better spatial and temporal cache locality for most dense matrix representations (e.g., CLR rectangular arrays, such as `int[,]`) due to the way individual elements in each row are stored adjacent to one another in memory.

Sometimes it may be useful to “tile” an array, for example, to assign $A \times B$ sections of the array to partitions at a time as the chunk unit size, such as 16×16 . This usually yields performance improvements due to locality and less frequent synchronization. In other circumstances, this kind of chunking might be a correctness condition of the algorithm. JPEG encoding, as an example, is a problem that can be parallelized (see Further Reading, Kodaka, Kimura, Kasahara), but requires that the input image be decoded into 8×8 chunks because of dependencies within individual chunks.

A plethora of additional loop restructurings is possible, often referred to by the general term **loop blocking**. The idea is to optimize loops, partitioning, and chunk sizes, based on the data access patterns of the code itself. Many exotic techniques have been explored over the years (see Further Reading, Lamport 1973; 1974), and much research has gone into the static optimization of such operations to achieve the best theoretical speedups (see Further Reading, Blelloch, Gibbons, Matias).

Reductions and Scans

A special kind of loop is one that reduces a whole list of values to a single scalar value, usually by applying a binary operator over the entire list. Computing the sum of a list of numbers is a fairly common programming

task, as is computing the average, finding the minimum or maximum element in a list, and so forth, all of which fall into this category. While these are just loops at their core (implementation-wise), we can take advantage of some special properties to represent them as so-called parallel reduction operations. We'd normally have trouble parallelizing such loops because they typically have one big loop carried dependency:

```
static int Add(int[] numbers)
{
    int sum = 0;
    for (int i = 0; i < numbers.Length; i++)
    {
        sum += i;
    }
    return sum;
}
```

This illustration reveals a problem: subsequent loop iterations depend on the writes made by all iterations prior to them. The intrinsic properties of such operations often allow us to work around this issue. The key is that many of the most popular kinds of reductions are associative and commutative. If these terms bring back nightmares from your high school math courses (as they do for me), here's a brief refresher: informally, an operator $+$ is associative if $(a + b) + c$ is equivalent to $a + (b + c)$, and commutative if $a + b$ is equivalent to $b + a$. Why does this matter? We can use this to partition the data, have multiple threads attack the same problem to achieve parallelism, and still yield the correct value at the end.

Taking this example, addition is both associative and commutative. It doesn't matter in what order we add numbers together, so long as each number is accounted for. We can, therefore, use the same techniques discussed earlier for partitioning the input and add up several thread local sums for each partition and, finally, add each partial sum at the end to yield the correct answer. This turns our $O(n)$ sum operation into $O(n/p + p)$, which is not a theoretical change but one that will practically yield a lot of benefit (particularly for large p). In order to reuse our `ParallelFor` API from earlier, we need one slight extension. Each thread is going to store its own partial sum, so it needs to know its task index out of the bunch. For illustration purposes, we will imagine a `ParallelFor` overload was available

that supplied the task's index (from 0 to $p - 1$) as the second argument to the body delegate, alongside the index itself.

```
static int ParallelAdd(int[] numbers, int p)
{
    // Compute partial sums:
    int[] partialSums = new int[p];
    ParallelFor(0, numbers.Length,
        (i, id) => partialSums[id] += numbers[i], p);

    // Compute final sum:
    int sum = 0;
    for (int i = 0; i < p; i++)
    {
        sum += partialSums[i];
    }

    return sum;
}
```

Some operations are nonassociative, which means we cannot use parallelism in this way. Yet others are noncommutative, which means that we can actually use parallelism but must take care to ensure that all combinations are done in the correct index order; that is, we must never swap the first and second arguments to the operator, when compared to sequential execution. A classic example is division, an operation that is associative but noncommutative.

Also note there is an inherent scalability limitation in the above example. At the end we have a sequential for loop from 0 to $p - 1$ that sums up the partial sums to produce the final answer. There are more scalable approaches to this step, the most popular being a so-called logarithmic reduction during which each thread adds two partial sums together at a time to produce half the number of partial sums, and so on, until only one sum remains. This yields a theoretical performance of $O(\log n)$, but this presumes an infinite number of processors. In reality, on the architectures Windows runs on (today) and given the small size of p compared to n , this approach does not perform nearly as well as the previous one, due to the high cost of synchronization, so we will omit any further discussion of it. For fine-grained parallelism hardware architectures that offer vector and

word level parallelism, such as those found in the supercomputing industry, however, it often makes sense to use such techniques.

Another data parallel technique related to reductions is called a scan. A scan is very much like a reduction except that the output of the operation is another list of values instead of a scalar. Each element i in the result is the partial reduction of the list, obtained by applying the particular binary operator to all elements $0 \dots i-1$ in the original list. In the case of a sum scan (also called the partial sums of a list), for instance, the tenth element contains the sum of elements 0 through 9, the eleventh contains the sum of elements 0 through 10, and so on. This seems like an inherently sequential problem, but again we can take advantage of associativity and commutativity in the same way we did to achieve parallelism (see Further Reading, Hillis, Steele).

Sorting

There are countless ways to sort a list. This is true of sequential software, and holds true for parallel software too. **Parallel quick-sort**, **parallel merge-sort**, **Batcher's bitonic sort**, and **radix sort** are just a few of the algorithms you can find written up in books and academic papers. Instead of spending a great deal of time comparing and contrasting the different approaches, let's look briefly at one particular technique: **parallel merge-sort**.

A parallel merge-sort works a lot like an ordinary merge-sort. The main difference is that we must partition the input among threads, have each of the threads locally sort their own copy, and each of the intermediary results must be merged. The individual sorts are perfectly parallel, but the merge step contains a fair bit of communication. This tends to be the limiting scaling factor for this particular algorithm and prevents it from achieving linear speedup. But it is the simplest to understand and implement, provided that you're somewhat familiar with the merge-sort algorithm already.

Before diving into the code, the two high level phases of the algorithm are as follows.

- We first split the input into p chunks. We use our `ParallelFor` construct to fork p workers, each of which uses the `Array.Sort` algorithm available in .NET to sort the arrays locally (using a quick-sort). Depending on the partitioning used, this may or may not lead to the

desired results. Chunking, for example, will prevent some tasks from running in parallel. We may be better off explicitly creating p tasks to ensure they run on separate processors.

- At this point, we have p sorted chunks. The next step is to merge them. This takes $\log p$ steps. Roughly speaking, adjacent tasks are paired up to merge: two tasks merge two arrays into one at a time. The logic for this is somewhat complicated: we ensure that both threads merge up to the midpoint in the array. Due to the way comparisons happen, we can be assured that this leads to an examination of all of the locally sorted inputs. At the end, we copy this intermediate result so the next phase in merging has access to the output.

Here is the code.

```
static T[] ParallelSort<T>(T[] input, int p) where T : IComparable<T>
{
    T[][] chunks = new T[p][];

    // Step 1: Sort the p chunks of the input.
    int chunk = input.Length / p;
    ParallelFor(0, p, delegate(int idx)
    {
        // Compute the bounds.
        int start = idx * chunk;
        int size;
        if (idx == p - 1)
            size = input.Length - start;
        else
            size = chunk;

        // Copy.
        chunks[idx] = new T[size];
        Array.Copy(input, idx * chunk, chunks[idx], 0, size);

        // And then actually sort.
        Array.Sort(chunks[idx]);
    },
    p);

    // Step 2: Merge the chunks.
    int remaining = p;
    while (remaining > 1)
    {
```

```
T[][] rchunks = new T[remaining][];
for (int i = 0; i < remaining; i += 2)
{
    if (i == remaining - 1 && (remaining & 1) == 1)
        rchunks[i] = chunks[i];
    else
        rchunks[i] = new T[
            chunks[i].Length + chunks[i+1].Length];
}

T[][] outchunks = new T[(remaining + 1) / 2][];
ParallelFor(0, remaining, delegate(int idx)
{
    // If an odd number, we just propagate the sorted chunk.
    if (idx == remaining - 1 && (remaining & 1) == 1) {
        outchunks[(idx+1) / 2] = rchunks[idx];
        return;
    }

    T[] dest = rchunks[idx & ~1];
    T[] left = chunks[idx & ~1];
    T[] right = chunks[idx | 1];
    int mid = (dest.Length + 1) / 2;

    if ((idx & 1) == 0)
    {
        // Even participants merge from left to right.
        int lix = 0; // left index.
        int rix = 0; // right index.
        int mix = 0; // merge index.
        for (int j = 0; j < mid; j++) {
            if (lix < left.Length &&
                left[lix].CompareTo(right[rix]) <= 0)
                dest[mix++] = left[lix++];
            else
                dest[mix++] = right[rix++];
        }
    }
    else
    {
        // Odd participants merge from right to left.
        int lix = left.Length - 1; // left index.
        int rix = right.Length - 1; // right index.
        int mix = dest.Length - 1; // merge index.
        for (int j = 0; j < mid; j++)
        {
            if (lix >= 0 && left[lix].CompareTo(right[rix]) > 0)
                dest[mix--] = left[lix--];
            else

```

```

        dest[mix--] = right[rix--];
    }
}

if ((idx & 1) == 0)
{
    // One of the partners propagates the result.
    outchunks[idx / 2] = dest;
}
}, remaining);

// Lastly, we know all threads are finished; propagate output.
for (int i = 0; i < outchunks.Length; i++)
    chunks[i] = outchunks[i];

remaining = (remaining + 1) / 2;
}

return chunks[0];
}

```

The code may look intimidating at first glance, but when broken down, it's straightforward. The two phases mentioned above translate into two separate calls to `ParallelFor`. The meat of the code is in the merging. In each merge step, the contents of two chunks are merged by two threads into a single `rchunks` array. Note that we use `idx & ~1` to get the even numbered partner for a pair, and `idx | 1` to get the odd numbered partner. This uses bitmasking to make code more concise and to allow for code sharing in the representation of the slightly different steps taken by odd and even numbered partners. Output is stored in a separate `outchunks` array, which is then propagated to `chunks` after the `ParallelFor` returns to avoid workers writing to `chunks` while others concurrently read.

Task Parallelism

Data parallelism is not always applicable to code that might be parallelizable. Often it is more natural to decompose a larger problem into independent and isolated smaller problems that can run in parallel with one another. This is often due to existing program structure. Imperative programs are organized as a collection of functions comprised of statements already, and it's often the case that sets of statements are independent

of one another and, hence, can benefit from parallelism. In other cases, statements may be dependent on each other, but in a way that can benefit from parallel execution. Unlike parallelizing for loops as shown earlier, task parallelism more frequently requires restructuring the original sequential algorithm's design so that the independent chunks of execution may be run individually.

With all that said, task parallelism inherently constrains the amount of latent parallelism in the program. Unlike data parallelism, where the dynamic size of the input data determines the upper bound on the number of processors that can be used to execute a program, task parallelism ordinarily statically limits the upper bound. This can lead to less scalable results.

Fork/Join Parallelism

The simplest instance of structured task parallelism involves a flat decomposition of a set of program operations. **Fork/join parallelism** is called such because it consists of two primary steps. The first step is the fork. When program execution reaches the fork, each operation in the set is scheduled to run in parallel. Sometime later, execution reaches the join step, which waits for forked parallel operations to complete. For instance, we may have a sequence of four independent method calls in our sequential program; running each of these calls simultaneously, one per processor, may be a fine way to achieve parallelism, provided that the work done by each method is significant. Moreover, fork/join is often great for encoding structured parallelism because the fork and join happen at once, that is, synchronously with respect to the caller.

Let's build a reusable fork/join construct, called `CoBegin`, which accepts an array of delegates and runs them in parallel. It can be built as a thin veneer over something like the thread pool, and we can start building other algorithms that depend on it.

```
CountdownEvent CoBegin(params Action[] actions)
{
    CountdownEvent latch = new CountdownEvent(actions.Length);
    for (int i = 0; i < actions.Length; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object obj)
```

```

    {
        try
        {
            actions[(int)obj]();
        }
        finally
        {
            latch.Signal();
        }
    }, i);
}
return latch;
}

```

This is pretty straightforward. All of the difficult synchronization is abstracted away inside the `CountdownEvent` primitive. We queue up a single thread pool work item for each delegate supplied by the caller, and return a handle that can be used to wait for all of the work items to complete. A nicer, more .NET-ish API might have returned an `IAsyncResult` for this purpose, but this is left as an exercise for the reader. (Building it isn't too difficult given the `SimpleAsyncResult<T>` class in Chapter 8, Asynchronous Programming Models.) Additionally, it might be useful to allow `Func<>` delegates to be supplied in cases where the parallel operations produce values of interest. Finally, exceptions during the invocation of the operations are not currently handled in any way—they will instead crash the thread pool thread on which the operation runs. Exceptions are discussed in depth at the end of this chapter.

With the `CoBegin` API, we can start a bunch of work and wait for it. Imagine we have a sequential program with independent function invocations of `B`, `C`, and `E`, and with dependent function invocations of `A`, `D`, and `F`, as follows.

```

T MyFunction()
{
    var a_val = A();
    B();
    C();
    var d_val = D(a_val);
    E();
    return F(d_val);
}

```

With a small amount of restructuring, we can offer the parallelism at the top of `MyFunction`'s definition, and wait for it before returning.

```
T MyFunction()
{
    CountdownEvent latch = CoBegin(
        () => B(),
        () => C(),
        () => E()
    );
    T f_val = F(D(A()));
    latch.Wait();
    return f_val;
}
```

Some assumptions have been made in this process. We assume the original ordering of function invocations, A, B, . . . , F, was mostly irrelevant. The original fictional program was not functional because the return values of B, C, and E have been ignored. This implies there is a good chance they are being executed for effect, and these effects may have subtle dependencies that are not evident from `MyFunction`'s definition alone. It could be the case that running them in parallel will expose race conditions, and/or that disturbing the ordering will change the behavior of the other function definitions, including the sequential ones A, D, and F. Because this is a purely fictional example, it matters very little, but it is brought up to reinforce the point that parallelizing a program goes far beyond the mechanisms required to do so.

It's quite common for fork/join parallelism to be lexically scoped. In other words, the fork and join happen at the same level in the program's lexical blocking, something called **structured fork/join**. This encourages a cleaner program design and reduces the chance of runaway parallelism and forgotten joins, which can lead to debugging problems. This would happen if the thread responsible for forking and joining happened to fail after the fork but before the join. There is no language construct that enforces this structure. However, we can build one easily by using our API and just doing the fork and join at once.

```
void DoAll(params Action[] actions)
{
    CoBegin(actions).Wait();
}
```

There is an obvious optimization to make here. Since we know that the thread will begin waiting immediately after invoking `CoBegin`, we could choose to run one action on the calling thread. This could be achieved by removing one action from the `actions` array passed to `CoBegin` and executing it after the call but before the call to `Wait` on the returned latch.

```
void DoAll(params Action[] actions)
{
    Action[] parallelActions = new Action[actions.Length - 1];
    Array.Copy(actions, parallelActions, actions.Length - 1);
    CountdownEvent latch = CoBegin(parallelActions);
    try
    {
        actions[actions.Length - 1]();
    }
    finally
    {
        latch.Wait();
    }
}
```

The caller that initiates the fork is now no longer running in parallel with the other operations. It blocks until all parallel work completes. If we return to the `MyFunction` example from earlier, it is a perfect candidate for `DoAll`, but we must restructure it slightly so that the previously sequential portion is offered as an action that runs in parallel with the others.

```
T MyFunction()
{
    T f_val = default(T);
    DoAll(
        () => B(),
        () => C(),
        () => E(),
        () => f_val = F(D(A())))
    );
    return f_val;
}
```

The behavior of this is effectively the same as the one shown earlier, that is, `F(D(A))` runs on the calling thread and all others delegate in parallel, but it leads to a more structured program.

Dataflow Parallelism (Futures and Promises)

Managing the sequence of events that happen in a parallel system takes some effort. We have seen earlier that data parallelism removes the need to encode this specific information, as it ends up being a byproduct of the data access patterns employed. Intelligent infrastructure, such as a `ParallelFor` function, can hide most of the difficult error prone decisions. We've now seen that task parallelism makes things slightly more complicated because the decision about when, where, and how to wait for things to occur is much more imperative in style. This style more easily leads to programming errors and bugs.

An alternative programming style to both of these, but closely related, is called **dataflow parallelism**. In dataflow algorithms, the decisions about waiting are encapsulated inside simple to use abstractions that hide the tedious work of managing waiting on and signaling events. Moreover, the coordination between threads is entirely derived from the way in which data is produced and consumed by agents in the systems. There are two closely related abstractions commonly used to build such dataflow systems: **futures and promises**.

Futures

A future is an object logically representing a value that is calculated at some unspecified point. It may have already happened, or it may happen at some point in the future. When a future's value becomes available, we say it has been "resolved." Code may request the value from a future, in which case it's up to the implementation to decide what to do. One reasonable approach is to wait for the future to execute. This is the simplest approach. Yet another reasonable approach is to execute the work on the thread requesting the value, resolving the future, assuming the future hasn't yet begun executing. This is called a **lazy future**.

Futures have been in existence since the late 1970s where they were first used in the context of garbage collection and argument evaluation order and then heavily in actor based systems meant for building medium- to coarse-grained asynchronous agents style programs (see Further Reading, Baker,

Hewitt). These systems were mostly done in the context of the MIT Scheme language. They have been subsequently used in many other programming environments, including mainstream ones like Smalltalk and Java. Perhaps the most pervasive use of them is in the functional language Alice ML (see Further Reading, Lieberman) and the programming languages Joule and E, where they are a first class and pervasive abstraction used in nearly every program written.

A common use for the future abstraction is to turn a synchronous API into an asynchronous one while still maintaining a very synchronous feel to it. Futures can be used in this manner to hide latencies such as those associated with I/O, or instead to achieve a parallel speedup for computationally intensive work, as the generation of the future's value occurs in parallel with respect to the requestor of the values. In any case, the API that is responsible for producing a value can return a future object in its stead (or an array of future objects) that is a "stand in" for the value that is to be created. The user of such an API can be confident the value(s) will be available if and when they are eventually needed.

Futures are a form of unstructured concurrency and are, therefore, somewhat more difficult to use, particularly when it comes to debugging runtime interactions among threads. They work best when the work done to compute a value is purely functional (i.e., doesn't have side effects and does not depend on shared, mutable state), though this is hard to guarantee in the kind of imperative languages common to Windows. Returning futures from an API also complicates the API design slightly because it must handle cases where subsequent invocations are made while futures for prior invocations are still outstanding and haven't yet resolved.

There is no future type available in the .NET Framework today, but it's simple to build one. We will use generics, so the type will be called `Future<T>`. It needs two things: a way to construct it, accepting a `Func<T>` delegate that will compute the value, and a `Value` property to access said value. The capability to lazily resolve a future on the calling thread if it has not yet begun executing is optional to the core future abstraction, but interesting enough that we will support it in our type here.

```
public class Future<T>
{
    private volatile int m_state = 0; // 0=unstarted, 1=running, 2=done
    private T m_value;
    private volatile Exception m_exception;
    private Func<T> m_func;
    private ThinEvent m_event = new ThinEvent(false);

    public Future(Func<T> func)
    {
        m_func = func;
        ThreadPool.QueueUserWorkItem(s_callback, this);
    }

    public T Value
    {
        get
        {
            if (m_state != 2 && !TryRun())
                m_event.Wait();
            if (m_exception != null)
                throw m_exception;
            return m_value;
        }
    }

    private static WaitCallback s_callback = Run;
    private static void Run(object obj) { ((Future<T>)obj).TryRun(); }

    private void TryRun()
    {
        if (m_state == 0 &&
            Interlocked.CompareExchange(ref m_state, 1, 0) == 0)
        {
            try
            {
                m_value = m_func();
            }
            catch (Exception e)
            {
                m_exception = e;
            }
            finally
            {
                m_state = 2;
                m_event.Set();
            }
        }
    }
}
```

Internally, the future type maintains a `m_state` field that can hold three values: 0 means the future has not begun executing, 1 means it is currently running, and 2 means it is complete. The `m_value` holds the value once it has been computed, and `m_exception` holds a reference to an exception object in case there is a problem while the future runs. Some fields are marked volatile to ensure reads of them are not reordered with respect to one another, which could cause issues in the `Value` property: for example, otherwise we might see `m_state` as 2 but subsequently read `m_value` as `null`. We remember the function in `m_func` so that we can invoke it later, and we use `m_event` to support waiting if it is needed. Notice that we use a `ThinEvent` type instead of a real event: this is meant to lazily allocate any needed kernel resources. A real `Future<T>` implementation probably ought to lazily allocate this object itself (since waiting should be rare) and consider implementing `IDisposable` so that the lazily allocated kernel resources can be cleaned up deterministically by users of our class.

Most of the magic happens in the `TryRun` method. It handles resolving the future's value. When the future is scheduled (from the constructor via `QueueUserWorkItem`), it shunts over to the `Run` method, which is a wrapper over `TryRun` that conforms to the expected thread pool delegate signature. This function is also called from the `Value` accessor when it is called before the future value has been published (i.e., `m_state` is not yet 2). `TryRun` immediately attempts to "steal" the future by changing `m_state` from 0 to 1. Whichever thread succeeds—and only one will—goes ahead and invokes the `m_func` delegate, storing its return value in `m_value`. If an exception occurs, it is stored in the `m_exception` field. The thread then sets `m_state` to 2 so subsequent accesses can just retrieve the value and sets `m_event` in the `finally` block to signal to any threads that have begun waiting.

The `Value` accessor does the right thing when it comes to propagating the exception or returning the future's value, depending on the state of the future object. There is a major downside to the way we handle exceptions: we destroy stack traces by saying `throw m_exception`, and the thread (along with all its locals) that ran `m_func` and encountered an exception will be long gone by the time another thread waits on the future. These are admittedly substantial flaws. We'll return to the topic of exceptions later in this chapter.

Promises

The future abstraction above tightly couples the logical fact that a value is to be generated (possibly concurrently) in the future with the specific mechanism used to resolve it. There is no way offered to decouple the two. In other words, in the `Future<T>` type we created, a function is always scheduled to execute on the thread pool for each new future object created. It is sometimes useful to have one without the other, that is, to allow a thread to wait on the generation of a value and for another to set the value in an unstructured way. Additionally, the only way to extract a value is to block waiting for it. Instead of doing this, it can often be preferable to queue a continuation that will execute once the value is bound.

The combination of both is often called a **promise** (see Further Reading, Liskov, Shrira). The line is quite blurred between a future and a promise, and many people (and indeed systems that have implemented both) have their own subtle differences. One could reasonably argue they are the same thing, and simultaneously one could reasonably argue they are worlds apart from one another. Nevertheless, these two new concepts are useful.

The implementation of the first idea ends up looking a lot like the `Future<T>` type above. In fact, were we interested in providing a cleanly factored type hierarchy, we might even consider unifying the two ideas. But here is a sample standalone `Promise<T>` type:

```
public class Promise<T>
{
    private volatile int m_state = 0; // 0=unstarted, 1=running, 2=done
    private T m_value;
    private volatile Exception m_exception;
    private ThinEvent m_event = new ThinEvent(false);

    public Promise() {}

    public T Value
    {
        get
        {
            if (m_state != 2)
                m_event.Wait();
            if (m_exception != null)
                throw m_exception;
            return m_value;
        }
    }
}
```

```
        set
    {
        Set(value, null);
    }
}

public void Fail(Exception exception)
{
    Set(default(T), exception);
}

private void Set(T value, Exception exception)
{
    if (m_state == 0 &&
        Interlocked.CompareExchange(ref m_state, 1, 0) == 0) {
        m_value = value;
        m_exception = exception;
        m_state = 2;
        m_event.Set();
    }
    else
    {
        throw new InvalidOperationException("Can only set once");
    }
}
```

We will omit many details from the discussion, since the implementation is quite similar to the previous future implementation. A few differences are worth pointing out. We offer a setter for the `Value` property, which delegates to the internal `Set` method, passing `null` for the exception argument. We also provide a `Fail` method used to communicate exceptions from the one providing the promise's value to the consumer. This also uses the `Set` method, passing `default(T)` for the value argument. All of the interesting logic happens inside of `Set`. We first ensure only one thread ever attempts to set the promise using a similar technique to the future (i.e., checking that `m_state` is `0`)—throwing an `InvalidOperationException` otherwise. Else, we just store the values into the fields, set the event, and we're done.

Because promises don't bake in any sort of scheduling policy, they can be used to build facades on top of existing infrastructure. For example, we could build an API that wraps the existing asynchronous I/O BCL functions exposed in `System.IO.Stream`.

```
Promise<byte[]> ReadChunk(FileStream fs, int size)
{
    Promise<byte[]> p = new Promise<byte[]>();

    byte[] bb = new byte[size];

    fs.BeginRead(bb, 0, size,
        delegate(IAsyncResult iar)
    {
        try
        {
            int read = fs.EndRead(iar);
            if (read != size)
            {
                byte[] bb2 = new byte[read];
                Array.Copy(bb, bb2, read);
                bb = bb2;
            }
            p.Value = bb;
        }
        catch (Exception e)
        {
            p.Fail(e);
        }
    }, null);

    return p;
}
```

While this offers little more than the existing `IAsyncResult` object returned by `BeginRead` (and other asynchronous programming model APIs), we will be building some additional features on top of promises that come in useful. Moreover, `Promise<T>` could easily implement the `IAsyncResult` interface if we chose to do so. The abstraction is a superset of the minimum functionality required by implementers of this interface.

Resolve Events vs. Blocking

We've implemented the first half of the promise idea. However, the coupling of blocking with the communication of value availability is worth revisiting. In the above types, we have made blocking a non-negotiable part of both types' `Value` property semantics. Clearly supporting a way of polling for the availability of a value so that a thread can decide not to block would be

useful, as would a timeout variant that waits for at most a specified period of time. However, blocking is often a bad idea to begin with.

We can work around blocking by using an event driven approach that encourages continuation passing to represent work to be done once a value has been resolved. Using this approach, a thread can queue a delegate to be invoked asynchronously once the value has been resolved, and the future or promise itself handles dispatching these work items. Since it is more general purpose, we will extend the `Promise<T>` type above to support this capability, via a new `When` API. It accepts an `Action<T>` that is to receive the resolved value once it is available.

As an illustration, say we have a promise that was generated via the `ReadChunk` API above and want to do some analysis on the `byte[]` read off the disk once it becomes available. The traditional approach would be to block waiting for it.

```
FileStream myFs = ...;
Promise<byte[]> p = ReadChunk(myFs, 4096);
// ... do other work ...
// Some time later when we want the value, we must wait for it ...
ProcessBytes(p.Value);
```

If we wrote this using `When` instead, we can immediately schedule the `ProcessBytes` to happen when the promise resolves and avoid all blocking. Additionally, there wouldn't be potential for arbitrary execution delays caused by the thread that will call `ProcessBytes` taking too long in the "... do other work ..." portion of its body.

```
FileStream myFs = ...;
ReadChunk(myFs, 4096).When(bb => ProcessBytes(bb));
```

Here is an example implementation of `When`. Only the changed portions are shown.

```
public class Promise<T> ...
{
    private Queue<Action<T>> m_resolveActions = new Queue<Action<T>>();

    ... as before ...

    public void When(Action<T> resolveAction)
    {
```

```
lock (m_resolveActions)
{
    if (m_state == 2 && m_exception == null)
        ThreadPool.QueueUserWorkItem(delegate {
            resolveAction(m_value);
        });
    else
        m_resolveActions.Enqueue(resolveAction);
}
}

private void Set(T value, Exception exception)
{
    if (m_state == 0 &&
        Interlocked.CompareExchange(ref m_state, 1, 0) == 0)
    {
        m_value = value;
        m_exception = exception;
        m_state = 2;
        m_event.Set();

        lock (m_resolveActions)
        {
            if (m_exception == null)
                foreach (Action<T> a in m_resolveActions)
                    ThreadPool.QueueUserWorkItem(delegate
                    {
                        a(m_value);
                    });
            m_resolveActions.Clear();
        }
    }
    else
    {
        throw new InvalidOperationException("Can only set once");
    }
}
```

We have added a new queue of completion actions, `m_resolveActions`, containing all of the registered delegates. It's worth considering lazily allocating this queue, particularly if `When` will only be called on a subset of `Promise<T>` objects. Once a caller invokes the `When` API, we lock on the actions queue (since many threads may try to access it at once); once inside the critical region, we will do one of two things: if the work has finished already, we immediately queue the work to run on the thread pool, otherwise we just add

the action into the queue. Then we make an addition to the `Set` method: after changing our `m_state` to 2, we lock on the actions queue and queue each to the thread-pool. Notice one thing: we never execute the completion actions if an exception was generated. It's worth considering whether to extend the `When` capability to accept `Action<T, Exception>` delegates, or to offer a separate API such as `WhenFail` that handles the exception continuations.

Future and Promise Pipelining

Now that we have the above capabilities, a natural extension is to pipeline the output of one future or promise to another future or promise. This chaining of dataflow dependencies can be quite useful and avoids having to block at several levels of dependence. In our earlier file I/O example, what if it was the case that `ProcessBytes` itself generated a value of interest? For instance, maybe it analyzes the `byte[]` array and returns a computed `int` based on some sophisticated analysis and computation.

In this situation, this is the code we might like to write.

```
FileStream myFs = ...;
Promise<byte[]> p0 = ReadChunk(myFs, 4096);
Promise<int> p1 = p0.When<int>(bb => ProcessBytes(bb));
... use p1 in some way ...
```

This is similar to our initial example, but for readability, the construction of the individual promises has been placed on separate lines.

It turns out that this is simple to enable with a new version of `When`.

```
public class Promise<T> ...
{
    ... as before ...

    public Promise<U> When<U>(Func<T, U> resolveFunc)
    {
        Promise<U> p = new Promise<U>();
        When(delegate(T val) { p.Value = val; });
        return p;
    }
}
```

As an extension of this example, imagine that we would like to chain the processing of the entire `FileStream`'s contents, combining values from the

calls to `ProcessBytes` in some way. For illustration purposes, let's imagine we want to add all the values together. A sequential approach to the scheduling of these operations might look like this:

```
FileStream myFs = ...;
int finalValue = 0;
Promise<byte[]> p;
do
{
    ReadChunk(myFs, 4096);
    ...
    finalValue += ProcessBytes(p.Value);
}
while (p.Value.Length == 4096);
```

This suffers from all the same drawbacks as the earlier example used to motivate `When`. With the new overload of `When` to enable pipelining of promises, we can create a sort of recursive pipeline of promises to handle this task.

```
FileStream myFs = ...;
Promise<int> finalValue = new Promise<int>();

Func<int, Action<byte[]>> cont = null;
cont = delegate(int curr)
{
    return delegate(byte[] bb)
    {
        if (bb.Length == 4096)
        {
            Promise<byte[]> bb2 = ReadChunk(myFs, 4096);
            bb2.When(cont(ProcessBytes(bb) + curr));
        }
        else
        {
            finalValue.Value = ProcessBytes(bb) + curr;
        }
    };
};

ReadChunk(myFs, 4096).When(cont(0));
```

This chains the reading and analysis of the entire file together into one string of dataflow operations, exposing the final result in the `finalValue` promise. The code that needs this value can go ahead and do what it wishes with the value, including scheduling a continuation via `When` to do something

with it, such as rendering the result to the UI. This implementation may be a little difficult to follow at first, since we're using a closure to capture some intermediate state that needs to get passed along for each completion event. Let's review it a little more closely.

The `finalValue` promise is first constructed at the top. We then define the `cont` delegate. It is typed as `Func<int, Action<byte[]>>`, which means it is a delegate that accepts an `int` argument and, when invoked, returns an action that processes a `byte[]`. It generates delegates that will be registered with the `When` function. We have pulled it out, as noted above, because each unique registration needs to pass a different value for `curr`.

(Notice that we first assign `null` to the `cont` local. This may look strange, but is done to work around a tricky issue with C#: we need to access `cont` recursively from within its own definition, but C# does not allow this since `cont` wasn't declared previously. If we just tried to assign it outright we would encounter a compiler error. The way it has been written eliminates the compiler error—and it's safe too, since by the time the delegate is invoked, `cont` will have been assigned a value.)

This delegate constructs and returns an inner delegate referencing an anonymous method. That inner method does one of two things. If the length of the `byte[]` supplied by the `ReadChunk` promise is 4096, the end of the file has not yet been reached. It responds by creating yet another promise for the next chunk in the same way, and then scheduling a `When` continuation for that promise. The delegate is constructed with a call to `cont`, and the `int` argument is the result of adding `curr` to the return value of `ProcessBytes`. This executes after the asynchronous I/O has already been initiated. If the length of the `byte[]` is less than 4096, on the other hand, the end of the file has been reached. We compute `ProcessBytes` for this chunk, add the value to `curr`, and then publish it to the `finalValue` promise.

Since this example is a bit mind bending, we might encapsulate all of this into a simpler API.

```
public class Promise<T>
{
    ...
    as before ...

    public Promise<V> WhenReduce<U, V>(
```

```

    U seed,
    Func<Promise<T>> promiseGenerator,
    Func<U, T, U> combine,
    Func<T, bool> continuePredicate,
    Func<U, V> resultSelector)
{
    Promise<V> finalValue = new Promise<V>();

    Func<U, Action<T>> cont = null;
    cont = delegate(U curr)
    {
        return delegate(T v)
        {
            if (continuePredicate(v))
            {
                Promise<T> p = promiseGenerator();
                p.When(cont(combine(curr, v)));
            }
            else
            {
                finalValue.Value = combine(curr, v);
            }
        };
    };
};

When(cont(seed));

return finalValue;
}
}
}

```

Given this API, we could encode our previous example as follows.

```

FileStream myFs = ...;
Promise<int> finalValue = ReadChunk(myFs, 4096).
    WhenReduce<int, int>(
        0,
        () => ReadChunk(myFs, 4096),
        (c, bb) => c + ProcessBytes(bb),
        bb => bb.Length == 4096,
        c => c
    );

```

This is slightly less mind bending, but still takes a fair bit of thought to follow. The `resultSelector` is unnecessary in this particular case, but often it's not—that is, it's useful to be able to do “one last step” before publishing the value. It's safe to say that this kind of dataflow programming, while

intellectually intriguing and useful in some circumstances, is more difficult to write, read, and debug. It is typically more useful for hiding latency and composing together concurrent operations than achieving parallel speedups. As you can see, it's hard to track all of the hidden object allocations, delegate invocations, lock acquires, etc., as the abstractions are used more and more liberally, particularly in the recursive and compositional cases.

Recursion

Many algorithms are better implemented using recursion than with looping constructs. This can be either due to the nature of the algorithm itself—such as mergesort, an inherently recursive algorithm—or because it is simply a convenient way of representing and processing certain kinds of problems and data structures—such as traversing a tree and doing something with each of its nodes.

Whatever the case, individual recursive calls are often completely independent of other recursive calls in a tree of computations. For example, the whole point of divide and conquer is to continually divide a problem space into smaller and smaller disjointed pieces so that they can be solved independently, combining results as the recursion unwinds. This is conducive to parallel execution of the individual parts. In other nonembarrassingly parallel cases, some or all of the recursive calls share state, such as fields of shared objects, at which point all of the state management issues we've outlined earlier must be taken into account. Without attention and care, this often leads to recursive lock usage, which is a bad idea for all the reasons outlined in Chapter 11, Concurrency Hazards.

For what it's worth, recursion usually straddles the line between data and task parallelism. In some cases, the depth of recursion and division of work is driven solely by the characteristics of the data being operated on, in which case recursion truly is a data parallel mechanism. In other cases, the recursion may be completely program structure dependent and have nothing to do with data, in which case it appears as a task parallel problem. Categorization aside, we discuss recursion in the task parallelism section because it is most typically reified using task parallel constructs; in fact, we'll make use of some of the task capabilities we just reviewed in the preceding paragraphs.

As an example of a simple recursive algorithm, imagine we have a binary tree and would like to mirror it in place. That is, for each node in the

tree, we would like to swap its left and right child subtrees with one another. This is easy to parallelize, since there are no dependencies at all in the individual recursive calls and can be done in a divide and conquer style. It is important that we ensure no two threads try to mirror the same node's children at once, which is done by virtue of the fact that the unit of work is an independent node. For a graph that might have cycles, this would be far more difficult to do, perhaps requiring fine-grained node locks.

The sequential version might look like this.

```
class TreeNode
{
    internal TreeNode left;
    internal TreeNode right;
}

void Mirror(TreeNode node)
{
    if (node == null) return;

    Mirror(node.left);
    Mirror(node.right);

    TreeNode tmp = node.left;
    node.left = node.right;
    node.right = tmp;
}
```

Parallelizing this algorithm is quite straightforward given our earlier definition of DoAll.

```
void ParallelMirror(TreeNode node)
{
    if (node == null) return;

    DoAll(
        () => ParallelMirror(node.left),
        () => ParallelMirror(node.right)
    );

    TreeNode tmp = node.left;
    node.left = node.right;
    node.right = tmp;
}
```

If, instead of performing side effects, the recursive function needed to compute values, we might consider using the Future<T> abstraction we

created above instead. Executing this algorithm generates a tree-like structure of dependent computations, as shown in Figure 13.2.

This entire problem could be generalized to any kind of binary traversal (or even arbitrary traversals) by adding more delegate invocations.

```
void Traverse<T>(
    T curr, Action<T> body, Func<T, T> left, Func<T, T> right)
{
    if (curr == default(T)) return;

    DoAll(
        () => Traverse<T>(left(curr)),
        () => Traverse<T>(right(curr))
    );

    body(curr);
}
```

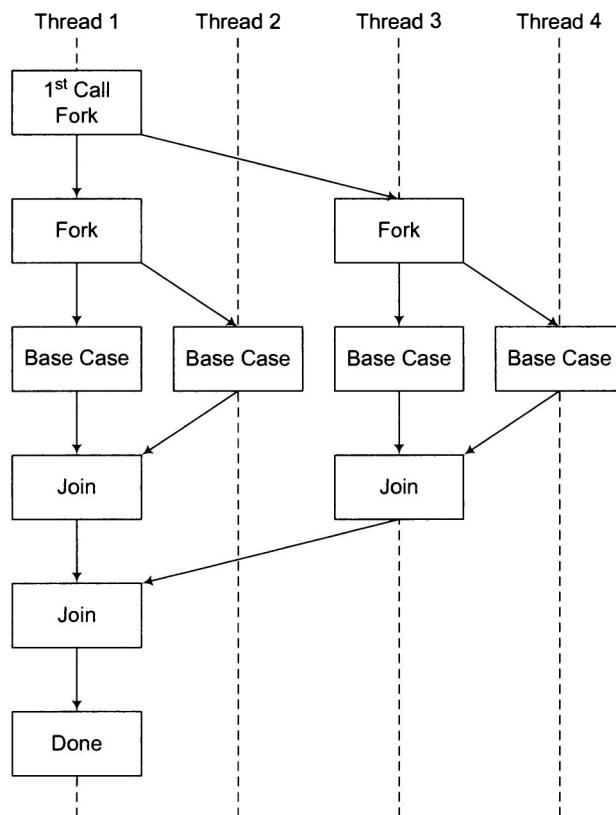


FIGURE 13.2: Graphical depiction of divide and conquer parallelism

The `ParallelMirror` method can now be written in terms of `Traverse<TreeNode>`.

```
void ParallelMirror(TreeNode node)
{
    Traverse<TreeNode>(
        node,
        n => {
            TreeNode tmp = node.left;
            node.left = node.right;
            node.right = tmp;
        },
        n => n.left,
        n => n.right,
    );
}
```

Now the question is: Would this trivial parallelization actually yield a benefit?

Maybe. There are overheads involved in performing this operation in parallel. The first obvious one is the delegate invocation for each recursive call versus the static call to the `Mirror` function directly. Additionally, a new `CountdownEvent` is internally allocated for each call to `DoAll`, and there are a couple calls to `CountdownEvent` APIs that may or may not result in interlocked operations and waits. And let us not forget the extra work done to enqueue work into the thread pool's work queue via `QueueUserWorkItem` and the latency between the time of queuing it and a CLR thread pool thread seeing it.

A far less obvious and worse dilemma is that this program will probably deadlock on the current CLR thread pool. At the very least, it will cause terrible performance degradation. The reason is that, aside from the first call to `ParallelMirror`, all subsequent executions will be running on thread pool threads. These calls wait for subsequent executions of work, requiring additional threads to free up in order to run them. Depending on the exact size of the processor count and the thread pool's maximum thread count, those executions may never get scheduled because the threads needed to run them are blocked.

A lot of this overhead could be avoided or mitigated with changes to our `DoAll` primitive (including lazy allocation of resources) and representation of the problem. This includes doing the following.

- We could use a threshold to stop parallel recursion at a certain depth in the tree traversal. When we reach this threshold, we switch over to calling the sequential implementation of `Mirror` rather than `ParallelMirror`. For large trees, this still allows for a great degree of parallelism, without many of the inefficiencies noted above. For instance, we may choose a depth of $\log_2 p$ where p is the number of processors on the machine, ensuring that we don't create more parallel units of work than there is hardware available to execute them.

This approach has several disadvantages in the general case, including being an overly static and restrictive form of problem decomposition similar to the static loop iteration cases noted before. This comes up as a practical issue in this particular case because there are no guarantees about whether a tree is balanced or not. A very unbalanced tree will lead to some workers doing vastly more work than others, dramatically reducing the amount of speedup we can expect to see.

- We could use an up front partitioning phase before doing the traversal of the tree structure. This phase could decide *a priori* which threads will work on which subparts of the tree and then assign the resulting units of work. One technique is to use a breadth first search starting at the root, sequentially, and proceeding until we have accumulated enough nodes to partition fairly across the threads.
(We probably don't want to traverse the entire tree in this phase. That would be pointless in the mirroring case stated above because a substantial portion of the work in this algorithm is the traversal itself. But, if work per node is sufficiently large, the benefits of load balancing may outweigh the drawbacks of this initial traversal.) We would then use a `ParallelFor` style loop to kick off the recursive algorithm sequentially on each thread.

This approach also has a number of downsides. The first is the obvious complexity and changes required to the original algorithm. We must also be careful that no two threads attempt to process the same regions of the tree simultaneously, which is harder since we need to ensure that a thread operating on a node doesn't access the ancestor or child tree which might be being actively processed by other threads.

Recursion encodes dependence in the program. And finally, it may or may not solve the fairness issue detailed before because the calculations required to perform a fair partitioning may end up being a substantial amount of work, offsetting any potential gains.

- We could dynamically monitor the number of nodes actively being processed, that is, by maintaining an “actively running” counter and then switching between sequential and parallel processing more dynamically. Many dynamic work stealing systems do this automatically. This incurs more overhead for runtime checking and is still not perfect because decisions tend to be “greedy,” which can lead to depth first parallelization over breadth first (the former usually tends to be more efficient), though we can offset that by combining this approach with the first.

Let’s illustrate the hybrid approach mentioned in the previous paragraph. First, we will use static decomposition to achieve good breadth first parallelization, and then, within each of those partitions, we will use the dynamic “active running” counter to scale up to a factor of the number of processors on the machine.

```
readonly int c_scaleUpTo = Environment.ProcessorCount * 2;

void ParallelMirror(TreeNode node)
{
    int active = 0;
    ParallelMirror(
        node,
        (int)Math.Log(Environment.ProcessorCount, 2),
        ref active);
}

void ParallelMirror(TreeNode node, int threshold, ref int active)
{
    if (node == null) return;

    if (threshold == 0 && active >= c_scaleUpTo)
    {
        Mirror(node.left);
        Mirror(node.right);
    }
    else
```

```

    {
        Interlocked.Increment(ref active);
        int newThreshold = threshold == 0 ? 0 : threshold - 1;
        DoAll(
            () => ParallelMirror(node.left, newThreshold, ref active),
            () => ParallelMirror(node.right, newThreshold, ref active)
        );
        Interlocked.Decrement(ref active);
    }

    TreeNode tmp = node.left;
    node.left = node.right;
    node.right = tmp;
}

void Mirror(TreeNode node, ref int active)
{
    if (node == null) return;

    if (active < c_scaleUpTo)
    {
        ParallelMirror(node, 0, ref active);
    }
    else
    {
        TreeNode tmp = node.left;
        node.left = node.right;
        node.right = tmp;
    }
}

```

In summary, we begin the computation in `ParallelMirror` by forwarding to the more specific overload, initializing threshold to $\log_2 p$, where p is the processor count, and passing a byref to a stack local `active` variable that has been initialized to 0. As before, each recursive parallel call still decrements the threshold by 1. This is where it gets a more difficult. Inside of `ParallelMirror`, we have modified the threshold detection logic to switch to sequential processing in the `Mirror` method if both the threshold of the current call is 0 and the `active` variable is greater than or equal to `c_scaleUpTo`. This deserves some explanation. Surrounding each call made to `DoAll`, which may introduce parallelism, we increment and decrement the `active` variable (by 1). This has the effect of permitting more dynamic parallelism: in our case, roughly twice the number of processors (since `c_scaleUpTo` is defined as `Environment.ProcessorCount * 2`). Notice also that the sequential `Mirror` API also checks the `active` variable! If it ever

sees it below `c_scaleUpTo`, it forwards back to the `ParallelMirror` API so that additional parallelism may be introduced.

This approach is not perfect, but it should produce decent results. Depending on the frequency of blocking inside of the processing logic, we might want to use a factor higher than 2 in the definition of `c_scaleUpTo`. One subtle issue in this code is that the reads of `active` are not guarded with any thread safety. It's possible, then, to introduce more parallelism than `c_scaleUpTo` if multiple threads see `active` below `c_scaleUpTo` and then go ahead and increment it. We could get around this by using `Interlocked.CompareExchange`, although that will lead to some degree of spinning and contention. Whether this is better depends on the penalties incurred by oversubscribing the processors. This can also be the source of ping-ponging between `ParallelMirror` and `Mirror`; imagine `ParallelMirror` sees `active` equal to `c_scaleUpTo`, calls `Mirror`, which sees it below and responds by calling `ParallelMirror`, which sees it equal to, and so forth. This problem could be bad in theory, but should seldom occur with such extremity in practice.

Pipelines

We saw in Chapter 12, Parallel Containers, some abstractions that are useful when units of work form a producer/consumer relationship with one another. In these cases, one or more producers actively generate items of interest to one or more consumers. Sometimes there is a one-to-one relationship, but one-to-many, many-to-one, and many-to-many relationships are equally common. Usually the communication between such workers is encapsulated in a shared container such as the blocking and bounded collections we examined in the last chapter.

The simplest producer/consumer system is one in which there are a fixed number of producers and consumers, where producers are homogeneous and consumers are homogeneous. Often—but not always—these workers sit in loops, enqueueing and dequeuing, respectively. For example:

```
void Run(int producerCount, int consumerCount)
{
    Thread[] producers;
    Thread[] consumers;
    BlockingQueue<T> sharedQueue = new BlockingQueue<T>();

    producers = new Thread[producerCount];
    for (int i = 0; i < producerCount; i++)
```

```

    {
        producers[i] = new Thread(ProducerLoop);
        producers[i].Start(sharedQueue);
    }

    consumers = new Thread[consumerCount];
    for (int i = 0; i < consumerCount; i++)
    {
        consumers[i] = new Thread(ConsumerLoop);
        consumers[i].Start(sharedQueue);
    }

    for (int i = 0; i < producerCount; i++) producers[i].Join();
    for (int i = 0; i < consumerCount; i++) consumers[i].Join();
}

void ProducerLoop(object obj)
{
    BlockingQueue<T> queue = (BlockingQueue<T>)obj;
    while (true)
    {
        T data = /* ... generate data ... */;
        queue.Enqueue(data);
    }
}

void ConsumerLoop(object obj)
{
    BlockingQueue<T> queue = (BlockingQueue<T>)obj;
    while (true)
    {
        T data = queue.Dequeue();
        /* ... process data ... */
    }
}

```

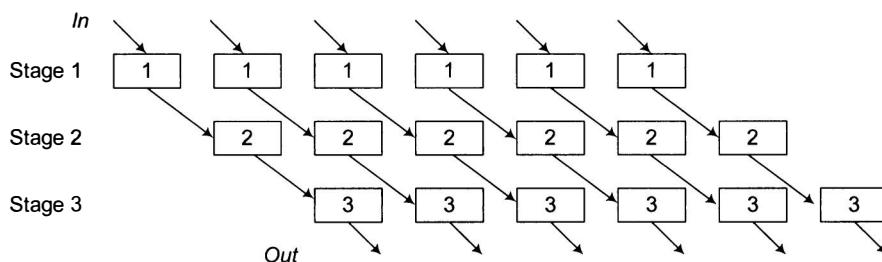
This is a vastly simplified example, but it's a good approximation of the structure. Usually we would have to handle shutdown. In this example, both `ProducerLoop` and `ConsumerLoop` go on forever (i.e., they use a `while(true)` loop); a more realistic design would be to use a shutdown flag set during shutdown that is polled periodically by both methods to determine when to quit. Often that would involve ensuring that the consumers have finished consuming all items of interest before quitting, whereas the producer may quit right away.

This is a very specific (and simplistic) example of a **pipeline**. Pipelines are akin to assembly lines in a production factory and arise in many settings.

A pipeline is generally comprised of one or more stages (usually at least two), and each stage is responsible for both consuming and producing some items of interest. In other words, each pair of adjacent stages forms a producer/consumer pair. In the simple example we just saw, the producers were one stage and the consumers were another. The “last” stage in a pipeline may or may not generate any data items of interest; in some cases, the “items” generated may simply be side effects that result from processing the data, such as displaying the results on a GUI.

Not only are there multiple stages in a pipeline, but, as with the previous example, there can be multiple threads of execution for any given stage. The number of threads dedicated to each stage need not be identical, and inequities are sometimes necessary to achieve load balance. When the number of threads differs from one stage to the next, the pipeline is said to be **nonlinear**. When they are identical for each stage, the pipeline is **linear**. This is illustrated in Figure 13.3.

Linear Pipeline



Nonlinear Pipeline

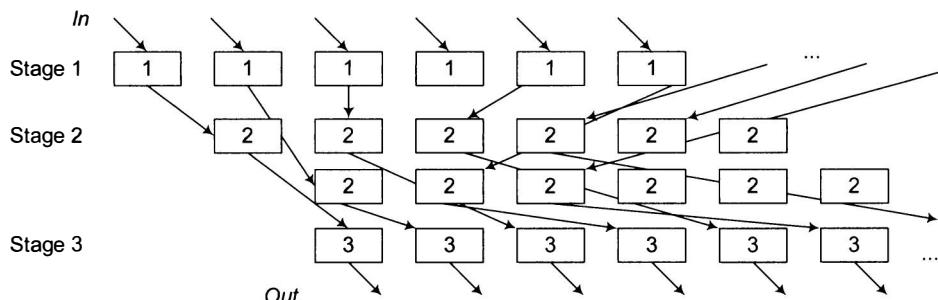


FIGURE 13.3: Illustration of linear and nonlinear pipelines

Pipeline stages are often configurable and pluggable. For instance, a pipeline that operates on `Car` objects can have stages added or removed depending on the operations being performed: that is, in one pipeline the stages might be dedicated to assembly (such as “install motor,” “add wheels,” “paint the car,” and so on), whereas in a completely different assembly they might not (e.g., “wash car,” “repair cracked fender,” and so forth). The `Car` itself needn’t know anything about the structure of this pipeline, stages needn’t know of each other, and in fact, the basic structure and logic of the pipeline itself doesn’t even need to know about the individual stages.

A Generalized Pipeline Type

Let’s look at a generalized `Pipeline<TSrc, TDest>` data structure. It allows you to build a pipeline comprised of an arbitrary number of stages, each of which has an arbitrary number of threads dedicated to it. `TSrc` represents the type of the source data fed into the start of the pipeline, and `TDest` is the final output for the whole pipeline. A pipeline is comprised of one or more `PipelineStage<TInput, TOutput>` objects, for which `TInput` represents the input type and `TOutput` represents the output for the stage in question. For each pipeline, the first stage’s input type will be the same as `TSrc`, and the last stage’s output type will be the same as `TDest`. Users of the `Pipeline<TSrc, TDest>` class never deal with individual stage objects—they are used for implementation only.

Before diving into the type’s implementation, here is a sample of its usage. Imagine we want to create a pipeline that represents the high-level process of turning copper ore into pure copper suitable for commercial use. There are three distinct phases in this process: the first phase takes the raw copper ore (represented with a `CopperOre` object) and crushes and grinds it into powder (`CopperPowder`); the second phase applies a pyrometallurgical process to turn the powder into pure unrefined copper (`UnrefinedCopper`); and the third and final stage roasts and smelts the unrefined copper to produce oxidized, pure copper (`PureCopper`) ready for consumption.

```
Pipeline<CopperOre, CopperPowder> p0 =  
    new Pipeline<CopperOre, CopperPowder>(  
        ore => CrushRawCopperOre(ore), 2  
    );
```

```

Pipeline<CopperOre,UnrefinedCopper> p1 =
    p0.AddStage<UnrefinedCopper>(
        powder => PerformCopperMetallurgy(powder), 2
    );
}

Pipeline<CopperOre,PureCopper> p2 =
    p1.AddStage<PureCopper>(
        unrefined => RefineCopper(unrefined), 2
);
;

CopperPowder CrushRawCopperOre(CopperOre ore) { ... }
UnrefinedCopper PerformCopperMetallurgy(CopperPowder powder) { ... }
PureCopper RefineCopper(UnrefinedCopper unrefined) { ... }

;

IEnumerable<CopperOre> minedOre = ...;
IEnumerator<PureCopper> refinedCopper = p2.GetEnumerator(minedOre);
while (output.MoveNext()) {
    PureCopper copper = output.Current;
    // ...
}

```

The allocation of `p0` sets up the initial stage. We are required to initially supply at least one stage for our pipeline. Then we use the `AddStage` method to produce successive stages in the pipeline; each call returns a new, modified pipeline object. Finally, we call `GetEnumerator` on `p2`, passing in a collection of `CopperOre` objects to transform into `PureCopper` objects. This kicks off the computation on several threads and returns a handle to the output being generated. All of the complicated coordination that occurs is hidden beneath a simple interface.

And with that, here's the definition of `Pipeline<TSrc,TDest>`. It depends on the `BlockingQueue<T>` type we defined in the previous chapter.

```

public class Pipeline<TSrc,TDest> : IPipeline
{
    private readonly IPipeline[] m_stages;

    public Pipeline(
        Func<TSrc,TDest> transform, int degree) :
            this(new IPipeline[0], transform, degree) { }

    internal Pipeline(

```

```

        IPipelineStage[] toCopy, Func<TSrc, TDest> transform, int degree)
    {
        // Copy current stages, and add a new one as the last.
        m_stages = new IPipelineStage[toCopy.Length + 1];
        Array.Copy(toCopy, m_stages, toCopy.Length);
        m_stages[m_stages.Length - 1] = new PipelineStage(transform, degree);
    }

    public Pipeline<TSrc, TNew> AddStage<TNew>(
        Func<TDest, TNew> transform, int degree)
    {
        return new Pipeline<TSrc, TNew>(m_stages, transform, degree);
    }

    public IEnumerator<TDest> GetEnumerator(IEnumerable<TSrc> e)
    {
        IEnumerable ef = e;
        CountdownEvent ev = null;

        for (int i = 0; i < m_stages.Length; i++)
            ef = m_stages[i].Start(ef, ref ev);

        foreach (TDest elem in ef)
            yield return elem;
    }
}

class PipelineStage<TInput, TOutput> : IPipelineStage
{
    private readonly Func<TInput, TOutput> m_transform;
    private readonly int m_degree;

    internal PipelineStage(Func<TInput, TOutput> transform, int degree)
    {
        m_transform = transform;
        m_degree = degree;
    }

    internal IEnumerable Start(IEnumerable src)
    {
        // Create a bunch of threads for this stage.
        Thread[] threads = new Thread[m_degree];
        BlockingQueue<TOutput> dest =
            new BlockingQueue<TOutput>();
        IEnumerator<TInput> sharedSrc =
            ((IEnumerable<TInput>)src).GetEnumerator();

        int active = threads.Length;
        for (int i = 0; i < threads.Length; i++)
    }
}

```

```

    {
        threads[i] = new Thread(delegate()
        {
            // Drain the source.
            TInput elem;
            while (sharedSrc.MoveNext(out elem))
                dest.Enqueue(m_transform(elem));

            // If we're the last one, mark the buffer as complete.
            if (Interlocked.Decrement(ref active) == 0)
                dest.IsDone = true;
        });
        threads[i].Start();
    }

    return dest;
}
}

interface IPipelineStage
{
    IEnumerable Start(IEnumerable src);
}

```

Despite it being fairly short, the implementation is subtle. So we'll spend a moment reviewing it. First notice the data structures involved: each pipeline object is comprised of an array of `IPipelineStage` objects that never change. Each of these is an instance of the `PipelineStage<TInput, TOutput>` type, which holds on to the `Func<TInput, TOutput>` transformation delegate and a degree that specifies how many threads to dedicate to the stage. The `IPipelineStage` interface just allows the implementation to invoke the `Start` method on a stage without having to know its type. The only purpose of `NewStage<TNew>` is to copy the current list of stages, tack a new stage to the end of type `PipelineStage<TDest, TNew>`, and return a pipeline object with a modified type signature of `Pipeline<TSrc, TNew>`. The old `TDest` is "lost" in the middle.

The interesting part happens when `GetEnumerator` is called on the pipeline. The data source is supplied in the `src` argument, which is typed as an `IEnumerable`. The method then starts each stage with calls to `Start` methods. For the first stage, we pass in the `src`; for each subsequent stage, we pass in the `BlockingQueue` returned from the previous stage, effectively gluing them together. After kicking off the stages, the `GetEnumerator` routine

enumerates the output from the last stage with a C# iterator via the `yield return` statement.

Most of the work happens inside of the `Start` routine on `PipelineStage<TInput, TOutput>`. It creates a set of threads whose size is equal to the `m_degree` value, passed in when the stage was constructed, and a `BlockingQueue<TOutput>` to hold elements generated by this stage. Each thread enumerates its `IEnumerator<TInput>` input until it is empty; each element is transformed with the stage's `m_transform` delegate, the result of which gets placed into the output collection. Recall from the last chapter that a blocking collection must be marked as being "done" to wake up blocked consumers when threads have stopped producing. To ensure this happens only when all threads in a stage is done, we keep a counter: each thread in a stage decrements the counter when finished, and the last one through signals to its output collection that it is done producing. This propagates through the stages.

A Good Pipeline Is a Balanced Pipeline

You might wonder why we'd want to change the number of threads dedicated to a particular pipeline stage. The reason is that any stage is apt to take more or less time to consume and produce elements than any other stage. This can lead to load imbalance that can result in inefficiencies in the pipeline. A balanced pipeline is a well performing pipeline.

What kind of inefficiencies does load imbalance lead to? Most pipelines use blocking queue style data structures such that when one stage is ready to consume the output of a previous stage and that previous stage hasn't yet made the next item available, the consumer will block waiting for it. Similarly, in many systems, these queues will be bounded to avoid any one stage getting too far ahead of any others. When load imbalance is high, the rate of blocking will be high, leading to stalls in the pipeline, increased latencies, and decreased throughput. Stalls can have a ripple effect on the pipeline: as one stage stalls, all subsequent ones will tend to stall as well. This has a damaging effect because all pipelines have a warm up time, which is the time before a pipeline is fully "primed." Because each stage has production latency, all subsequent stages must wait for all predecessor stages to produce elements too. For a 10-stage pipeline in which each stage

takes 100 milliseconds to produce a single item, the warm up time will be about a second; this is the latency incurred to produce one full item from the pipeline. Once primed, however, new elements will be produced every 100 milliseconds.

Now let's look at an example of load imbalance. Imagine a 3-stage pipeline. Say that, the first stage takes, on average, 100 milliseconds to produce an item; the second stage takes, on average, 500 milliseconds to consume and produce an item; and, the final stage takes, on average, 50 milliseconds to consume and produce an item. On a 16-core machine, a naïve implementation might assign 5 threads to each stage. But this would perform very poorly: the first stage would complete in one-fifth the time of the second stage, and its 5 processors would then idle; and the third stage would spend most of its time blocked, waiting for the slow second stage to produce elements. To see why this is true, imagine a pipeline with one thread dedicated to each of these stages. The first element takes 100 milliseconds to produce; until then, the second stage waits; it then consumes the element and produces one of its own, in 500 milliseconds elapsed time; in that amount of time the first stage has produced 5 more elements for it to work on; and the last stage had to wait 500 milliseconds to access something and will finish with it in a mere 50 milliseconds before having to wait 450 more for another.

There are many solutions to this problem, ranging from static allocation of threads to dynamic load balancing, much like the loop iteration division conundrum described earlier. For illustration's sake, let's explore a static allocation that would help. Say that, instead of 5 threads per stage, we vary the number per stage: the first stage gets 2 threads; the second stage gets 10 threads; and the last stage gets 1 thread. (Yes, this fails to add up to 16—which is one of the drawbacks to static allocation—but let's continue.) Now the pipeline is fairly balanced. The first stage produces 2 new items every 100 milliseconds, for a production rate of 1 element/50 milliseconds; the second stage runs with 10 threads every 500 milliseconds which, on average, for a consumption and production rate of 1 element/50 milliseconds; and the last stage runs with a single thread with its ordinary consumption rate of 1 element/50 milliseconds. Some degree of randomness and/or work variation can disrupt this.

Search

Many parallel algorithms take the form of search algorithms. I'm not talking about the kind of search that you use to find content on the Internet, but rather the more general idea of search in terms of data structures, as is commonly used in AI programming. Here are some examples of search problems for which parallelism might apply.

- Matching documents from a sample set containing certain related terms. Or, matching documents with common structural characteristics as determined through natural language processing style analysis. Many parallel workers might work at the problem until a global search condition is established, such as the presence of a certain number of paired documents.
- Similar to searching documents to find a particular pattern, we may search a list of images in order to perform facial recognition. All images can be processed in parallel, but as soon as a match is found all workers should quit.
- Solving an NP-hard problem with some kind of exhaustive search or heuristics based technique. For example, many puzzles require such solving techniques (Sudoku, n-Queens, etc.). In this case, usually all parties will search entirely different parts of the search space; the first to find a solution terminates the computation and reports success.
- Simulating or finding optimal solutions to a game using game tree searches, such as an alpha–beta search (see Further Reading, Knuth, Moore). Alpha–beta searches use a technique called alpha–beta pruning, which allows the search space to be trimmed as new information is found, leading to less wasted work. This is amenable to parallelism (see Further Reading, Russell, Norvig). Since many parallel workers can search different parts of the game tree at a time, they can also communicate to each other when potential cuts can be made. This leads to finding the set of solutions more quickly and increases the possibility of a more optimal solution, because more of the tree can be searched in less time.

All of these examples share common characteristics, specifically that many threads do work in parallel to locate a matching solution. When a solution is found, this is communicated to other workers (e.g., by setting a shared flag polled by all), and they halt the search right away. By throwing more workers at the problem, we hope to find the solution more quickly. Two terms can be used to summarize this: **cooperative** and **speculative**. These algorithms are cooperative because all threads share information as needed to help each other; and they are speculative because threads search more of the space, possibly doing wasted work, often leading to more CPU cycles spent on the problem but less wall-clock time. Other kinds of speculation are possible outside of the search space, such as the kind used by processors during branch prediction.

Search algorithms also routinely enjoy something called **super-linear speedups**. We describe speedups in more detail in the next chapter, but it's a pretty self-descriptive term: the parallel speedup may grow **superlinearly** as more processors are added. The reason is due to the speculative nature, that is, more of the search space is covered in less time, increasing the probability of finding a solution more quickly in a nonlinear fashion. With that said, some problems may see no benefit from throwing parallelism at it, or even see **sublinear speedups**. Much of the performance analysis we will encounter in the next chapter doesn't apply in the same way to cooperative search algorithms.

Message-Based Parallelism

Out of the three categories, we will spend the least amount of time discussing message based parallelism. There are many books available on how to build coarse-grained message passing systems (e.g., using Windows Communication Foundation [WCF] and Workflow Foundation [WF]). But there is little in the way of fine-grained, intraprocess message passing in Windows and .NET today. The Microsoft Robotics SDK contains a technology called the Coordination and Concurrency Runtime (CCR), which provides a programming model and tooling that support of these patterns (see Further Reading, Richter). Windows Workflow (WF) enables sophisticated

orchestration capabilities for fine-grained intraprocess work, but is limited in that true concurrency is not used in the resulting programs (see Further Reading, Shukla, Schmidt). Message Passing Interface (MPI) is a common programming model used in distributed HPC situations. There is other fragmented support throughout the Windows platform for message based parallelism, such as the windows messaging subsystem COM RPC and .NET Remoting, but in the absence of one true way, we will avoid in-depth discussions of any of these.

In message based parallelism systems, concurrency is driven by sending and receiving messages. To the extreme, the only way to generate concurrency is by creating separate agents with enforced isolation, and the only way to perform synchronization is through messages. Specialized languages such as Erlang take this approach (see Further Reading, Armstrong). In addition to the basic capability to send and receive messages, these systems usually offer sophisticated pattern matching capabilities, much like those available in functional programming languages such as F#. This often includes an ability to filter messages based on a predicate, to form conjunctions and disjunctions in the wait clauses (e.g., wait for a message from [A and B] or C, and so forth), and to have multiple end points to handle success and failure messages differently. The CCR also supports similar capabilities through library calls.

Other programming models exhibit much of the same style of programming of message based parallelism but without the sophisticated capabilities. For example, GUI programming—as we'll discuss more in Chapter 16—is based on sending messages from worker threads to the GUI thread. The GUI thread has a top-level event loop where its sole purpose is to receive and dispatch messages via event handlers. This is a messaging system at its core.

Cross-Cutting Concerns

There have been a few topics mentioned throughout this chapter that cut across all the different kinds of parallelism discussed. This includes handling exceptions in a parallel computation and cancellation of asynchronous operations.

Concurrent Exceptions

Windows structured exception handling (SEH) was built for sequential programs. It is fundamentally based on thread stacks and uses them to store handler frames, search for handlers during a throw, and so on. As a result, there are many conceptual mismatches that need to be addressed when dealing with exceptions in a concurrent program. To see the effect this has, consider the `DoAll` method shown earlier. It runs a set of delegates in parallel, but we completely ignored the fact that any of the delegates may throw an exception when invoked. If one of them were to throw an exception with the `DoAll` code as written, the exception will occur on a completely separate thread from the one that called `DoAll`; in this case, that will be a thread pool thread. And this will crash the program.

This might be OK. For instance, if we required that each delegate passed to `DoAll` were responsible for catching and dealing with any exceptions, this could be a perfectly reasonable choice. But it requires extra discipline for users of our API, discipline that can be cumbersome and error prone (and feels very different from sequential programming). An alternative approach is to rethrow any such exceptions in the context of the caller of `DoAll`. But to enable this, there is extra work we must do. Several important topics arise, such as whether we must wait for all of the concurrent work to complete before propagating the exception, impacts of rethrowing to debuggability, and so forth. Even trickier, it might be the case that multiple exceptions are thrown (simultaneously), which begs the question, “How are multiple exceptions exposed to the programmer calling `DoAll`?”. We could excuse ourselves from the business of caring about exceptions altogether, but users of `DoAll` would have to build these facilities themselves. Doing it once and in a consistent way would seem to be a good idea.

Marshaling Exceptions Across Threads

There are clearly a series of choices to be made when it comes to representing exceptions in a concurrent program. The first dimension to be considered is whether to marshal exceptions across threads automatically. The act of marshaling means that the body of each parallel unit of work will be wrapped in a try/catch block that communicates thrown exceptions back to the calling thread. The communication mechanism and definition of

calling thread change from one programming model to the next, but the principles are the same. The answer here is almost always “Yes” because the alternative is to allow an exception go unhandled, which, as mentioned earlier and in Chapter 4, Threads, leads to process crashes. Some systems, such as OpenMP, explicitly state that exceptions are not allowed to cross thread boundaries, but most people find this restriction undesirable.

Mechanically, marshaling exceptions across threads is simple. Let’s look at an example of this technique by returning to a simplified variant of our `Future<T>` class.

```
class Future<T>
{
    private T m_result;
    private Exception m_exception;
    private ThinEvent m_event = new ThinEvent(false);

    public Future(Func<T> func)
    {
        ThreadPool.QueueUserWorkItem(delegate
        {
            try
            {
                m_value = func();
            }
            catch (Exception e)
            {
                m_exception = e;
            }
            m_event.Set();
        });
    }

    public T Value
    {
        get
        {
            if (!m_event.IsCompleted)
                m_event.Wait();
            if (m_exception != null)
                throw m_exception;
            return m_value;
        }
    }
}
```

The delegate queued to the thread pool invokes the user supplied `func` delegate inside a `try/catch` block. If an exception is caught, it is stored in the

future's `m_exception` field and the thread remains alive. No matter whether the `m_value` field is successfully set or an exception occurs, `m_event` will be signaled afterward. Any thread that subsequently accesses the `Value` property will check the `m_exception` field and, if non-null, it will be rethrown. Otherwise, the value is returned. This is similar to the technique used by all `IAsyncResult` implementations in the .NET Framework.

While it achieves our desired behavior and is straightforward to implement, this approach has a few negative impacts to debugging that might not be immediately obvious.

- Because we rethrow the specific exception on a different thread with the `throw` statement, the original stack trace is lost. It is not possible to use the version of `throw` that doesn't perturb stack traces. This makes locating the source of failure more difficult. One workaround for this is to wrap the originally thrown exception in a new `Exception` object by storing it in the `InnerException` property. In this case, at least the original stack trace is preserved.
- If the marshaled exception ultimately goes unhandled, it will appear to have originated from the point at which it was rethrown. Breaking into the debugger will not go to the original `throw` site, but rather the API that is doing the rethrow. In the above example, that means the exception appears to come from accessing `Value`, rather than whatever `func` call that triggered the exception. This masks the original source of failure. Turning on first chance exception notifications in your debugger of choice (such as Visual Studio) enables you to see when the original exception is thrown but can be cumbersome, particularly when many exceptions are thrown leading up to the one of interest.
- The thread local state associated with the original failure will be gone by the time the unhandled exception is seen. So even if you can uncover the original exception and stack trace, any thread local state that might help debug the cause for failure will be gone. First chance exceptions can help the debugging experience here.
- Because the exception is rethrown by a specific API, it's possible that the program will never call it and, hence, the failure will go

unnoticed. For instance, in the above example, the exception only gets communicated if the value of the future object is requested. Forgetting to join is sometimes accidental—and can be a real headache to track down—or it can be explicit—such as when a dire failure has been discovered on another thread, and blocking could lead to hangs. It could be attractive to use a finalizable object to track whether an exception was seen and to crash the finalizer thread if it wasn't.

Neither the platform nor tools such as Visual Studio 2008 offer great support for solving any of these issues. Future releases will undoubtedly tackle some of them. Despite the drawbacks, marshaling is usually the right approach for these kinds of parallel invocation abstractions.

Aggregating Multiple Exceptions

All of the above is fine for single exceptions, but what about our `DoAll` method, in which many exceptions could occur? A common initial approach—which appears to be acceptable at first glance (mostly due to its simplicity and avoidance of the core problems)—is to rethrow the “first” exception to occur and to ignore the rest. Any reasonable implementation would try to stop all work associated with a complex operation once the first exception arises, but this approach doesn't responsibly admit that many failures might occur. In fact, some frameworks take this approach, such as the JCilk system (see Further Reading, Danaher, Lee, Leiserson).

The Flaws with Throwing “Just the First.” Though attractive because it keeps a familiar programming model, there are problems with this approach. To illustrate one such flaw, imagine if `DoAll` took this approach and threw only the first exception to occur, and we wrote the following.

```
BigResourceHandle brh = null;
try
{
    DoAll(
        delegate
    {
        // Prefer to use an in-memory resource:
        using (MemoryFailPoint mfp =
            new MemoryFailPoint(1024 * 1024 * 256))
```

```
        {
            brh = InMemoryBrh(...);
        }
    },
    delegate
    {
        ... accidentally trigger a NullReferenceException ...
    }
);
}
catch (InsufficientMemoryException)
{
    // Use disk storage if insufficient memory...
    brh = DiskStorageBrh(...);
}

// Continue (whoops!) ...
```

In this example, there are two parallel work items. The first tries to initialize some “big resource” using in memory resources. It uses the .NET `MemoryFailPoint` type to trigger an `InsufficientMemoryException` if there is not enough RAM to hold the resource before trying to allocate it. If an exception occurs, the catch handler goes ahead and uses a network storage location instead. The second work item does something that is immaterial to the discussion—all that matters here is that it could accidentally trigger a `NullReferenceException` under some circumstances, due to a bug in the program. Once this happens, some data structure is corrupt.

The approach of throwing only the first exception in this particular example means that if the `InsufficientMemoryException` occurs “first,” the `NullReferenceException` would be lost. The program would then proceed, unknowingly hobbled, and might cause even worse damage, possibly leading to additional data corruption and/or additional exceptions (which, one hopes, will eventually be noticed).

Aggregating Multiple Exceptions into One. All of this is a long winded build up to the recommended solution: preserve all of the failures, aggregate them into some wrapper exception type that can hold them all, and require users of APIs such as `DoAll` to determine how to handle them. This happens to have a side benefit, which is that the stack traces of original exceptions remain intact because we don’t rethrow them; we store them in

some array or list on the aggregate exception type. An extension to DoAll to use this technique follows.

```
void DoAll(params Action[] actions)
{
    List<Exception> exceptions = null;
    CountdownEvent latch = new CountdownEvent(actions.Length);

    for (int i = 0; i < actions.Length; i++)
    {
        ThreadPool.QueueUserWorkItem(delegate(object idx)
        {
            try
            {
                actions[(int)idx]();
            }
            catch (Exception e)
            {
                lock (actions)
                {
                    if (exceptions == null)
                        exceptions = new List<Exception>();
                    exceptions.Add(e);
                }
            }
            latch.Signal();
        }, i);
    }

    latch.Wait();

    if (exceptions != null)
        throw new AggregateException(exceptions);
}
...

class AggregateException : Exception
{
    private List<Exception> m_innerExceptions;

    public AggregateException(IEnumerable<Exception> exceptions)
    {
        m_innerExceptions = new List<Exception>(exceptions);
    }

    public Exception[] InnerExceptions
    {
        get { return m_innerExceptions.ToArray(); }
    }
}
```

Notice that we chose to *always* aggregate exceptions. That is to say, even if a single exception happens, we still wrap it up inside an `AggregateException` object. The reason is a bit subtle. If code that uses the `DoAll` API wants to catch a particular kind of exception—like the `InsufficientMemoryException` shown earlier—it always needs to consider the aggregate exception case, since, even if we just rethrew the original exception when one occurred, it is always possible multiple exceptions might arise. And so, if we only threw the single exception when it occurred, it would require two catch clauses.

```
try
{
    DoAll(...);
}
catch (InsufficientMemoryException)
{
    /*... handle it ...*/
}
catch (AggregateException ae)
{
    foreach (Exception e in ae.InnerExceptions)
        if (e is InsufficientMemoryException)
            /*... handle it ...*/
}
```

This leads to massive code duplication. Moreover, many people would not realize the need for the code duplication, leading to code that works under some circumstances (such as when one exception happens) but not others (such as when many happen). This is a kind of race condition. Therefore, I have chosen to always aggregate in the above example, and recommend you always do the same in your own code.

Impacts to Sequential Programming Models. There are clear downsides to this approach too. In fact, they are rather large. The most obvious is the fundamental change to how exceptions are dealt with in your programs. You can catch individual exceptions and handle them as usual. But you must overcatch, look for the right exception type in the `InnerExceptions` property and somehow decide whether to handle or repropagate individual exceptions within. This feels unnatural.

Another more subtle impact is the change in method contracts. In languages such as Java, where checked exceptions are pervasive, this impact is

more obvious. In C++ and C#, however, it is less obvious. Imagine, for sake of discussion, that we have an existing `Baz` API in a V1 library that may throw `FooException` or `BarException`. Callers of `Baz` know that it can throw and have written code that wraps calls to it in try/catch blocks that deal with these particular exception types. Then in V2 we decide to parallelize `Baz`. If the two different exceptions are thrown from different parallel units of work inside of it, `Baz`'s contract with users has suddenly changed dramatically. Now `Baz` might throw an `AggregateException` containing one `FooException`, one `BarException`, or both. This is a breaking change and could cause compatibility issues. When we release the new and improved `Baz` implementation, existing code now may not correctly deal with exceptions.

This is unfortunate. One possible solution is to offer a new API, such as `ParallelBaz` or another overload of `Baz`. This issue is yet another factor that drives people towards the solution to throw only the “first” exception that occurs.

Opportunities for Collapsing Homogeneous Exceptions. Often—particularly in data parallel problems in which homogeneous operations are being performed in parallel—it’s possible to turn many failures into one, preserving the original sequential exception model. For instance, imagine we are doing a division operation on an aggregate data structure; further imagine that certain elements in the input could occasionally lead to a divide by 0 exception, that is, the BCL type `DivideByZeroException`. If there are many 0s in the input, it may be acceptable to collapse many exceptions into one. It is worth noting right away that this clearly isn’t always true; for instance, the individual exceptions might carry unique information, such as the ordinal index of the element that triggered the exception.

The criteria used to determine what is “homogenous” is usually very program dependent, especially since it deeply impacts the way exceptions are propagated and caught. And so, if you want to take this approach, you’ll need to build it yourself. Here are some examples of information that can be used to determine homogeneity: the type of exception; the individual fields of the exception objects; the `TargetSite` of the exception objects, which contains a reflection handle to the exact method that threw the exception; and so on.

To illustrate, pretend we wanted to collapse `DivideByZeroException` objects, as explained above. At a certain point, we will have aggregated all instances of the exceptions, and we can apply our criteria for eliminating duplicates.

```
Exception[] GetUnique(Exception[] exceptions)
{
    List<Exception> uniqueExceptions = new List<Exception>();

    for (int i = 0; i < exceptions.Length; i++)
    {
        Exception current = exceptions[i];
        if (current.GetType() == typeof(DivideByZeroException))
        {
            for (int j = 0; j < uniqueExceptions.Count; j++)
            {
                Exception compare = uniqueExceptions[j];
                if (compare.GetType() == typeof(DivideByZeroException) &&
                    compare.TargetSite == current.TargetSite)
                {
                    break;
                }
                else if (j == uniqueExceptions.Count - 1)
                {
                    uniqueExceptions.Add(current);
                }
            }
        }
    }

    return uniqueExceptions.ToArray();
}
```

This is a simplified example, since `DivideByZeroException` doesn't contain any unique fields of interest. But it at least illustrates the point. Instead of `DoAll` throwing an aggregate exception containing the raw exceptions above, it could instead throw the result of calling `GetUnique`; this would result in duplicate `DivideByZeroException`s being removed. It could even just throw that single exception.

Cancellation

The term **cancellation** is certainly a loaded one. It has come up in a few contexts already in this chapter and earlier (and will again later) in the chapters of this book. It is commonly used to describe the following scenarios.

- Cancellation initiated from the GUI. When a user has initiated a long running operation, they often wish to have the ability to cancel it (if it is taking too long, or they realize the results are no longer needed). We discuss in Chapter 16, Graphical User Interfaces, mechanisms for supporting cancellation (via the `BackgroundWorker` type), but all that usually does is initiate the kind of cancellation we are about to discuss. It is not cancellation in and of itself.
- Canceled search algorithms caused by one worker locating an answer that obviates the need for other workers to continue searching. The most common way of supporting this is to use a boolean flag: it is set to `true` when it is time to terminate, and remains `false` otherwise. Sometimes the cancellation is more sophisticated than just a boolean condition. For example, imagine that workers are searching an input for the first element that satisfies some complicated criteria; one worker finds that element 33 satisfies the criteria, but another worker is still examining elements 8 through 12. It may be necessary that the other worker continues scanning until it exceeds element 33, to guarantee the “first” element was truly found.
- Periodic polling inside a long running (but not search) parallel task. For example, some external agent (like the GUI) may inform the task that it no longer needs to produce an answer. In this case, like the search algorithm, the task may periodically check a boolean flag for cancellation.
- Canceled blocking calls, such as I/O and synchronization. Related to the above, it is sometimes necessary to interrupt a thread while it is blocked waiting. We described thread interruption in Chapter 5, Windows Kernel Synchronization, which interrupts blocking calls in managed code due to synchronization waits. But we also described the pitfalls with that technique (interruptions that are not cooperative and may impact code not prepared for the interruption). Additionally, we will review I/O cancellation techniques in Chapter 15, Input and Output, which can be used to interrupt I/O blocking calls.

Code must be carefully written to support all of these scenarios. Supporting a shared boolean flag is simple; reacting to it is a matter of checking

its value periodically. But usually some combination of a flag and blocking cancellation is required. Rather than relying on thread interruption, it's recommended that you build cancellation by hand for those waits that cooperate with cancellation in your program in order to ensure that unexpected cancellations don't cause corruption. Typically this is done by ensuring all waits are done with a `WaitHandle.WaitAny` call, passing in a special cancellation event alongside the real event.

```
private bool m_isCanceled = false;
private ManualResetEvent m_cancelEvent = new ManualResetEvent(false);
...
void Cancel()
{
    m_isCanceled = true;
    m_cancelEvent.Set();
}
...
void Work()
{
    while (!m_isCanceled)
    {
        /*... do some work ...*/

        if (m_isCanceled) break;

        ManualResetEvent mre = /*... some interesting event ...*/;

        /*... do some work ...*/

        if (WaitHandle.WaitAny(
            new WaitHandle[] { m_cancelEvent, mre }) == 0) break;

        /*... do more work ...*/
    }
}
```

Notice that when it comes time to wait on `mre`, some application specific event of interest, we also pass in `m_cancelEvent`. When the wait returns, we check to see if the thread was awakened because the cancellation event was signaled. If so, we treat it as if we witnessed `m_isCanceled` as `true` and break out of the loop, terminating the work. This ensures we are disciplined about the termination of the work and have an opportunity to ensure application data is not left in an invalid state.

Where Are We?

We focused primarily on data and task parallelism in this chapter, the two most common kinds of parallelism you are apt to encounter in real-world programs. We saw some useful patterns, such as parallel for loops, reductions, sorts, fork/join, and divide and conquer. Once these concepts are known, applying them to particular problems becomes far simpler. Message based parallelism is quite common too, but due to the lack of a single standard programming model, we did not spend too much time reviewing the common patterns.

In the next chapter, we'll focus on the motivation for most of this discussion: performance and scalability. In it, concepts like parallel speedups and efficiencies will be reviewed, which are useful success metrics for most of the ideas presented in this chapter.

FURTHER READING

- J. Armstrong. *Programming Erlang: Software for a Concurrent World*. (Pragmatic Bookshelf, 2007).
- H. G. Baker, C. Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages* (1977).
- G. E. Blelloch, P. Gibbons, and Y. Matias. Provably Efficient Scheduling for Languages with Fine-Grained Parallelism. *Journal of the ACM*, 46(2) (1999).
- J. S. Danaher, I. A. Lee, C. E. Leiserson. Programming with Exceptions in JCilk. *Science of Computer Programming Special Issue on Synchronization and Concurrency in Object-oriented Languages*, Vol. 63, Issue 2 (2006).
- J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI)*(2004).
- D. E. Knuth, R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6 (4) (1975).
- T. G. Mattson, B. A. Sanders, B. L. Massingill. *Patterns for Parallel Programming* (Addison-Wesley, 2005).

- D. Hillis, G. Steele. Data Parallel Algorithms. *Communications of the ACM*, Vol. 29, Issue 12 (1986).
- T. Kodaka, K. Kimura, H. Kasahara. *Multigrain Parallel Processing for JPEG Encoding on a Single Chip Multiprocessor* (IWIA, 2002).
- L. Lamport. The Coordinate Method for the Parallel Execution of DO Loops. In *Proceedings of the 1973 Sagamore Conference on Parallel Processing* (1973).
- L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17, 2 (1974).
- H. Lieberman. Thinking about Lots of Things at Once without Getting Confused: Parallelism in Act 1, MIT AI Memo 626 (1981).
- B. Liskov, L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)* (1988).
- J. Richter. Concurrent Affairs: Concurrency and Coordination Runtime, *MSDN Magazine* (2006).
- S. J. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach* (Pearson Education, Inc., 2003).
- D. Shukla, B. Schmidt. *Essential Windows Workflow Foundation* (Addison-Wesley, 2006).

14

Performance and Scalability

CONCURRENCY IS OFTEN used in performance sensitive situations. In fact, a growingly popular reason people turn to concurrency is to better utilize parallel hardware due to the increasing mass market availability of multicore and SMP computers. But concurrency hasn't always had a place in the PC market. Historically, concurrent programming has dominated server-side scenarios, where scalability and utilization are very important. This includes Web and more exotic high performance computing (HPC) applications. The kind of performance consciousness needed to do fine-grained client-side concurrency is similar to that which is needed for server-side scaling—much more than the traditional style of performance tuning, which tends to focus much more on algorithmic complexity and cycles.

This chapter will examine the differences and highlight some of the key areas of focus and metrics when doing parallelism. It's impossible to overstate how incredibly important **sequential performance** remains. Slapping a parallel for loop around a poorly implemented algorithm is a terrible way of doing things and just wastes more of the machine's resources. You should always ensure you've chosen an appropriate sequential algorithm, tuned it, and then move on to parallelization. One caveat is that sequential optimizations often require breaking abstraction boundaries and increasing coupling and, thus, increasing complexity, all of which can make parallelism more difficult to retrofit.

A basic understanding of parallel hardware architecture is crucial to getting good parallel scaling because it often requires exploiting certain characteristics of the underlying hardware. It's an unfortunate fact that parallel programming demands a deep familiarity with hardware architecture, much like sequential systems software such as compilers and operating systems. This is not too onerous. The popular architectures that Intel and AMD currently provide are still straightforward and consistent. Memory systems haven't changed too much in the shift from symmetric multiprocessors (SMPs) to chip multiprocessors (CMPs), although research systems and intuition suggest that more fundamental changes will be needed in the not too distant future.

Parallel Hardware Architecture

Let's begin by reviewing some fundamental aspects of parallel hardware architecture, specifically those that impact parallel performance the most. Windows programmers have life a lot simpler than supercomputer programmers. That's because the number of disparate architectures to program is very small, and the number of processors to exploit is still small enough that the memory hierarchy hasn't changed too dramatically. Many lessons learned from cache conscious sequential programming directly apply. The descriptions found below are somewhat basic and only intended to paint a high-level picture of parallel computer architecture and how it can impact the performance of your programs. (For a more thorough overview of parallel hardware architecture, please refer to *Further Reading*, Culler, Singh, Gustafson.)

SMP, CMP, and HT

Three variants of multiprocessors are readily available for the computer architectures on which Windows currently runs: **symmetric multiprocessing** (SMP), **chip multiprocessing** (CMP), and **hyperthreading** (HT). The differences between these lie in the packaging of the processors, how they communicate with one another, and which resources are shared between them.

A single **processor package** (or **die**) is what occupies a socket on the motherboard. For very basic single processor machines, this package holds a single processor. The simplest way to extend this to a multiprocessor architecture is by adding more sockets to the motherboard and placing completely independent processor packages into them. This is SMP, and is the oldest form of parallel hardware that Windows has supported since NT. The processors typically share a single bus to a single main memory, and there is some level of caching that is usually shared among them.

As die sizes shrink (thanks to Moore's Law), and as power consumption and static leakage have become limiting factors, it has become more attractive to place additional processors on the same package as an alternative way of providing improved performance. This is CMP, is usually called **multicore**, and is becoming increasingly more common than SMP for client-side machines.

The third kind, HT, is currently only used by some Intel processors and is very similar to CMP. The primary (and quite substantial) difference is that the individual logical processors sharing the same package also share execution units instead of being entirely independent.

It's reasonable for any particular computer to use a any combination of these three, or even all three of them together. For example, imagine we have 4 packages (SMP), each with 4 cores (CMP), and each with 2 logical processors (HT). The result is 32 schedulable processors, and by creating that many threads Windows will freely and uniformly schedule threads onto each.

When looking at what a single processor needs to run, the basics include interrupt controllers, volatile state (i.e., registers), a connection to the memory system (ordinarily via a shared bus), and a processor core (i.e., something to actually execute instructions). In both SMP and CMP, each processor has its own independent set of each of these things. In HT, however, the processor core itself is shared among more than one logical processor. This may seem worthless, but HT can actually be used to hide memory access latencies. When one logical processor on a physical package stalls waiting on a memory operation (such as a fetch from main memory), other logical processors on that package can use the execution

unit in the meantime to perform useful work. Unlike SMP and CMP, scheduling many CPU-bound threads that do not frequently access memory at a HT logical processor will probably do more damage than good; that is, you're apt to see a slowdown rather than a speedup as a result, because units are shared.

Superscalar Execution

Aside from clock speed increases, a source of sizeable hardware performance improvements over the past decade has been **superscalar execution**. The purpose of superscalar execution is to take an existing sequential stream of instructions—such that programs needn't be rewritten—and exploit the natural parallelism lurking within. Processors that employ these techniques are often referred to as **out-of-order processors**, in contrast to **in-order**, because instructions are executed in a different order than laid out in the compiled program.

The kind of parallelism that results is called **instruction-level parallelism (ILP)**. You might be wondering where this natural parallelism comes from, given that the program is still sequential. But there are a few ways in which this can be accomplished.

- Processors can use multiple functional units simultaneously. At the bare minimum, a single arithmetic logic unit (ALU) can be doing integer math while a separate floating point unit (FPU) performs floating point math. A separate SSE unit can be doing vector operations simultaneously. And, depending on the level of inherent parallelism in sequential programs, multiple ALUs and FPUs can be used so that adjacent operations of the same kind (such as a stream of integer arithmetic) can be running at once.
- Memory move operations are extremely common, and yet memory access times are far greater than a single clock cycle. By **pipelining** many adjacent operations in a program—that is, having many of them executing at once—these latencies can be hidden by having operations complete out of order.
- To cope with the inability to read ahead of branches—in other words, not knowing which instructions to run ahead of time—many

modern superscalar processors also use **branch prediction**. This permits the processor to pre-execute instructions that would have been needed if a certain branch was taken, in anticipation that it will be taken; if the prediction is wrong, this leads to a **mispredicted branch**, and the results executed ahead of time are thrown away.

There are still inherent limitations to the degree of parallelism that can be realized with these techniques. Clearly a processor must respect the basic rules of data dependence that were discussed in Chapter 10, Memory Models and Lock Freedom. Moreover, it must respect some basic memory model rules—such as not reordering stores—so that systems and lock free programmers can reason about the concurrency behavior of their code.

In addition to these limitations, superscalar processors are more complex. This complexity manifests in three ways. First, they are more expensive to build. Second, they use more power than a corresponding in-order processor. This has been a contributing factor to the power wall that has stopped the continued clock speed improvements. This also means that out-of-order processors are sometimes inappropriate for use in low-power devices, such as in the embedded and mobile space. Finally, superscalar processors devote more of the die space to extra ALUs, FPUs, pipelining capabilities, and so forth. This reduces the number of possible cores and size of cache that can be added on the die and also contributes to power consumption.

The Memory Hierarchy

The primary differentiating factor in the performance of parallel programs, believe it or not, typically isn't the specific processor itself. It's the **memory hierarchy**. SMP and CMP have very different performance characteristics mostly because the memory systems are very different: the distance between processors and memory, the cache layout, and so on, vary greatly. The number of caches, their size, and which processors share which caches plays a huge role in determining the number of cycles that memory operations will consume, the level of contention in the memory system that can be introduced due to parallelism, and so on.

Nonuniform Memory Access

The first major decision a computer architect makes about a memory system is whether to make a **uniform memory access** (UMA) or **nonuniform memory access** (NUMA) machine. The distinction is that a UMA machine shares a single memory controller among all processors, whereas a NUMA machine has multiple. In a NUMA machine all processors are organized into **nodes**, each of which has its own physical memory. Each node typically contains a few processors. All processors can freely access any virtual memory address, but some addresses will be mapped to nodes that are far away; in other words, not in that processor's closest node's memory banks. The cost of such communication is vastly more expensive than accessing close memory. Additionally, cache coherence costs more on NUMA machines, so atomic interlocked operations are also more expensive. NUMA only applies to SMP architectures and is more commonly found on server-side machines.

Windows has intrinsic NUMA support in a few different areas. The OS will attempt to satisfy memory allocations via `VirtualAlloc` on the closest physical node, for example. And the OS thread scheduler will attempt to keep each thread on its home node when its ideal processor is not available. Managed programs should almost always use the server GC for NUMA machines because it has processor private heaps. This ensures that relocations keep memory on the correct node while the workstation GC may slide pages across nodes.

Cache Layouts

The next major decision is how to lay out the caches. Because the cost of accessing main memory is so costly and can saturate the bus (which can easily become a bottleneck when more and more processors are added to the system), it is attractive for computer architects to add several levels of caching. Registers are the most extreme form of caching; it's just that compilers are responsible for managing their contents instead of the hardware. The standard naming for such levels are L1, L2, L3, and so on; the smaller the number, the closer it is to the processor core, the smaller the size, and the faster it tends to be. L1 cache typically occupies on-die space, so that the processor can access it very quickly; but this means the capacity is quite limited.

On-die cache typically consists of two separate caches: an **I-cache** and a **D-cache**, responsible for caching program instructions issued to the processor and data, respectively.

SMP machines are often laid out such that each processor gets a reasonably sized L1 cache, and an L2 cache is shared among all the different processors. CMP machines are slightly different. Because multiple processors share the same die space, it can be attractive to give each (or some portion of them) independent L1 caches. It can also be attractive to share even more die space for an L2 cache shared among them all and to have an off-die L3. This is where you will see the most creative freedom applied by processor architects, both today and in the future.

Another design decision for cache design is the **cache-line size**. This is the smallest unit of memory that can be transferred to and from main memory. On most Intel machines lines are 64 bytes in size, while most AMD machines use cache lines that are 128 bytes in size. Line sizes can even change from one level in the cache to the next; for example, some Intel machines in the past used 128 bytes for L2 cache and only 64 bytes for L1 caches.

An example of a cache hierarchy is shown in Figure 14.1. In this illustration, a hypothetical 4-processor SMP system is depicted in which each processor has its own local L1 cache (1MB each) and a single level of L2 shared cache (16 MB), caching data which comes from the shared main memory (1GB). This is a fairly typical layout for modern SMP machines.

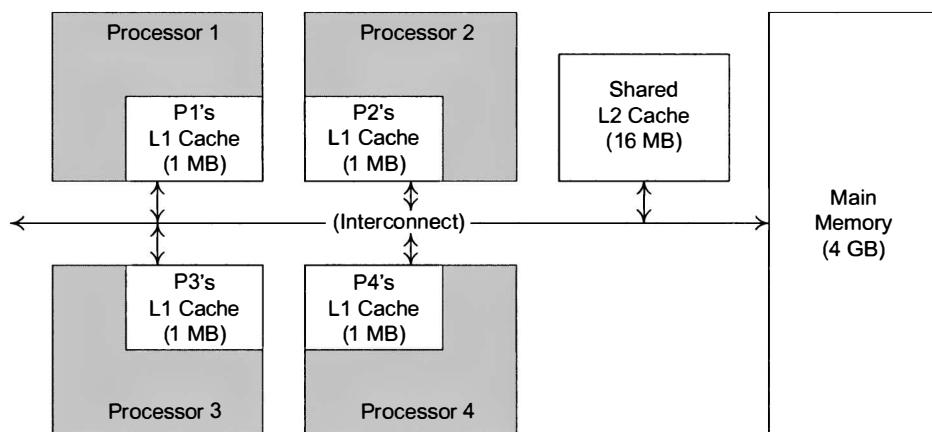


FIGURE 14.1: An example 4-processor SMP memory hierarchy

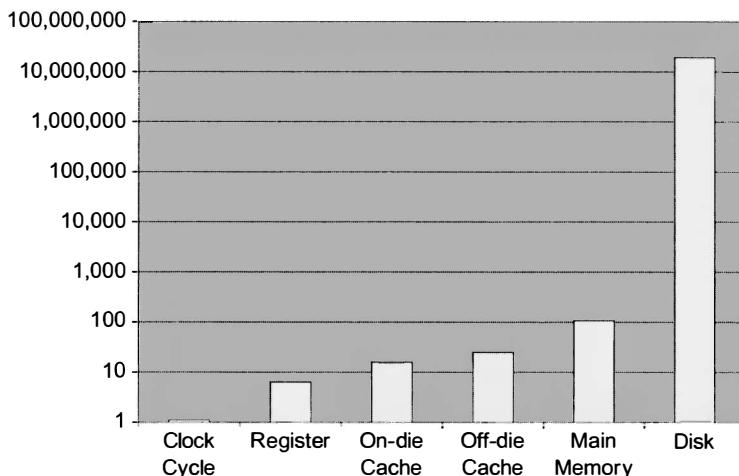


FIGURE 14.2: Logarithmic graph of memory and disk latencies

So the primary differences between different levels of caches are their size and access times. Figure 14.2 contains a chart that illustrates some rule of thumb measurements of memory access times, in terms of clock cycle time.

An interesting measure of performance is **cycles per instruction** (CPI). This is a measure of the average number of cycles each instruction executed by a program (or some subset of the program) consumed. This can be used to explain the cache behavior and its impact to performance, specifically whether trips to main memory were frequent. A higher CPI means that more time was wasted waiting for memory operations to complete.

Cache coherence is the act of keeping caches synchronized with what is in main memory. We already saw in Chapter 10, Memory Models and Lock Freedom, that caches, ILP, and write buffering—techniques all used to hide memory access latencies—can cause some real headaches. But you have to appreciate the amount of complexity that goes into making it all work. Most modern AMD and Intel processors use a directory based snooping structure, which is a fancy way to say that each processor is responsible for watching cache transactions that are going to main memory. As cache transactions are witnessed, the processor must update any of its own cache lines, tracking their status, and possibly **invalidating** local copies so that they are subsequently refetched from main memory when needed.

Most processors use a **MESI protocol** to track cache line state. Each line is given a status.

- **M is for Modified.** The local processor has pending updates on the line (e.g., in the write buffer), and the value in main memory is considered stale.
- **E is for Exclusive.** The local processor has exclusive access to the line. This is used for interlocked operations such as XCHG. Only one processor may have a given line marked as E in its local cache.
- **S is for Shared.** The cache line is valid and may be shared for read access by multiple processors at a given time.
- **I is for Invalid.** Due to snooping a write back to main memory performed by a separate processor, this line is no longer valid. It must be refetched.

Contention arises for all modes but S. When processors write to the same cache line a large amount of cache maintenance and memory traffic is generated. This is expensive, so it is ideal to try and avoid concurrent access by multiple processors to the same memory locations. That is particularly true of E mode. This is a topic we'll explore in depth momentarily.

Caches are fixed in size, so another event that would cause lines to be evicted is a cache becoming full. Most caches use a least recently used (LRU) policy to determine which lines to evict first in such cases. Subsequent access of evicted lines will be satisfied elsewhere in the hierarchy.

You can query about the layout of the memory hierarchy—to obtain information such as what processors share what levels of cache, whether hyperthreading is enabled, NUMA node layout, and so forth—using the `GetLogicalProcessorInformation` function. This API was added to Windows Server 2003 and beats out `GetSystemInfo` and querying the CPUID to determine similar information.

```
BOOL WINAPI GetLogicalProcessorInformation(
    PSYSTEM_LOGICAL_PROCESSOR_INFORMATION Buffer,
    PDWORD ReturnLength
);
```

The function stores a bunch of interesting data in the array of SYSTEM_LOGICAL_PROCESSOR_INFORMATION records supplied. The number of records is system dependant, so calling the API with a NULL Buffer, and ReturnLength of 0 allows you to determine what the correct buffer size is beforehand. The API will return FALSE and GetLastError will be ERROR_INSUFFICIENT_BUFFER, but the ReturnLength parameter will have received the correct size in bytes. You must then allocate a buffer of at least ReturnLength/sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION) elements. After calling the method again with the correct arguments, the array will be populated.

Each record contains a lot of useful information.

```
typedef struct _SYSTEM_LOGICAL_PROCESSOR_INFORMATION
{
    ULONG_PTR ProcessorMask;
    LOGICAL_PROCESSOR_RELATIONSHIP Relationship;
    union
    {
        struct {
            BYTE Flags;
        } ProcessorCore;
        struct {
            DWORD NodeNumber;
        } NumaNode;
        CACHE_DESCRIPTOR Cache;
        ULONGLONG Reserved[2];
    };
} SYSTEM_LOGICAL_PROCESSOR_INFORMATION,
*PSYSTEM_LOGICAL_PROCESSOR_INFORMATION;

typedef enum _LOGICAL_PROCESSOR_RELATIONSHIP
{
    RelationProcessorCore,
    RelationNumaNode,
    RelationCache,
    RelationProcessorPackage
} LOGICAL_PROCESSOR_RELATIONSHIP;

typedef struct _CACHE_DESCRIPTOR
{
    BYTE Level;
    BYTE Associativity;
    WORD LineSize;
    DWORD Size;
```

```
PROCESSOR_CACHE_TYPE Type;  
} CACHE_DESCRIPTOR,  
*PCACHE_DESCRIPTOR;  
  
typedef enum _PROCESSOR_CACHE_TYPE  
{  
    CacheUnified,  
    CacheInstruction,  
    CacheData,  
    CacheTrace  
} PROCESSOR_CACHE_TYPE;
```

Each `SYSTEM_LOGICAL_PROCESSOR_INFORMATION` record applies to one or more processors on the machine, specified by the `ProcessorMask` field, and represents one of four things, indicated by its `Relationship` field:

- `RelationProcessorCore`: This specifies that one or more logical processors share the same physical core. If the `ProcessorCore`'s `Flags` field is 1, the processors share the execution units, that is, they are hyperthreaded.
- `RelationNumaNode`: The processors indicated share a NUMA node. The node number is indicated by the `NumaNode`'s `NodeNumber` field. For non-NUMA machines, there will always be a single node that all processors share.
- `RelationCache`: The entry captures a description of a cache that one or more processors share access to. The corresponding `CACHE_DESCRIPTOR` contains all sorts of useful information. The `Level` field indicates whether the cache is L1, L2, or L3 with values 1, 2, or 3, respectively. The associativity is available, with a value of `0xFF` meaning the cache is fully associative, and both the cache line size and the total size (both in bytes) are also available. Lastly, the type of cache is specified by the `Type` field.
- Finally, `RelationProcessorPackage` specifies that one or more processors share the same physical package or socket.

Here is a sample program, written in C#, that queries all of this information and pretty prints it to the screen.

```
using System;
using System.Runtime.InteropServices;

class Program
{
    public static unsafe void Main()
    {
        if (IntPtr.Size != 8)
        {
            Console.WriteLine("Only works on 64-bit.");
            return;
        }

        int entrySize = 0;

        // Make a call to get the necessary size info. Success assumed.
        GetLogicalProcessorInformation(null, ref entrySize);

        int entryCount = entrySize /
            sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION);

        SYSTEM_LOGICAL_PROCESSOR_INFORMATION * pEntries =
            stackalloc SYSTEM_LOGICAL_PROCESSOR_INFORMATION[entryCount];

        if (!GetLogicalProcessorInformation(pEntries, ref entrySize))
        {
            Console.WriteLine("GLPI call failed: {0}",
                Marshal.GetLastWin32Error());
            return;
        }

        string[] relationshipStrings = new string[] {
            "Processor Cores",
            "NUMA Nodes",
            "Caches",
            "Sockets"
        };

        for (int i = 0;
            i < Enum.GetValues(
                typeof(LOGICAL_PROCESSOR_RELATIONSHIP)).Length;
            i++)
        {
            Console.WriteLine("{0}", relationshipStrings[i]);
            for (int j = 0; j < relationshipStrings[i].Length; j++)
                Console.Write("=");
            Console.WriteLine();

            for (int j = 0; j < entryCount; j++)
            {
                SYSTEM_LOGICAL_PROCESSOR_INFORMATION entry=pEntries[j];

```

```
if ((int)entry.Relationship == i)
{
    ulong pmask = entry.ProcessorMask.ToInt64();
    ulong trymask = 1;
    for (int k = 0; k < Environment.ProcessorCount; k++)
    {
        if ((trymask & pmask) != 0)
            Console.Write("*");
        else
            Console.Write("-");
        trymask <<= 1;
    }

    Console.WriteLine("\t");

    switch (entry.Relationship)
    {
        case LOGICAL_PROCESSOR_RELATIONSHIP.
            RelationProcessorCore:
        if (entry.Flags == 1)
            Console.WriteLine("Hyperthreaded");
        break;
        case LOGICAL_PROCESSOR_RELATIONSHIP.
            RelationNumaNode:
        Console.WriteLine("#{0}", entry.NodeNumber);
        break;
        case LOGICAL_PROCESSOR_RELATIONSHIP.
            RelationCache:
        CACHE_DESCRIPTOR cache = entry.Cache;
        Console.WriteLine(
            "{0}, {1}k, Assoc {2}, LineSize {3}, {4}",
            cache.Level, cache.Size / 1024,
            cache.Associativity, cache.LineSize,
            cache.Type);
        break;
    }
}

Console.WriteLine();
}

Console.WriteLine();
}

[DllImport("kernel32.dll", SetLastError = true)]
private unsafe static extern bool GetLogicalProcessorInformation(
    SYSTEM_LOGICAL_PROCESSOR_INFORMATION * buffers,
```

```
        ref int returnLength
    );

[StructLayout(LayoutKind.Explicit)]
struct SYSTEM_LOGICAL_PROCESSOR_INFORMATION
{
    [FieldOffset(0)]
    internal UIntPtr ProcessorMask;
    // Note! Works on 64-bit only [assume UIntPtr==64bits].
    [FieldOffset(8)]
    internal LOGICAL_PROCESSOR_RELATIONSHIP Relationship;

    // These fields are unioned together.

    [FieldOffset(16)]
    internal uint Flags;

    [FieldOffset(16)]
    internal uint NodeNumber;

    [FieldOffset(16)]
    internal CACHE_DESCRIPTOR Cache;

    [FieldOffset(16)]
    internal ulong Reserved1;
    [FieldOffset(24)]
    internal ulong Reserved2;
}

enum LOGICAL_PROCESSOR_RELATIONSHIP : int
{
    RelationProcessorCore = 0,
    RelationNumaNode = 1,
    RelationCache = 2,
    RelationProcessorPackage = 3
}

[StructLayout(LayoutKind.Explicit)]
struct CACHE_DESCRIPTOR
{
    [FieldOffset(0)]
    internal PROCESSOR_CACHE_LEVEL Level;
    [FieldOffset(1)]
    internal PROCESSOR_CACHE_ASSOCIATIVITY Associativity;
    [FieldOffset(2)]
    internal ushort LineSize;
    [FieldOffset(4)]
    internal uint Size;
    [FieldOffset(8)]
```

```
internal PROCESSOR_CACHE_TYPE Type;
}

enum PROCESSOR_CACHE_LEVEL : byte
{
    L0,
    L1,
    L2,
    L3
}

enum PROCESSOR_CACHE_ASSOCIATIVITY : byte
{
    FullyAssociative = 0xff
}

enum PROCESSOR_CACHE_TYPE : int
{
    Unified = 0,
    Instruction = 1,
    Data = 2,
    Trace = 3
}
}
```

I've personally found this particular program very useful. (Note that, as written, it only works on 64-bit systems. The layout of `SYSTEM_LOGICAL_PROCESSOR_INFORMATION` changes to be 4 bytes smaller; handling that properly would have lead to an increase in code size, hence it has been omitted.) There is typically plenty of information readily available with Task Manager, various other Windows tools, `systeminfo.exe`, and so on, but getting detailed information about the cache layout of a machine is particularly difficult. System manuals seldom even go into this kind of detail, except to describe at a high level cache sizes and capacities. And yet cache layout affects the performance of parallel programs tremendously.

Here is some sample output on a commodity dual-core, dual processor machine.

```
Processor Cores
=====
*--*
-*-
--*-
---*
```

```

NUMA Nodes
=====
**** #0

Caches
=====
*--- L1, 32k, Associativity 8, LineSize 64, Data
*--- L1, 32k, Associativity 8, LineSize 64, Instruction
-*-- L1, 32k, Associativity 8, LineSize 64, Data
-*-- L1, 32k, Associativity 8, LineSize 64, Instruction
**-- L2, 4096k, Associativity 16, LineSize 64, Unified
--*- L1, 32k, Associativity 8, LineSize 64, Data
--*- L1, 32k, Associativity 8, LineSize 64, Instruction
---* L1, 32k, Associativity 8, LineSize 64, Data
---* L1, 32k, Associativity 8, LineSize 64, Instruction
--** L2, 4096k, Associativity 16, LineSize 64, Unified

Sockets
=====
**_
--**

```

We can see in this particular computer that each processor has its own 32KB L1 cache (both I-cache and D-cache) and that each socket has a shared 4MB L2 cache. There is no cache common to all processors.

On the Importance of Locality

As discussed, cache coherence adds cost. Not only do the additional memory transactions cost something, but the need for a processor to invalidate and refetch a cache line will add considerable overhead to any program. Therefore, thoughtful memory access behavior is important, and modern caches are designed to reward memory conscious programming. This kind of memory friendly behavior is called **locality**.

Spatial and Temporal Locality. There are two basic kinds of locality.

- **Spatial locality.** Memory that is physically close together should be used together. For example, if an operation must access multiple memory locations, prefer to access those that will reside on the same cache line close together in the operation. Typically this kind of locality is inherent in many programs. If your program accesses one

field of an object, the chances are very good that your program will need to access another field of that same object. Larger cache lines prefetch data that is likely to be needed soon afterward.

- **Temporal locality.** Memory that must be used multiple times should be done as close (in time) as possible. By doing so, the chance that the cache line on which the location resides will still be in the closest cache when subsequent operations are reached is greater.

Both are important. Not programming in a locality conscious way will lead to an increase in CPI, which will slow your program down and increase memory bus traffic. This can easily cause the memory system to become the bottleneck on parallel machines; ideally, the CPU would be the bottleneck, such that adding more processors will allow inherent scalability to use them freely. Programming in a locality conscious way is more of a heuristics based art than a well defined and verifiable methodology but is important to always keep in mind when designing data structures and algorithms for parallel programs.

The Cost of Sharing. Let's see specifically why locality is important and what the effects of not paying attention to it can be.

When more than one processor shares access to a location in memory that resides on the same cache line, coherence traffic will increase and can negatively impact performance. This is especially bad when the processors are performing writes, because it requires invalidation of lines in local processor caches. This is particularly true of atomic (interlocked) operations because they must acquire cache lines in exclusive (E) mode. Contention like this can even lead to an exclusive bus lock on older memory architectures.

What's worse, **false sharing** often leads to the symptoms of sharing, but is not always evident in the program. This happens when two different memory locations are spatially collocated in memory, but logically distinct in the program. For heap memory this is often a byproduct of how memory gets allocated. In .NET, the server GC has processor local heaps and so allocations on separate processors should be physically separate enough to avoid this issue. Similarly, many native memory allocators have processor local pools of free pages; this is primarily to avoid contention, but also helps

avoid false sharing too. Unfortunately, it's very easy to get into a situation where allocations happen together.

Another common situation in which false sharing crops up is when commonly read fields are close in memory to commonly written fields, usually on the same object. A popular technique to reduce working set overhead is called **hot/cold splitting**, which results in commonly used fields being collocated in memory together. This is exactly the wrong thing to do, however, for parallel programs. You want the commonly written fields as far away from the commonly read fields as possible. This is important to keep in mind when designing new data structures.

Here is an example program that shows that a small mistake can make a large difference.

```
using System;
using System.Threading;

class Program
{
    class Counter
    {
        internal int m_count;
    }

    public static void Main()
    {
        int p = Environment.ProcessorCount;
        Console.WriteLine("P={0}", p);

        long withSharing = Run(p, 1000, true);
        Console.WriteLine("Sharing    = {0}", withSharing);

        long woutSharing = Run(p, 1000, false);
        Console.WriteLine("NoSharing = {0}", woutSharing);

        Console.WriteLine("%           = {0}",
                         woutSharing/(float)withSharing);
    }

    private static long Run(int p, int runTimeMs, bool falseSharing)
    {
        GC.Collect();

        Counter[] counters = new Counter[p];
```

```
if (falseSharing)
    for (int i = 0; i < counters.Length; i++)
        counters[i] = new Counter();

bool stop = false;
using (ManualResetEvent mre = new ManualResetEvent(false))
{
    Thread[] tt = new Thread[p];
    for (int i = 0; i < p; i++)
    {
        int idx = i;
        tt[i] = new Thread(delegate()
        {
            Counter c;
            if (falseSharing)
                c = counters[idx];
            else
                c = counters[idx] = new Counter();

            mre.WaitOne();

            while (!stop)
                for (int j = 0; j < 100; j++)
                    c.m_count++;
        });
        tt[i].Start();
    }

    mre.Set();
    Thread.Sleep(runTimeMs);

    // Notify threads to stop and then wait.
    stop = true;
    foreach (Thread t in tt)
        t.Join();
}

// Compute the total counts.
long total = 0;
for (int i = 0; i < p; i++)
    total += counters[i].m_count;
return total;
}
```

All this program does is spawn one thread per processor. Each thread continuously increments its own private counter object until told to stop by the

main thread. There is no synchronization or locking that would contribute to any sort of slowdown. We run this same test two ways, with a slight variation. The first time, we pass `true` for the `falseSharing` argument to `Run`. This causes it to allocate the counter objects on the primary thread. Each thread will just index into a shared array to fetch its own private counter object; remember they are operating on entirely different objects. But doing so ensures the objects are allocated close together in memory. When `falseSharing` is `false`, on the other hand, each thread allocates its own counter object immediately when it starts to run. Due to thread local GC allocation contexts, this helps to ensure objects are allocated further apart from one another in memory. At the end, we count how many increments the threads were able to perform in the given amount of time; higher numbers are better (i.e., it maps to throughput).

The exact numbers you will witness are likely to be very nondeterministic because they depend on memory layout and timing. But when run on a modern 64-bit, dual-core, dual-CPU Intel machine (that's 4 cores in total), I see anywhere from a 30 to 45 percent increase in the number of increments when false sharing is eliminated. On larger machines, the effects will be worse because of the increased cost of cache coherence. On an experimental 24-processor machine, the test can perform 180 to 200 percent more increments when there is no false sharing. In the worst case, false sharing more than halved the amount of increments that could be performed!

A Brief Word on Profiling in Visual Studio

Visual Studio has had an integrated performance profiling tool since Visual Studio 2005. In Visual Studio 2008, this can be accessed through the Analyze menu. Under this menu, there are several options, including a Profiler submenu with a link to New Performance Session. By creating a new session, adding your project or binary as a target, and kicking off a performance profile, you will be presented with a summary of where the time went during execution. The default mode is to periodically sample the instruction pointer (IP) as threads execute, tally up the statistics, and then count up the total number of samples spent in each function.

This is very useful for sequential and parallel programs alike. There are several things, however, that aren't captured that are very important for

parallel performance. An example is which threads were waiting at what points and why. You can play some tricks here. For example, by changing all your locks to spin locks, all waiting will begin to show up as CPU time and, thus, will show up in your profiling session.

You may also use this same profiler to examine memory behavior. You can get to the Properties window for your session by right clicking on it. (Note: You must right click on the session itself and not a particular target.) In the Sampling area, you can change the sampling interval to smooth out statistical inconsistencies that arise due to the sparse default interval. But even better, you can change the Sample Event from Clock Cycles to something else, including various superscalar execution and memory related events.

Here are some examples of useful hardware performance counters that you can sample.

- **Instructions Retired.** This tracks the number of instructions that actually completed and can be used to compute CPI. Dividing the number of instructions retired by the number of cycles the processor is capable of executing over that period of time tells you the CPI, although things like waiting, thread scheduling, interrupts, and the like makes this more difficult to compute in practice. You can do two individual runs—one for instructions retired and the other with the usual cycle sampling—and then do some spreadsheet magic to aggregate like functions together and compute an approximation of CPI. Nonetheless, measuring the total number of instructions retired in the false sharing example above shows that there is a direct correlation between retirement counts and cache behavior.
- **L2 Misses.** This provides a count of L2 cache misses, so you can track down where your program is spending most of its time as a result of them. These are good places to focus your time on improving locality behavior. Note that many processors won't actually support this specific option, but that most of them offer other specific counters to see things like L2 Lines In, L2 Lines Out, and so forth, which provide a more detailed view of cache traffic. Sampling the false sharing program shown above indicates a 59-fold increase in

L2 cache misses when compared to the more cache friendly variant shown alongside.

- **Mispredicted Branches.** This tells you how many branches were predicted incorrectly, possibly impacting the performance improvements a program sees as a result of superscalar execution. It's really difficult to analyze this data for tangible improvements you can make to your code, but it is interesting nonetheless.

There are plenty of other counters that you'll find, including ones to do with misaligned memory references, floating point operations per second, memory reordering, SIMD SSE execution, and much more. These can be useful to track down specific kinds of performance problems.

Speedup: Parallel vs. Sequential Code

When it comes to using concurrency for performance, a.k.a. *parallelism*, your success will be measured in terms of **speedups** and **efficiencies**. These are two direct measures of how well a parallel algorithm fares against its sequential counterpart. We'll spend a fair bit of time reviewing how to measure such things, and what kinds of program characteristics will impact them the most. But first, how do you know when to even begin looking at parallelism?

Deciding to “Go Parallel”

Consider a simple for loop:

```
for (int i = 0; i < N; i++)
    body(i);
```

Imagine we want to answer the simple question: Should this be a parallel **for** loop? (The question, we will find, is actually not so simple after all.) This question might be asked because we profiled our application and found that this single loop is where the program spends the bulk of its time. It turns out there are many factors to consider in deciding whether to “go parallel.”

- Is there enough work being done by all iterations of the loop to warrant parallelism? Presumably we’re asking the question because

we believe that the answer will be yes, at least for some values of N and `body`. But it could be that there is only “enough work” in some cases, such as when N exceeds a threshold or some condition causes `body` to exceed a certain cost (in CPU cycle count). And determining exactly what “enough work” means is difficult because we must consider the unique overheads introduced by parallelism (allocations, thread switches, synchronization objects, and synchronization waits).

- In what context is this `for` loop run? If a massively parallel computation calls this `for` loop at the leaves of its callstacks when there are expected to be many outstanding such calls, it may not be wise to introduce additional parallelism at this level in the application. This is called **nested parallelism** and some (but not all) schedulers account for it. The Windows and CLR thread pools, for example, do not efficiently handle nested parallelism. It may be better to exploit parallelism at a coarser-granularity by using something like an agents model.
- What does `body` do? If `body` executes entirely within a global lock, it would be foolish to parallelize this loop. The result would lead to nearly zero parallelism, but the addition of the unique parallelism costs noted above. Accessing any locks, even if only for short periods of time, will decrease the efficiency of parallelism. The same is true of any kind of shared resource, including the file system. The addition of parallelism may also introduce extra memory contention that would have otherwise not been a problem; in fact, a cache aware loop may go out of its way to ensure better locality—and yet this can lead to problems with parallel loops depending on how iterations are scheduled.
- Even if `body` doesn’t currently acquire locks, will it need to if parallelism were to be introduced? We’d need to ensure that it is thread safe. But if this code was originally authored as a sequential `for` loop, the callgraph may be making assumption about being able to freely access shared state.

In summary, we are trying to answer the question: Will we see a speedup by making this a parallel `for` loop? The term speedup is an

important one and will be the dominant focus of this section. As software developers considering adding parallelism to otherwise sequential programs, we need to be able to reason intuitively about speedup as a first level of analysis. Often this requires building up some kind of model of the expected performance and thread interactions. But after doing this initial analysis and modeling, it's incredibly important to measure the expected performance characteristics with the observed ones. Many of the factors above—such as synchronization and memory effects—are too subtle to reason about alone.

Measuring Improvements Due to Parallelism

Knowing what to look for when measuring is challenging, particularly when determining whether an algorithm is scaling as well as it could be, what its upper limit might be, and so on. That's where things like speedup and efficiency become useful concepts.

Sublinear, Linear, and Superlinear Speedups

The application of parallelism to some sequential code can have four basic outcomes. We will use the word *speedup* to describe these outcomes. To calculate speedup, we first measure the execution time of the sequential version of the algorithm, calling it $T(1)$, then the execution time of this same algorithm parallelized on P processors, calling it $T(P)$, and last divide one by the other: $\text{Speedup} = T(1)/T(P)$. Given this, the four basic outcomes are:

1. Speedup < 1 indicates a slowdown, or the absence of a speedup.
2. Speedup $< P$ indicates a sublinear speedup.
3. Speedup of $\sim P$ indicates a linear speedup.
4. Speedup $> P$ indicates a superlinear speedup.

A slowdown is bad. It is often an indication that some code may be better off run sequentially rather than in parallel. This is not always true. It could be a result of an improperly parallelized algorithm, cache unfriendliness, synchronization bottlenecks, implementation mistakes, and so forth. The algorithm itself may be theoretically capable of attaining some kind of appreciable speedup. And some algorithms may see speedups on a certain

number of processors, but slowdown at some point: for example, a parallel algorithm may not break even with a sequential algorithm until 4 processors have been applied and will scale well beyond this. This could be due to constant overheads introduced by parallelism that dwarf the advantages with small degrees of parallelism. The same is true of using too many processors. It could be that a parallel algorithm exhibits too much interthread communication and / or memory contention that end up dominating execution time when higher numbers of processors are used.

Most properly written parallel algorithms exhibit sublinear speedup. The lack of perfect linear speedups is often due to the added costs of parallelism and natural scaling inhibitors such as interthread communication. For example, the parallel merge sort we examined in the previous chapter had a portion that was only moderately parallel and required communication—the merge—which will prevent us from seeing a perfect linear speedup. Moreover, a linear speedup of exactly P (without rounding) is highly unlikely; more often than not, the speedup will fall on one side or the other. And, more often than not, the speedup will fall on the sublinear side.

At first, superlinear speedups may appear to be impossible. How is it possible that, by applying P processors, some bit of code can execute more than P times as fast?

There are two basic ways in which this can happen (see further Reading, Sutter).

- Do more work in less time.
- Use more resources that could only be utilized by doing so in parallel.

The first way, do more work in less time, seems like an obvious way to make any code go faster. But parallelism can help in a unique way because multiple threads may be sharing information with one another. This is normally exploited in search style algorithms.

To illustrate, imagine we are searching an array for a single element that has some particular criteria. Perhaps evaluating an element against these criteria involves running a fairly complicated algorithm, such as some alpha-beta pruning game search. As we go, we may decide to skip certain elements because they are similar (or identical) to other elements found to

have been disqualified. Each thread takes its own chunk of the input array to work on in parallel; for simplicities sake, we'll say there are N elements in the array, P threads, and each thread takes a contiguous chunk of N/P elements to work on by itself.

Here is the key insight: by sharing the disqualifications, some threads may do less work than they would have done sequentially because of the way the list has been traversed. If thread P finds that elements with certain properties are disqualified, it lets threads $0 \dots P-1$ know about that and they can skip any similar occurrences that they run across. Less input needs to be examined than if we had simply walked the list sequentially.

The second way, use more resources which could only be utilized by doing so in parallel, applies to many kinds of resources. The basic point is that instead of using one resource first, processing the results, moving on to the next, and so on, it is sometimes possible to use more resources at once. This is similar to the way that multiple ALUs can be used in superscalar execution. One kind of resource that immediately comes to mind is processor caches. Because each processor has some private cache, a parallel algorithm can use more cache at once (across the machine) than the sequential version could. This can lead to superlinear speedup.

Efficiency: Natural Scalability versus Speedups

Placing speedups into the four buckets is useful for theoretical analysis but is not always sufficient. There is a big difference between achieving a speedup of 2 on a 32-processor machine and a speedup of 30, and yet both are lumped together into the single sublinear category. Additionally, both values are absolute and depend greatly on the specific value of P , while we are often more interested in the **natural scalability** of an algorithm.

The parallel efficiency of an algorithm can be calculated by dividing the speedup by the number of processors: Efficiency = Speedup/ P . With this new metric, we can rephrase the definitions of our sublinear, linear, and superlinear categories.

1. Efficiency < 1 indicates a sublinear speedup.
2. Efficiency of exactly 1 indicates a linear speedup.
3. Efficiency > 1 indicates a superlinear speedup.

We now have a way to plot an algorithm's performance regardless of particular processor count. That's not to say an algorithm's efficiency will be the same for all possible values of P. It will undoubtedly exhibit different efficiency numbers on machines with different processor counts. Many parallel algorithms will differ in performance greatly depending on machine specific architectural artifacts too, such as the memory hierarchy. This fact aside, the efficiency metric is a useful way of normalizing the data so that you can more accurately compare how your algorithm scales as the number of processors and machine architecture does change.

As an example, if we measure efficiency numbers of 0.75 on a 2-processor machine, 0.55 on a 4-processor machine, 0.35 on an 8-processor machine, and 0.2 on a 16-processor machine, the drop off in scaling may be significant cause for concern. As the number of processors increases, the algorithm in question does not scale. This problem is much easier to identify with efficiency numbers than with the speedups—which are 1.5, 2.2, 2.8, and 3.2, respectively—because it is tempting to settle for any kind of sublinear speedup when sublinear is expected. The speedup numbers can be misleading. They are, after all, increasing as the number of processors increase. A drop off in efficiency can be due to the reality of speedups—such as Amdahl's Law, which we are about to examine—but can represent a flawed algorithm too.

Measuring Speedup and Efficiency

It's trivial to measure speedups and efficiency. In C++ you can use the `QueryPerformanceCounter` function and in .NET you can use `System.Diagnostics.Stopwatch`. For example, here is a simple C# harness that wraps some sequential and parallel variants of the same algorithm.

```
using System;
using System.Diagnostics;

public abstract class SpeedupTest
{
    public void Run(int times, int p)
    {
        Stopwatch seqSw = Stopwatch.StartNew();
        for (int i = 0; i < times; i++)
            RunSequential();
        seqSw.Stop();
```

```
Stopwatch parSw = Stopwatch.StartNew();
for (int i = 0; i < times; i++)
    RunParallel(p);
parSw.Stop();

Console.WriteLine("Sequential Time: {0}ms",
    seqSw.ElapsedMilliseconds);
Console.WriteLine("Parallel Time : {0}ms",
    seqSw.ElapsedMilliseconds);

float speedup = seqSw.ElapsedTicks / (float)parSw.ElapsedTicks;

Console.WriteLine("Speedup      : {0}X", speedup);
Console.WriteLine("Efficiency : {0}%", speedup / p);
}

protected abstract void RunSequential();
protected abstract void RunParallel(int p);
}
```

An implementation of `SpeedupTest` overrides `RunSequential` and `RunParallel`. A test framework then invokes `Run` with a number of times to execute the test (the `times` parameter) and the degree of parallelism (the `p` parameter). Running the test multiple times during the measurement is a good way to normalize deviations in the statistical output. More clever statistical techniques can be used, such as eliminating outliers, examining standard deviation to pinpoint nondeterminism in tests, and the like, but this example is a useful and simple starting point.

Amdahl's Law

An often cited problem with parallel speedups is called **Amdahl's Law** (see Further Reading, Amdahl). This law states something that will seem obvious once you understand it. The ability of a parallel algorithm to exhibit speedup over its sequential counterpart is inherently limited by the remaining sequential parts after parallelization. At some point, even if the parallel parts scale perfectly, the sequential parts still remain and still take just as long to execute as they did before. Taking a more holistic view, an entire program's performance increase due to parallelism will inherently be limited by its sequential portions.

This is unavoidable. Even an algorithm that is embarrassingly parallel—that is, it will scale linearly—will have some amount of overheads associated with forking and joining work.

More formally, if S is the percentage of execution time that remains sequential (i.e., $1 - S$ is the percentage that has been parallelized), and P is the degree of parallelism, then the maximum theoretical speedup you can expect to see is

$$\frac{1}{S + \frac{(1 - S)}{P}}$$

As the value of P grows, this expression approaches a limit of $1/S$. Thus, if you've only managed to parallelize 85 percent of your algorithm, S is 15 percent, and your code will be at best capable of achieving a speedup of $1/.15$, or approximately 6.66. This is illustrated by Figure 14.3. In effect, no matter how small the P portions become, the S portions will still remain and do not become any smaller than in the original sequential program.

In theory, based on these calculations, throwing any more processors than seven at this particular problem would be worthless. In practice, however, this law tends to oversimplify a lot. For example, the positive effect that using more cache provides could mean that additional processors will actually yield gains. The reverse is also true: the added contention on

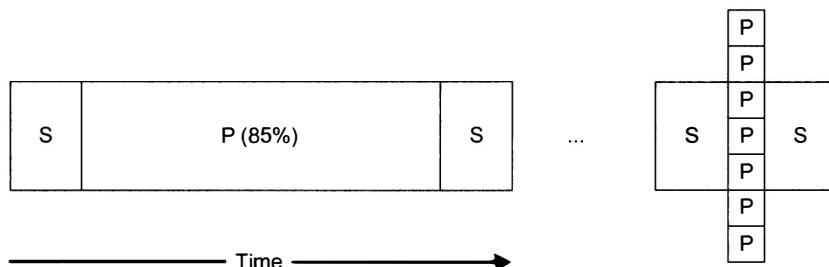


FIGURE 14.3: Effect of Amdahl's Law

shared resources, whether that is memory or synchronization objects, could mean that even using seven processors will be wasteful and degrade performance.

And **Gustafson's Law** (see Further Reading, Gustafson)—which is really the same as Amdahl's Law with a more positive spin—is worth keeping in mind. Gustafson pointed out that once parallelism has been added to the most compute-intensive parts of a problem, the problem size is apt to grow to consume more execution time proportional to the less interesting sequential parts of the program. While this doesn't do away with the fundamental problem Amdahl points out, it tends to be true. If you parallelize the right parts of your program, scalability will only improve over time as the problem size expands due to application requirements, increase in business data size, and so forth.

Critical Paths and Load Imbalance

In addition to the speedup of your parallel algorithm being limited by any sequential portions, it is also limited by the length of the longest parallel part of that algorithm. In effect, when there is load imbalance, the tail end of parallel computations can become serial, or less than perfectly parallel. Every parallel algorithm has a **critical path**, which is the longest path that must be traversed before the computation is complete. To achieve the scalability you desire, it is imperative that you spend time focused on reducing the length of this critical path.

To illustrate the effect of a critical path, imagine we are on a 4-processor machine and we break apart our computation into 4 distinct pieces. Each runs independently of the other, with no shared resources, and the serial portions are reduced to the overhead of fork and join. You would expect this embarrassingly parallel problem to scale linearly. But if the first of the 4 parallel chunks of work takes 20 percent longer than the others to complete, you have effectively serialized that last 20 percent of the work. If the execution time for a single processor is $T(1)$, then $T(4)$ will be $((1 - 0.2) * T(1))/4 + 0.2 * T(1)$. The result is that, instead of a linear 4 times speedup, you will find your speedup to be limited at 2.5 times. That's a large difference.

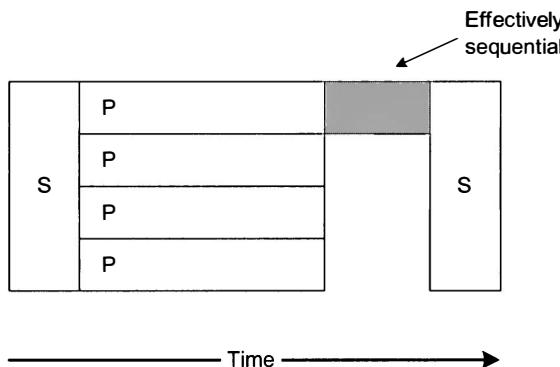


FIGURE 14.4: Critical paths and load imbalance

This effect can be illustrated by Figure 14.4.

This is a simple case. More often than not, the parallel portions of a problem will complete at entirely different times. The critical path is important, but a common source of this overall issue is **load imbalance**. With a statically partitioned parallel for loop, for instance, we may find that some iterations complete much faster than others. As an extreme example, consider:

```
ParallelFor(0, N, delegate(int i)
{
    for (int j = 0; j < i; j++)
        Work();
});
```

In this case, loop iterations take an amount of time proportional to the iteration number. (Each iteration will run one more invocation of `Work` than the previous one.) Statically dividing this up into equal sized and contiguous iteration chunks would be terrible for parallel performance. Every processor would take substantially longer than the one that was assigned a chunk before it. We may see some kind of speedup, but it's not going to be very impressive. Dynamic partitioning and load balancing are necessary in such cases.

In addition to or instead of inherent load imbalance, threads can be delayed for any number of reasons. For instance, should a thread experience an unusually high number of cache misses, or page faults due to physical memory pressure, or get context switched out because another process is eligible to run, it may be delayed so that it becomes part of the critical path. Contention on locks and other shared resources, exact timing of GCs, and I/O

latency can all contribute to this effect. The result can be nondeterministic in nature and difficult to track. The effect could be that an algorithm sometimes performs quite well, exhibiting impressive speedups, but some proportion of the time appears to perform abysmally.

Garbage Collection and Scalability

The CLR provides three garbage collection (GC) engines, each with varying degrees of concurrency utilization. Any parallel program will, at some point, find itself running into GC interference because of the pause times and automatic introduction of sequential steps. If we're running a perfectly parallel algorithm, for instance, and suddenly a GC gets triggered on a single processor, it will freeze our algorithm for some period of time, effectively making it sequential for some amount of time. The three flavors of GC are:

- **Workstation.** This is the default GC used on single processor machines. It uses a single thread to perform collections.
- **Workstation (concurrent).** This is the default GC used on multi-processor machines. This mode uses a single thread for most activities, such as generation 0 collections, physically relocating memory, and so forth, but also employs a separate thread running concurrently with the application to do some amount of concurrent scanning of generation 2 collections ahead of time. This reduces pause time when it comes to finally performing the collection, because a large portion of the heap has already been scanned. Additionally, the workstation GC uses processor local allocation contexts to amortize the cost of allocating memory, reduce contention on heap locks, and to improve locality for memory allocated on separate processors.
- **Server.** The server GC must be chosen through configuration and is the best choice for highly parallel applications where throughput is important. It manages a private heap for each processor and has a dedicated thread affinitized to each CPU whose job is to perform collections for its own private heap. Like the concurrent workstation GC, per thread allocation contexts are used. All processors are involved in the collection process: each of them first partake in traversal and marking, synchronize with each other at a barrier, and

then are responsible for compacting their own private heaps. Although the whole application must be suspended, all of the machine's processors are utilized.

To turn on the server GC mode, you can use ordinary .NET configuration files.

```
<configuration>
  <runtime>
    <gcServer enabled="true" />
  </runtime>
</configuration>
```

You might be wondering why server GC isn't automatically used for multiprocessor machines. The reason is two-fold. First, the bulk of .NET programs are not highly parallel. For those kinds of programs, particularly interactive ones, concurrent workstation GC provides better performance. Second, using the server GC forces all processors on the machine to be used during collections. The fact that threads are affinitized makes this even worse. On systems with many programs running at once, this is generally not a good idea because it is intrusive. If many programs need to collect at once, the effect can be disastrous. This is the reason it is called the **server GC**; most of the time, servers have few very busy programs running (often just one) that effectively own the machine and where throughput is a primary focus in performance tuning (versus responsiveness and fairness).

Spin Waiting

Spin waiting can sometimes be advantageous to true blocking. This would initially seem to contradict advice given in Chapter 2, Synchronization and Time, where true blocking was sold as a more efficient way of waiting. Subsequent chapters have pointed out that many synchronization primitives—such as CLR monitors and Win32 critical sections—use a so-called **two-phase locking protocol**, where a period of brief spinning is used when a lock is unavailable before falling back to a true wait on a kernel object. Alternative but similar designs are possible. When in doubt, however, just stick to these existing primitives.

The reason that spinning can be appropriate is two-fold: context switches and kernel transitions are very expensive. On a multiprocessor machine, spinning can avoid both of them. Think about a common sequence of events that would occur if we were programming with a lock without built in spinning.

1. Thread T1 acquires lock L and begins running its critical region.
2. Thread T2 tries to acquire lock L; it's already held, so T2 blocks.
(This incurs a kernel transition and context switch.)
3. Thread T1 exits its critical region, releasing lock L. This signals T2.
(The signal itself also incurs a kernel transition, and possibly a switch depending on priority boosting and the current state of the system.)
4. Thread T2 awakens and again tries to acquire lock L.
(This also incurs a context switch, for T2 to awaken and become rescheduled.)

There are always two context switches in this example: one when T2 initially finds lock L to be held (step 2) and another when T1 releases L and signals T2 to wake up and acquire it (step 4). If T2 is preventing T1 from making forward progress at step 2—perhaps because this example is run on a single processor machine—then putting it to sleep so that T1 can run is the best thing we can do. But if T1 and T2 are running concurrently, and step 3 is very short, the two context switches add considerable overhead: anywhere from a few thousand to more than 10,000 cycles, in addition to the possibility of dirtying caches. Because of priority boosting, the thread releasing the region, T1, may get context switched out so that T2 can run in its place. This helps to mitigate convoys that might have otherwise occurred, but the threat of convoying due to all of these context switches remains very real.

Locks that spin briefly can avoid the context switches entirely. Instead of blocking at step 2, T2 will spin wait for L to become available. This also avoids the switch at step 4, because T2 is already running when it notices that L has become available. Because massive contention is typically uncommon, and because lock hold times are on average meant to be very short, spin waiting can be advantageous.

The implementation of a general purpose spin lock is a more difficult task than you might imagine, however. There are many trivia-like details to ensure spin waiting works properly on Windows and the kinds of processors on which Windows runs; these have to do with the thread scheduler, Intel HyperThreading (HT), and caches. In addition, most spin locks really should fall back to true waiting in worst case situations, such as when the cost of a context switch has already been exceeded at some implementation complexity. Even when the worst cases seem statistically improbable, they can occur if a thread is interrupted by a context switch while in a critical section or when the arrival rate at a lock becomes unusually high.

In this section, we'll look at two spin lock approaches. The first spins on a shared variable, and doesn't fall back to true waiting, although it does explicitly yield the thread's timeslice after some time. The second is a lock called a Mellor-Crummey-Scott (MCS) lock, which reduces contention on shared memory locations. It has been proven to exhibit higher degrees of scalability on large multiprocessor machines with nonuniform memory access.

(Both are shown in C# code. The transformation to C++ is typically much easier than the reverse because C# needs to deal with the possibility of asynchronous thread aborts. This fact can complicate matters, particularly when we look at MCS locks.)

How to Properly Spin on Windows

Before moving on to the lock specifics, there are some basic rules you should consider when using spin waits on Windows.

- Issue calls to `YieldProcessor` (in Win32) or `Thread.Yield` (in .NET) on each iteration of your spin wait loop. These emit YIELD or PAUSE instructions on relevant processors—which is only Intel's Hyper-Threading (HT) enabled processors—and NOPs on other processors where HT isn't present. (`Thread.Yield` in .NET takes a numeric argument and emits that number of these instructions in a loop.) This ensures the processor is made aware that the code currently running is performing spin waits and will make the execution unit available to other logical processors so they can make true forward progress.

- In most spin wait circumstances, shared state will be read during each iteration. This can lead to memory traffic and cache contention. Therefore, it is wise to introduce a growing delay—called **exponential backoff**—on each spin iteration. It also sometimes makes sense to introduce randomization to avoid multiple threads from executing in a lock step fashion, which would possibly lead to a severe case of livelock.
- When pure spin waiting is being used (versus two phase), it is sometimes worth issuing explicit context switches with one of the appropriate platform APIs. The reason is that if a thread has already consumed a full context switch of spinning, it may be more appropriate for it to allow others to make forward progress than continuing to use processing resources (possibly interfering with the very thread that is being waited for).
- When issuing explicit context switches, the Win32 function `SwitchToThread` is most appropriate to use. (The equivalent is not available in .NET unless you P/Invoke.) It relinquishes the calling thread's timeslice and runs another runnable thread in its place. This is in effect for a single timeslice. It returns `TRUE` to indicate that a switch occurred, and `FALSE` otherwise. As of Windows Vista and Server 2008, this function may not consider all threads on the system.
- Because `SwitchToThread` may not consider all threads on the system for execution, it is wise to occasionally call `Sleep` or `SleepEx` (in Win32) or `Thread.Sleep` (in .NET). Passing a value of `0` as the argument is best because it does not result in a context switch if there are no threads of equal priority ready run. However, passing a value of `1` occasionally is also wise: if you ever get into a situation where a higher priority thread is spin waiting on a lower priority thread, this can help avoid a nasty starvation problem that would require getting the balance set manager involved to fix.

Because of the tricky rules, we can create a reusable `SpinWait` data structure that encapsulates all of this logic. Replicating it repeatedly in a program's code base would create a maintenance problem. Determining the ratio of calls to `SwitchToThread`, `Sleep(0)`, and `Sleep(1)` is left as a performance

profiling exercise for the reader. Those chosen for illustration intuitively make sense, but different numbers will work better or worse for different workloads. You may even want to make them tunable by passing arguments to the constructor.

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

public struct SpinWait
{
    internal const int YIELD_THRESHOLD = 25; // When to do a true yield.
    internal const int SLEEP_0_EVERY_HOW_MANY_TIMES = 2;
    internal const int SLEEP_1_EVERY_HOW_MANY_TIMES = 10;
    internal const int MAX_SPIN_INTERVAL = 32; // Max spin iterations.

    private int m_count;
    private static int s_processorCount = Environment.ProcessorCount;

    public int Count
    {
        get { return m_count; }
    }

    public bool NextSpinWillYield
    {
        get { return s_processorCount==1 || m_count >= YIELD_THRESHOLD; }
    }

    public void SpinOnce()
    {
        if (NextSpinWillYield)
        {
            int yieldsSoFar =
                (m_count >= YIELD_THRESHOLD ?
                 m_count - YIELD_THRESHOLD :
                 m_count);
            if ((yieldsSoFar % SLEEP_1_EVERY_HOW_MANY_TIMES) ==
                (SLEEP_0_EVERY_HOW_MANY_TIMES - 1))
                Thread.Sleep(0);
            else if ((yieldsSoFar % SLEEP_1_EVERY_HOW_MANY_TIMES) ==
                     (SLEEP_1_EVERY_HOW_MANY_TIMES - 1))
                Thread.Sleep(1);
            else
                SwitchToThread();
        }
        else
    }
}
```

```

        Thread.SpinWait(
            (int)(m_count *
                ((float)MAX_SPIN_INTERVAL / YIELD_THRESHOLD)) + 1);

        m_count = (m_count == int.MaxValue ?
                    YIELD_THRESHOLD : m_count + 1);
    }

    public void Reset()
    {
        m_count = 0;
    }

    [DllImport("kernel32.dll")]
    internal static extern int SwitchToThread();
}

```

We cache the `Environment.ProcessorCount` value because it currently allocates garbage objects (due to a security demand it performs) and must P/Invoke to `SwitchToThread` because .NET doesn't expose any such method. There is also a `NextSpinWillYield` property. We can use this property in our spin lock primitives to determine when to fall back to blocking (e.g., on an event or condition variable), as in the following pseudo-code:

```

SpinWait sw = new SpinWait();
while (!... some condition ...)
{
    if (sw.NextSpinWillYield)
        ... block ...
    else
        sw.SpinOnce();
}

```

A Spin-Only Lock

Spin-only locks are only appropriate for extraordinarily tiny critical regions. This point can't be stated enough. A good rule of thumb is a critical region is made up of less than 10 instructions and is expected to take less than 50 cycles to execute. That rules out a lot of things, including memory allocation, dynamically dispatched calls (including virtual method calls), and any access of high latency resources such as the file system.

After the previous section, building a spin-only lock will be simple. We'll use a single flag that is 0 when the lock is available, and threads will use interlocked operations to compare and swap (CAS) a non-0 value when holding it. Threads will use their own IDs to claim ownership. This can help during debugging and also allows us to detect recursion to provide more friendly error messages. The most difficult part in building such a lock lies in tuning the spin logic based on intended workloads.

Here's a sample implementation of a SpinLock in C#.

```
using System;
using System.Runtime.ConstrainedExecution;
using System.Threading;

struct SpinLock
{
    private volatile int m_state;
    private const int LOCK_AVAILABLE = 0;

    public void Enter()
    {
        int tid = Thread.CurrentThread.ManagedThreadId;
        if (m_state == tid)
            throw new Exception("Recursion not allowed");

        Thread.BeginCriticalRegion();
        if (Interlocked.CompareExchange(
            ref m_state, tid, LOCK_AVAILABLE) != LOCK_AVAILABLE)
        {
            SpinWait sw = new SpinWait();
            do
            {
                Thread.EndCriticalRegion();

                // Spin until we see the lock available.
                do
                {
                    sw.SpinOnce();
                }
                while (m_state != 0);

                Thread.BeginCriticalRegion();
            }
            while (Interlocked.CompareExchange(
                ref m_state, tid, LOCK_AVAILABLE) != LOCK_AVAILABLE);
        }
    }
}
```

```
public void Exit()
{
    Exit(false);
}

public void Exit(bool flushCacheWithRelease)
{
    if (m_state != Thread.CurrentThread.ManagedThreadId)
        throw new Exception("Lock not owned by thread");

    if (flushCacheWithRelease)
        Interlocked.Exchange(ref m_state, LOCK_AVAILABLE);
    else
        m_state = LOCK_AVAILABLE;

    Thread.EndCriticalRegion();
}
```

Several factors are interesting.

- Our `SpinLock` type is a .NET value type (`struct`). This makes it a very lightweight 4-bytes type that can be allocated inline, within another heap-allocated object. This has one downside: if you box an instance and share it among threads, all unboxed instances will be separate and won't know of each other. This is a mistake that could lead to some surprising races if not caught.
- We have marked `m_state` as `volatile` to prevent compilers from hoisting reads outside of loops, which could lead to infinite spinning. This problem was encountered in Chapter 2, Synchronization and Time, where some examples of historically interesting critical region techniques were examined.
- We store the thread's ID into `m_state` to mark it as acquired. This allows us to detect recursion, cases when a thread that doesn't own the lock tries to erroneously release it, and aids debugging. That said, we could take alternative approaches. We could use a value of 1 to mean the lock is held and avoid the cost of accessing `Thread.CurrentThread.ManagedThreadId` (which incurs a TLS lookup). Additionally, we could have allowed recursion—though for a spin lock, this is highly suspect—by having a second field; when `Enter` is called, we increment and skip the interlocked operation if it's

already equal to the current thread's ID; when `Exit` is called, we decrement it and only switch `m_state` to 0 when the recursion counter also hits 0.

- `Thread.BeginCriticalSection` and `EndCriticalSection` are used to notify CLR hosts that we're in a region of code which, if interrupted, could lead to system instability. Since spin locks are used to protect important data and because an interrupt could lead to infinite spinning in some threads, this is a must for any critical code. We must ensure `BeginCriticalSection` has been called before a successful interlocked operation has marked the lock as being owned, and call `EndCriticalSection` when we know the current thread doesn't own the lock: either because of a failed interlocked operation or because the lock was released.
- When contention is detected, we only attempt the interlocked operation on the shared flag once we have subsequently read it as 0 (the innermost do-while loop). This reduces problematic contention caused by multiple processors acquiring a cache line in exclusive mode only to find that it doesn't contain the correct value. There is a race between seeing it as 0 and writing, but at least this ensures contention happens only when the lock was observed as being truly available. This is sometimes called a **test and test and set (TATAS) lock**.
- When releasing the lock, we have a choice. Do we use an interlocked operation for the write, or not? The lock will remain correct if we do not—and will undoubtedly perform better—but this can lead to starvation because the “release” write may never leave a processor's cache in time. For example:

```
SpinLock slock = ...;
void f()
{
    while (true)
    {
        slock.Enter();
        try
        {
            // Do some work.
        }
    }
}
```

```

        finally
        {
            slock.Exit();
        }
    }
}

```

If the thread loops around and tries to reacquire the lock very soon after it releases it, as in this example, it may be given immediate access. This is unfair to other threads that may have been waiting for the lock for a much longer time. In fact, it could lead to indefinite starvation if the thread never stops. This is why we offer an `Exit` method with a boolean parameter: when `true` we use an interlocked operation to release the lock.

- One feature whose omission may be surprising is timeouts. You could build this by occasionally querying a counter (using Win32's `QueryPerformanceCounter` or the .NET `Stopwatch`), but this is left as an exercise to the reader. Because spin lock critical regions are meant to be very small, you should seldom need a timeout capability anyway.

A couple items are unimportant to C++ (first and fourth), but the others apply equally.

Another optional feature that would apply to .NET only might be the ability to reliably acquire our spin lock type. Recall from Chapter 6, Data and Control Synchronization, that managed threads can be aborted, and that the acquisition of CLR monitors via the language supported keywords ensures an abort can't lead to an orphaned lock. Wouldn't it be nice if we supported this too? We can do so by adding a `ReliableEnter` method. Everything else about the above implementation remains the same.

```

using System;
using System.Runtime.ConstrainedExecution;
using System.Threading;

struct SpinLock
{
    // As before ...

    [ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
    public void ReliableEnter(ref bool taken)
    {
        Interlocked.CompareExchange(
            ref taken,
            true,
            false);
    }
}

```

```
{  
    Thread tid = Thread.CurrentThread.ManagedThreadId;  
    if (m_state == tid)  
        throw new Exception("Recursion not allowed");  
  
    SpinWait sw = new SpinWait();  
    while (true)  
    {  
        if (m_state == LOCK_AVAILABLE)  
        {  
            Thread.BeginCriticalRegion();  
  
            RuntimeHelpers.PrepareConstrainedRegions();  
            try { /*intentionally blank*/  
            finally  
            {  
                if (Interlocked.CompareExchange(  
                    ref m_state, tid, LOCK_AVAILABLE) ==  
                    LOCK_AVAILABLE)  
                {  
                    taken = true;  
                }  
            }  
  
                if (taken) break; // Lock acquired, leave  
  
                Thread.EndCriticalRegion();  
            }  
  
            sw.SpinOnce();  
        }  
    }  
}
```

`ReliableEnter` functionally achieves the same as our previous `Enter` method, but with some additional reliability guarantees. We have marked the method with `ReliabilityContractAttribute` to indicate that it will never corrupt state and may fail with an exception (e.g., if recursion is detected). The main loop is restructured slightly to make things easier to follow. We call the `System.Runtime.ConstrainedExecution.RuntimeHelpers` method `PrepareConstrainedRegions` to enter the CER. This call ensures the JIT compiler has pre-jitted all code run inside the `finally` block (a one time cost) in addition to probing to ensure enough stack exists at the time of the call (a cost incurred during each call). This, in addition to the guarantee that CLR threads won't abort us mid-CER, ensures that the

finally block will run to completion. (Just running in the finally block would be sufficient without a CER if we didn't care about rude thread aborts.) This in turn ensures that callers can rely on the taken ref parameter being reliably set to true when the lock was acquired.

Using this lock alters the ordinary lock acquisition pattern.

```
SpinLock slock = ...;
void f()
{
    bool taken = false;
    try
    {
        slock.ReliableEnter(ref taken);
        // Execute critical region ...
    }
    finally
    {
        if (taken)
            slock.Exit();
    }
}
```

We needn't check taken after calling ReliableEnter because the only way it returns nonexceptionally is when the lock has been acquired, but do check it in the finally block. If a thread abort occurs before ReliableEnter has finished, the finally block will correctly skip the lock release. But if one happens anywhere else after the lock was acquired, we are guaranteed that taken will be true, and, thus, we will release the lock appropriately. Real life scenarios would also need to check that protected state was not corrupt.

Mellor-Crummey-Scott (MCS) Locks

The Mellor-Crummey-Scott (MCS) lock was invented by two researchers, John Mellor-Crummey and Michael Scott (see Further Reading), hence its name. The idea builds on the TATAS lock in order to reduce memory contention for the cache line on which that the lock's state lives. The only real difference is that instead of spinning on reads of the shared lock state, threads spin on a thread private lock state flag. Each thread that detects contention allocates a new local flag and enqueues it onto a shared waiter list. The thread

proceeds to spin on its own local flag. When the lock holder subsequently exits the lock, it signals one of the waiting threads, and the awakened thread then tries to acquire the lock as usual.

Here's a sample implementation in C#.

```
#pragma warning disable 0420

using System;
using System.Threading;

public struct ScalableSpinLock
{
    private volatile int m_state;
    private const int LOCK_AVAILABLE = 0;
    private volatile LockFreeStack<SpinLockFlag> m_waiters;

    public void Enter()
    {
        Thread tid = Thread.CurrentThread.ManagedThreadId;
        if (m_state == tid)
            throw new Exception("Recursion not allowed");

        Thread.BeginCriticalRegion();
        if (Interlocked.CompareExchange(
            ref m_state, tid, LOCK_AVAILABLE) != LOCK_AVAILABLE)
        {
            // Enqueue our flag.
            SpinLockFlag flag = new SpinLockFlag();

            try
            {
                // Spin until it has been set and we succeed.
                SpinWait sw = new SpinWait();
                do
                {
                    flag.m_flag = SpinLockFlagEnum.Reset;
                    GetWaiters().Push(flag);
                    Thread.EndCriticalRegion();

                    // So long as it wasn't released before we pushed...
                    if (m_state != LOCK_AVAILABLE)
                    {
                        // Spin until we see the lock available.
                        while (flag.m_flag != SpinLockFlagEnum.Set)
                            sw.SpinOnce();
                    }
                }
            }
        }
    }
}
```

```
        Thread.BeginCriticalRegion();
    }
    while (Interlocked.CompareExchange(
        ref m_state, tid, LOCK_AVAILABLE) != LOCK_AVAILABLE);

        flag.m_flag = SpinLockFlagEnum.Done;
    }
    catch
    {
        // If we've died due to an exception, signal someone.
        // This ensures no lost wake-ups.
        flag.m_flag = SpinLockEnum.Done;
        SignalOneWaiter();
        throw;
    }
}

public void Exit()
{
    Thread tid = Thread.CurrentThread.ManagedThreadId;
    if (m_state != tid)
        throw new Exception("Lock not owned by thread");

    m_state = LOCK_AVAILABLE;
    SignalOneWaiter();

    Thread.EndCriticalRegion();
}

private void SignalOneWaiter()
{
    SpinLockFlag flag;
    while (GetWaiters().TryPop(out flag))
    {
        if (flag.m_flag != SpinLockFlag.Done)
        {
            flag.m_flag = SpinLockFlag.Set;
            break;
        }
    }
}

private LockFreeStack<SpinLockFlag> GetWaiters()
{
    if (m_waiters == null)
        Interlocked.CompareExchange(
            ref m_waiters, new LockFreeStack<SpinLockFlag>, null);
    return m_waiters;
}
```

```
class SpinLockFlag
{
    internal volatile SpinLockFlagEnum m_flag;
}

enum SpinLockFlagEnum
{
    Reset = 0,
    Set = 1,
    Done = 2
}
```

Most of the code shown is very similar to the `SpinLock` in C# shown earlier. The interesting changes are what happens when the lock is found to be not available and what happens in the `SignalOneWaiter` method. Notice also that a fairly similar approach could have been used to build an event-based lock, to avoid spinning indefinitely. Instead of using wait lists and spin flags, we'd just use an ordinary kernel event object. This would make it usable in cases where wait times are expected to be long.

Where Are We?

We've now put a lot of pieces together. All of the core concurrency mechanisms of the platform are behind us, and we've seen many of them being used to build concurrent data structures such as containers and parallel algorithms. And we've spent time exploring the performance ramifications of it all.

This chapter explored parallel hardware and its impacts on parallel software performance and scalability, particularly in the realm of memory issues. It's probably a worthwhile exercise to reread some earlier chapters with these concepts in mind. We then took some time to understand important fundamental concepts such as parallel speedup, and came to realize the humbling nature of Amdahl's Law. Finally, we closed on some important specific information about when it's appropriate to spin wait and how to properly do it.

In the next chapter, we'll look at another area of practical concern to programmers building real concurrent systems: input and output. The platform provides a lot of rich support around asynchronous I/O, and understanding

how to use these facilities to avoid blocking threads is crucial to getting a well performing system.

FURTHER READING

- G. M. Amdahl. Validity of the Single-processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, Vol. 30 (1967).
- D. E. Culler, J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach* (Morgan Kaufmann, 1998).
- J. Duffy. Concurrency for Scalability. *MSDN Magazine* (2006).
- M. Friedman, O. Pentakalos. *Windows 2000 Performance Guide* (O'Reilly Media, 2002).
- J. Gustafson. Reevaluating Amdahl's Law. In *Communications of the ACM* 31(5) (1988).
- J. L. Hennessy, D. A. Patterson. *Computer Architecture: A Quantitative Approach*, Fourth Edition (Morgan Kauffman, 2006).
- W. D. Hillis. *The Connection Machine* (MIT Press, 1993).
- A. R. Karlin, K. Li, M. S. Manasse, S. Owicki. Empirical Studies of Competitive Spinning for a Shared-memory Multiprocessor. In *ACM SIGOPS Operating Systems Review*, Vol. 25, Issue 5 (1991).
- C. Lyon. Server, Workstation and Concurrent GC. Weblog article: <http://blogs.msdn.com/clyon/archive/2004/09/08/226981.aspx> (2004).
- J. M. Mellor-Crummey, M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. In *ACM Transactions on Computer Systems*, Vol. 9, No. 1 (1991).
- H. Pulapaka, B. Vidolov. Performance: Find Application Bottlenecks with Visual Studio Profiler. *MSDN Magazine* (2008).
- H. Sutter. Going Superlinear. *Dr. Dobb's Journal* (2008).

PART IV

Systems

15

Input and Output

MOST PROGRAMS TODAY spend the majority of their time performing I/O versus pure computational work. This can encompass reading from and writing to files on disk, making Web service invocations, doing raw network socket communication, and so on. For anybody wanting to use parallelism to speed things up, this can pose some unique challenges. There's *one disk* on most client machines, after all, so if most of the time is spent waiting for it, how are we to speed things up? If we parallelize across 16 cores, and yet all of those threads just spend most of their time accessing a single disk, I/O will be a bottleneck limiting our speedup.

I/O is interesting (and challenging) for another reason: I/O operations, much like synchronization waits, block the thread of execution. Just as having many threads doing nothing but waiting for a single hot lock is a bad idea, having lots of threads doing I/O simultaneously against a single resource is also usually a bad idea. It can result in context switching, caches becoming cold, and a variety of other secondary performance effects. I/O often also causes responsiveness issues in GUI programs. This is especially true when very long latencies are involved, like accessing network resources, causing the notorious Not Responding message to be placed into an application's title bar. A related problem is that when a runaway I/O has been made, it can be difficult to cancel its effects when they are no longer desired (e.g., when a user has clicked a Cancel button in the application's GUI).

We explore the impact to GUIs further in the next chapter, which will build on this chapter's content.

In all of these cases, the net effect is the same: in a responsive, scalable system, the ripple effect of synchronous I/O can be substantial. Threads are wasted (space), and performance degrades (time). Sometimes this is just inherent in the problem; there isn't any work to do while the I/O happens. In other cases, I/O is so short and the latency so predictable that synchronous I/O is more efficient (not to mention easier to program). For many cases, however, the Windows platform's deep support for asynchronous I/O can be used to achieve better results. Asynchronous I/O masks latency by eschewing waiting while an asynchronous I/O is in process.

This chapter will review asynchronous I/O in depth. These capabilities are surfaced through various asynchronous file and socket APIs in addition to **I/O completion ports**, a scalable I/O completion mechanism. We'll see how this works from both native and managed code. We'll then look into **I/O cancellation**, which allows cancellation of runaway I/O requests. This, as noted above, is particularly useful when building responsive GUIs.

Overlapped I/O

Asynchronous I/O on Windows is generally referred to as **overlapped I/O**. While the name is a little funny sounding, conceptually it allows you to overlap one or more I/O requests with other useful work. While there are many details and a few different modes of how asynchronous I/O is used in the programming model, they all work very similarly. First, you must initiate an I/O operation, much like you would an ordinary synchronous I/O. The difference is that the request returns right away so the caller can continue doing other work. The OS will keep track of all outstanding asynchronous I/O requests, manage them, and ensure each eventually executes by using interrupts and working directly with the I/O device driver.

Notice from this description that no thread is needed for the I/O as it executes. This is a tremendous benefit, given the overheads that threads imply. You can effectively have an unlimited number of outstanding I/Os running at any given time for a single thread.

Once the I/O executes and some result is ready for the program, user-mode code will again be notified. It is this last notification step that differs from one completion model to the next. There are actually six different models: (1) synchronous completion for “fast” I/O, (2) polling, (3) signaling the device kernel object directly, (4) signaling an event object provided when I/O was started, (5) posting a packet to an I/O completion port, or (6) posting an APC to the initiating thread. We’ll discuss the mechanics of each in just a few pages.

Asynchronous I/O carries a number of benefits.

- CPU work can happen while the operation runs in the background, effectively hiding the latency involved with I/O. Disk and network I/O are orders of magnitude more latent than memory operations. The result is that useful work can be done rather than introducing idle time, gaps in computation, and unnecessary context switches that result from blocking on I/O requests.
- Initiating multiple operations for many devices at once allows those devices to do work concurrently and independently, leading to better utilization of the machine. Each device can complete in whatever order it manages to finish, without needing to serialize each call one after the other. For example, we can load a Webpage over the network while simultaneously mapping a file from disk into memory. Because the two are not related and rely on different hardware devices, they can happen entirely independently and concurrently.
- Having multiple outstanding requests for even just a single device can increase utilization, leading to an overall speedup. For example, having multiple outstanding disk I/Os will allow the I/O subsystem to optimize the movement of the hard disk arm to reduce seek time. Similarly, having multiple network requests outstanding can ensure that requests complete as they are ready; this is particularly useful since each request will complete in some unpredictable order based on the latency and traffic of network hops in between.

Using asynchronous I/O is crucial to obtain good scalability on heavily loaded servers. Similarly, asynchronous I/O is important for any parallel

algorithms that use I/O in or around the computation, to achieve good scaling. As programs become more connected over time and more data must be loaded from disk and analyzed, high- and variable-latency operations will become more prevalent. If this latency isn't hidden, there will be little chance to fully utilize the available CPU power, leading to less efficient scaling on multiprocessor machines. This is an undesirable situation.

You'll find that Win32 offers a much more exhaustive set of primitives for doing asynchronous I/O than .NET does. There are more ways to rendezvous with an outstanding I/O request than are available in the .NET Framework, for example, although they are vastly similar patterns. This power comes at a cost; understanding it and using it all effectively is a difficult proposition. .NET's simpler support is often good enough for most situations. But because it covers more ground and lays a good foundation, we'll start by looking at Win32.

Overlapped Objects

No matter which of the six mechanisms you choose for completing I/O requests, one thing is common: you'll be using a common data structure named **OVERLAPPED** to access the results of asynchronous I/O operations. This structure communicates information about the operation and its completion, such as how many bytes were transferred. It looks like this.

```
typedef struct _OVERLAPPED
{
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union
    {
        struct
        {
            DWORD Offset;
            DWORD OffsetHigh;
        };
        PVOID Pointer;
    };
    HANDLE hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

There is also an equivalent value type in .NET's `System.Threading` namespace.

```
[StructLayout(LayoutKind.Sequential, ComVisible(true))]
public struct NativeOverlapped
{
    public IntPtr InternalLow;
    public IntPtr InternalHigh;
    public int OffsetLow;
    public int OffsetHigh;
    public IntPtr EventHandle;
}
```

Most of these fields are for system use only. For instance, `Internal` is used to carry error information around in an OS specific way, and `InternalHigh` provides the length of data transferred (for nonerror transfers). `Offset` and `OffsetHigh` provide information about the start and end position of the file I/O in question, but are 0 if the operation wasn't file related. The only field that will be of specific interest is the `hEvent` field, as we'll see later, which allows you to provide an event that will be automatically signaled when I/O completes.

In .NET, you will create `NativeOverlapped` objects using the `Overlapped` class, also in the `System.Threading` namespace. It provides several APIs that convert between the managed object and a `NativeOverlapped` value that can then be used in asynchronous I/O operations. The `Pack` and `Unpack` methods perform these conversions. There is also a `Free` method that de-allocates the associated native memory.

```
[ComVisible(true)]
public class Overlapped
{
    // Constructors
    public Overlapped();
    public Overlapped(
        int offsetLo,
        int offsetHi,
        IntPtr hEvent,
        IAsyncResult ar
    );

    // Static Methods
    public static unsafe void Free(
        NativeOverlapped * nativeOverlappedPtr
    );
    public static unsafe Overlapped Unpack(
        NativeOverlapped * nativeOverlappedPtr
    );
}
```

```
// Instance Methods
public unsafe NativeOverlapped * Pack(IOCompletionCallback iocb);
public unsafe NativeOverlapped * Pack(
    IOCompletionCallback iocb,
    object userData
);
public unsafe Overlapped UnsafePack(
    IOCompletionCallback iocb,
    object userData
);

// Properties
public IAsyncResult AsyncResult { get; set; }
public IntPtr EventHandleIntPtr { get; set; }
public int OffsetHigh { get; set; }
public int OffsetLow { get; set; }
}
```

(This class contains a few obsolete APIs. They have been omitted.)

It's worth mentioning right away that it's fairly uncommon that you'll even need to touch these types. Because of this fact, we won't spend too much time discussing them. If you're doing asynchronous file or sockets I/O, for instance, using the classes we'll be looking at later, they have encapsulated all of its usage within. These APIs become necessary if you are doing custom Win32 interop, or using the `ThreadPool.UnsafeQueueNativeOverlapped` function to access the CLR `ThreadPool`'s I/O completion port as a work item dispatcher.

There's a bit of magic hidden inside these APIs, and, to be truthful, they were designed to facilitate specific asynchronous I/O usage in the .NET Framework, not to be generally useful. The `Pack` method accepts an I/O callback and optional user data. The callback is embedded at the end of the `NativeOverlapped` object to which a pointer is returned so the CLR `ThreadPool`'s I/O completion logic can find it and run it once the I/O completes. The `userData` must be a `byte[]` or `byte[][]` and is automatically pinned so that the I/O data may safely be written to it. The `NativeOverlapped` structure is allocated such that it will never be moved (e.g., by the GC) and is also tracked so that, even if the `AppDomain` in which it is allocated gets subsequently unloaded, the memory will be kept stable until the I/O completes. Notice there is no finalization involved here. This is one of the few places in the .NET Framework where, if you forget to free the `NativeOverlapped` after

having packed it, memory can leak. The `Unpack` method allows you to retrieve the managed object's equivalent native object.

Given an `OVERLAPPED` in Win32, you may query the status of any I/O issued against it.

```
BOOL WINAPI GetOverlappedResult(
    HANDLE hFile,
    LPOVERLAPPED lpOverlapped,
    LPDWORD lpNumberOfBytesTransferred,
    BOOL bWait
);
```

This allows you to query the status of an outstanding I/O request. Given the file `HANDLE` and a pointer to the `OVERLAPPED` structure being used for an asynchronous operation, this API will check whether it has completed. If it has, the API returns `TRUE` and the number of bytes transferred is stored into `lpNumberOfBytesTransferred`. Else, if the `bWait` argument is `TRUE`, the API blocks until the I/O has finished and then returns the result of the I/O as usual. (The waiting happens via the `OVERLAPPED`'s `hEvent` field, if non-NULL, or the device kernel object itself otherwise. More on this later.) If `bWait` is `FALSE` and I/O is still in progress, the API returns `FALSE` and `GetLastError` will return `ERROR_I/O_INCOMPLETE`.

Though it is imperative that an `OVERLAPPED` data structure is never freed while an I/O is in flight, it's possible to pool and reuse them. Most server applications will use heap allocation for the memory associated with `OVERLAPPED` objects, which, when a large number of I/Os are happening (as is common on servers), can lead to wasted time spent allocating and freeing them. While you need to guarantee structures aren't used by multiple I/Os at once, the problem is akin to any sort of object pooling problem, for example, a reclamation policy must be decided upon, per CPU caches can be used to reduce lock contention, and so forth. In fact, the CLR internally pools Overlapped data inside a cache whenever you call the constructor and `Free`.

A new API was added to Windows Vista and Server 2008 to take advantage of the fact that many I/Os use caches of `OVERLAPPED` data structures. When an I/O completes in the Windows kernel, it needs to lock the virtual memory pages containing the `OVERLAPPEDs` to guarantee they

don't get paged out while devices are copying data to them. But all of this locking adds overhead to each I/O completion. The `SetFileIoOverlappedRange` function tells the kernel to lock the memory associated with a particular file's OVERLAPPED structures, so that it can avoid this overhead on subsequent I/Os.

```
BOOL WINAPI SetFileIoOverlappedRange(
    HANDLE FileHandle,
    PUCHAR OverlappedRangeStart,
    ULONG Length
);
```

When called, you specify the start address `OverlappedRangeStart` for your OVERLAPPED objects along with the `Length` of the array (e.g., if you are pooling). Calling this function is irreversible for a period of time and will only work with unbuffered I/O. This adds to nonpageable virtual memory usage (much like `VirtualLock`), so it should be used with care. Aggressive use on many files may lead to the OS needing to page other important virtual memory pages to disk. The locked pages are automatically unlocked when the file `HANDLE` is later closed.

Win32 Asynchronous I/O

There are two major components to using asynchronous I/O: (1) how you **initiate** an asynchronous operation, and (2) how you **rendezvous** with (or react to) the completion of that operation. The first depends a lot on what kind of asynchronous I/O you're performing (e.g., files versus network), and the second is more general to all asynchronous I/O. So we'll treat them in that order, starting with how to do asynchronous file I/O. Since much of the API detail is specific to Win32 or .NET, we'll examine them separately in turn.

Initiating Asynchronous Device (“File”) I/O

Because the `ReadFile`, `WriteFile`, and related functions operate on several kinds of devices and kernel objects, they are lumped together in one section. These devices include: files on disk, mailslots, serial and parallel ports, and named pipes. In fact, the only resource that supports Win32 asynchronous I/O directly that isn't in this file oriented category is sockets.

Each of the aforementioned resources must be created for asynchronous access explicitly before the asynchronous versions of read and write APIs can be used. All but one use the `CreateFile` function to open a `HANDLE` that can be used for reading and writing (files, mailslots, and serial and parallel ports), while `CreateNamedPipe` is used for pipes. All of this is fairly straightforward, so let's run through the relevant creation flags. We'll ignore the other interesting but nonconcurrency specific aspects of these functions.

```
HANDLE WINAPI CreateFile(
    LPCTSTR lpFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile
);
HANDLE WINAPI CreateNamedPipe(
    LPCTSTR lpName,
    DWORD dwOpenMode,
    DWORD dwPipeMode,
    DWORD nMaxInstances,
    DWORD nOutBufferSize,
    DWORD nInBufferSize,
    DWORD nDefaultTimeOut,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes
);
```

In order for the resulting `HANDLE` to be usable in subsequent asynchronous operations, you must pass the `FILE_FLAG_OVERLAPPED` flag in the `dwFlagsAndAttributes` argument (for `CreateFile`) or the `dwOpenMode` argument (for `CreateNamedPipe`). `CreateFile` can block because it must access the disk while opening; there is no asynchronous version of the `CreateFile` API itself, which is a limitation. Named pipes separate creating the connection itself from the creation of a new `HANDLE`, and the `ConnectNamedPipe` function does in fact support asynchronous execution much like with reading and writing.

```
BOOL WINAPI ConnectNamedPipe(
    HANDLE hNamedPipe,
    LPOVERLAPPED lpOverlapped
);
```

Once you have a HANDLE opened via CreateFile or CreateNamedPipe, you can read from and write to it using any of the usual file read and write functions.

```
BOOL ReadFile(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPDWORD lpNumberOfBytesRead,
    LPOVERLAPPED lpOverlapped
);
BOOL ReadFileEx(
    HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
BOOL WriteFile(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped
);
BOOL WriteFileEx(
    HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
```

In addition to these APIs, there is a more general purpose function to send a control code directly to a device driver, DeviceIoControl. Unless you're writing low-level device interface code, you are far less likely to need to use this particular function.

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

Again we won't go into each of these in great detail here. The functions ending in `Ex` are asynchronous only, while the others support both synchronous and asynchronous I/O. The determining factor for those is whether the `lpOverlapped` argument is `NULL` or not. If the file `HANDLE` was originally opened for overlapped I/O, by the way, you are *required* to supply an `OVERLAPPED` structure when reading or writing; that is, you can't use the `HANDLE` for synchronous I/O. The `LPOVERLAPPED_COMPLETION_ROUTINE` is a function pointer definition. The callback routines will be discussed in detail later, but its definition is as follows:

```
VOID CALLBACK FileIOCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped
);
```

Asynchronous file I/O is distinctly different from synchronous file I/O in one interesting way; unlike synchronous I/O where each file `HANDLE` tracks the current position pointer, enabling each read and write to pick up where the previous one left off, asynchronous I/O requires that the starting offset is specified for each new file operation. In other words, if you've already read 4,096 bytes from the file, you will need to explicitly pass 4,096 as the start of the next read. The offset is specified with the `DWORD` fields `Offset` and `OffsetHigh` in the `OVERLAPPED` structure. They are combined into a 64-bit value as `(Offset | ((LONGLONG)OffsetHigh << 32))`. *Note that this requirement applies to file I/O only:* these fields must be explicitly set to `0` for nonfile I/O operations, otherwise reading and writing will return an error.

In addition to requirements around `Offset` and `OffsetHigh`, the read and write APIs also require that the `hEvent` field of the `OVERLAPPED` structure be set. We'll see how it gets used in the various completion methods below, but for now we will always set it to `NULL`.

End of file is treated subtly differently when doing asynchronous I/O too. Instead of completing the I/O and simply saying that 0 bytes were read, the API will return `FALSE`, and `GetLastError` will return `ERROR_HANDLE_EOF`.

Finally, the thread that initiates an asynchronous I/O must not exit before that I/O completes. Doing so will possibly prevent the completion

from ever being seen by your program. It is possible to dynamically query whether the current thread has I/O pending.

```
BOOL WINAPI GetThreadIOPendingFlag(HANDLE hThread, PBOOL lpIOIsPending);
```

The function takes a HANDLE to the thread to inquire about and returns TRUE in lpIOIsPending if there are outstanding asynchronous I/O requests on the thread.

By exiting before pending I/O completes, some I/O packets would be lost completely. This might subsequently impact the application code because some I/O completion events would never happen. In addition, this can lead to memory leaks because it's commonplace for associated resources, such as buffers and OVERLAPPED data structures, to be freed in the I/O completion routines. Ensuring threads don't exit before pending I/O is completed can be somewhat difficult, especially for ordinary threads that are not under the control of low-level asynchronous APIs. Components that manage threads, such as the CLR and Win32 thread pools, ensure that threads do not exit prior to all asynchronous I/O finishing.

Completing an Asynchronous I/O

After initiating an asynchronous I/O operation, we need to rendezvous with it to complete the I/O. This usually entails processing a block of data that has been read or written, and/or to kick off another asynchronous I/O request for the next block of data. As already stated at the outset, there are several mechanisms for this, useful for different reasons. Choosing one over the other often entails many of the same tradeoffs we examined in Chapter 8, Asynchronous Programming Models, where the .NET APM pattern provides a similar set of completion options.

No matter what mechanism you choose, one thing is extremely important to keep in mind: the data buffer and OVERLAPPED structure involved in the read or write operation *must* be kept alive for the duration of the I/O operation. Data will be copied into and out of these while the I/O routine executes; if you were to free the data structures prematurely, the device would then attempt to access freed memory—leading to memory corruption and possible crashes. This was already mentioned earlier, but is important enough to repeat again.

Method #1: Synchronous Completion. If Windows is able to complete your I/O request quickly, no separate rendezvous will be necessary. This can happen because the OS keeps a file cache of recently accessed files in memory, alleviating the need to access the disk altogether. If a cache hit occurs, there's no need to pay extra asynchronous rendezvous overhead that arises when you use overlapped I/O. You must always handle this case in your code and have no control over whether it happens or not.

When an I/O request completes synchronously, the call to `ReadFile`, `ReadFileEx`, `WriteFile`, or `WriteFileEx` returns TRUE. The asynchronous completion that would have otherwise been associated with the I/O request will not happen. If synchronous completion does not occur, the function returns FALSE and `GetLastError` will return `ERROR_IO_PENDING`. This might come as a surprise, but yes—successfully starting an asynchronous I/O is communicated as an error.

Here's a small snippet of code. It reads 4,096 bytes from a file starting at position 8,192 bytes from the beginning of the file. Although we open the file for overlapped I/O, the read operation may still complete synchronously.

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

OVERLAPPED olap;
olap.Offset = 8192;
olap.OffsetHigh = 0;
olap.hEvent = ...;

BYTE data[4096];
DWORD bytesRead;

if (ReadFile(hFile, &data, sizeof(data), &bytesRead, &olap))
{
    // Synchronously completed...
    // data contains bytesRead number of bytes read from disk cache.
}
else
{
    if (GetLastError() == ERROR_I/O_PENDING)
    {
        // Async I/O is happening in the background ...
        // We will complete it through async-specific mechanisms.
    }
    else
```

```
{  
    // Other kind of error ...  
}  
}
```

Notice here that we're passing a stack allocated array (`data`) as the location where the read operation will put data from the read. Recall from earlier that this data must last at least as long as the asynchronous I/O itself. So this technique, while applicable to such a simple example, is usually not going to work. We'll continue using it as long as possible because it simplifies the example, but typically you'll need to resort to heap allocation and manual freeing of buffers.

If I/O completion is used, a completion packet will still be generated even though we are able to handle the I/O synchronously. Additionally, the file HANDLE will be set by the OS (as we'll see later). If code has been written to handle the synchronous completion, these two things are unnecessary and can lead to performance degradation. A new API was added to Windows Vista and Windows Server 2008 to allow suppression of these steps.

```
BOOL WINAPI SetFileCompletionNotificationModes(  
    HANDLE FileHandle,  
    UCHAR Flags  
);
```

Two flags are available for the `Flags` argument, corresponding directly to the two unneeded steps mentioned above: `FILE_SKIP_COMPLETION_PORT_ON_SUCCESS` avoids queuing a packet to a port if the `HANDLE` has been bound, and `FILE_SKIP_SET_EVENT_ON_HANDLE` skips setting the file `HANDLE`. If a custom `HANDLE` was provided in the `hEvent` field of the `OVERLAPPED` structure, it will still be set even if this flag was passed.

Method #2: Polling with `GetOverlappedResult`. Next to synchronous completion, the simplest rendezvous technique is to poll for completion. Polling is the act of periodically checking whether the I/O has completed: if it hasn't, some useful application specific work can be done, and if it has finished, the I/O request can be processed accordingly. This is done using the `GetOverlappedResult` API shown earlier.

The following code snippet demonstrates how one might use polling to continue doing work while some asynchronous I/O is underway. Synchronous completion is omitted (see the previous code snippet).

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

OVERLAPPED olap;
olap.Offset = 8192;
olap.OffsetHigh = 0;
olap.hEvent = NULL;

BYTE data[4096];
DWORD bytesRead;

if (!ReadFile(hFile, &data, sizeof(data), &bytesRead, &olap))
{
    switch (GetLastError())
    {
        case ERROR_I/O_PENDING:
            // Asynchronous I/O is still underway.
            while (TRUE)
            {
                // Do some useful work in the meantime...

                if (!GetOverlappedResult(
                    hFile, &olap, &bytesRead, FALSE))
                {
                    if (GetLastError() == ERROR_I/O_INCOMPLETE)
                    {
                        // Async I/O is still occurring. We just loop
                        // around and keep doing some useful work.
                        continue;
                    }
                    // (Handle other types of errors.)
                }

                // Asynchronous I/O is done -- just exit the loop.
                break;
            }

            break;

        // (Handle other types of errors.)
    }
}
else
```

```
{  
    // Error or synchronous completion ...  
}  
  
// Process the results of I/O ...
```

In this example, I/O happens completely asynchronously. Once we notice a TRUE return value from `GetOverlappedResult`, we switch over to processing it. Otherwise, there's a placeholder where "useful" work is done. This might involve any sort of application specific bookkeeping, such as computing some background statistics, running a Windows message loop to process GUI message, dispatch COM RPC calls, APCs, and so forth. You could even dispatch additional I/O requests. If you find that there's no useful work to do, pass TRUE to the `GetOverlappedResult` function and it will block until the I/O completes.

A higher performance macro is available that inspects data on the OVERLAPPED object instead of making a function call. This can be used instead of `GetOverlappedResult`.

```
BOOL HasOverlappedIoCompleted(LPOVERLAPPED lpOverlapped);
```

The polling approach generally has the benefit of being low overhead because there are no additional kernel objects to create and manage. The code also looks like a synchronous I/O would have, so there isn't much restructuring of program logic needed. A disadvantage of polling, however, is that there may be latency between the time an I/O completes and the time our loop gets around to noticing and processing it. These delays can add up.

Method #3: Waiting on the Device Handle Directly. The polling mechanism shown above allows you to block waiting for I/O to complete by passing TRUE for the `bWait` parameter to `GetOverlappedResult`. This is often sufficient if you'd like to wait. But as we saw in prior chapters, sometimes you need more flexibility over the way a thread waits. Maybe you need to pump for GUI messages and run APCs. Or maybe you'd like to use a timeout so that if I/O doesn't complete quickly, you can go off and do some more application specific bookkeeping (or at least check if any needs to be done). Or perhaps you'd like to wait for multiple kernel objects simultaneously,

with `WaitForMultipleObjects`, including the possibility of waiting on multiple outstanding asynchronous I/O operations.

All of this is simple to achieve by using the wait APIs to which you've grown accustomed. The question then becomes: What `HANDLE` should be used? The `hEvent` field of the `OVERLAPPED` structure has probably piqued your interest. But we'll get to that shortly. For now, you can wait on the same device `HANDLE` used to start the asynchronous operation itself. The implementation of asynchronous I/O unsignals this `HANDLE` before returning from the function used to start execution and will later signal the `HANDLE` once the I/O completes. Notice that multiple threads may not use the same `HANDLE` in this manner, since the signals will get jumbled up across threads in a way that makes it impossible to determine when I/O has actually finished.

For example, this code waits on the file `HANDLE` to ensure that messages are pumped while we wait for I/O to finish rather than looping around and continuously polling for completion.

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);

OVERLAPPED olap;
olap.Offset = 8192;
olap.OffsetHigh = 0;
olap.hEvent = NULL;

BYTE data[4096];
DWORD bytesRead;

if (!ReadFile(hFile, &data, sizeof(data), &bytesRead, &olap))
{
    BOOL fIODone = FALSE;

    switch (GetLastError())
    {
        case ERROR_I/O_PENDING:
            // Asynchronous I/O is still underway.
            while (!fIODone)
            {
                switch (MsgWaitForMultipleObjects(
                    1, &hFile, FALSE, INFINITE, QS_ALLINPUT))
                {
                    case WAIT_OBJECT_0:
                        // Async I/O completed. Remember byte count.
                }
            }
    }
}
```

```

        bytesRead = olap.InternalHigh;
        fIODone = TRUE;
        break;
    case WAIT_OBJECT_0 + 1:
        // We have a message to dispatch.
        MSG msg;
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        break;
    default:
        // (Handle failure case.)
        break;
    }
}

break;

default:
    // (Handle other types of errors.)
}
}

// Process the I/O...

```

We use the OVERLAPPED structure's InternalHigh field in this example to determine the number of bytes transferred during file I/O. This is identical to the value returned in the out parameter for functions like ReadFile and GetOverlappedResult. Using it directly as shown above avoids having to make a call to GetOverlappedResult after waiting on the device HANDLE completes. The Internal field will contain a non-0 error code if the I/O failed while executing, much like GetLastError for synchronous completion.

Method #4: Waiting on an Event Handle. With the first three techniques, there is a subtle limitation. They only support a single in-flight asynchronous I/O operation against a given device HANDLE at once. Sometimes you'll want to perform multiple asynchronous operations on the same HANDLE at once, such as reading and writing to nonintersecting portions of a file simultaneously. By now, you've probably noticed that the OVERLAPPED structure has a hEvent field. And you've probably also noticed that we keep setting it to NULL in all of the examples above. But you can actually set this

to a valid Win32 HANDLE, such as an event object. If you do, Windows will reset the event while initiating the I/O and set it once I/O finishes. You can then go about waiting on it, similar to waiting on the device HANDLE directly.

This takes advantage of the ability for the Windows file system to intelligently schedule many I/Os targeting the same device. Similar techniques can be used when multiple threads are involved, such as when dealing with a file shared by all clients of a server program.

As an example, this code begins 10 simultaneous read operations against the same file at once and then processes completions in whatever order they happen to finish. We have to create a separate distinct OVERLAPPED structure for each in-flight I/O.

```
// File to be used for many asynchronous IOs:  
HANDLE hFile = CreateFile(  
    "Test.txt",  
    GENERIC_READ,  
    FILE_SHARE_READ,  
    NULL,  
    OPEN_EXISTING,  
    FILE_FLAG_OVERLAPPED,  
    0);  
  
const DWORD    PACK_COUNT = 10;  
const DWORD    BYTES_PER  = 4096;  
  
OVERLAPPED    olaps[PACK_COUNT];  
BYTE *         bytes[PACK_COUNT];  
DWORD          bytesRead[PACK_COUNT];  
HANDLE         inFlightHandles[PACK_COUNT];  
ZeroMemory(inFlightHandles, PACK_COUNT * sizeof(HANDLE));  
  
// Phase 1:  
// Initialize primary structs, byte arrays, and events.  
// Also kick off the asynchronous I/O operations themselves.  
for (int i = 0; i < PACK_COUNT; i++)  
{  
    ZeroMemory(&olaps[i], sizeof(OVERLAPPED));  
    olaps[i].Offset = BYTES_PER * i;  
    olaps[i].OffsetHigh = 0;  
    olaps[i].hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);  
    bytes[i] = new byte[BYTES_PER];  
  
    if (!ReadFile(hFile, bytes[i], BYTES_PER, &bytesRead[i], &olaps[i]))  
    {  
        switch (GetLastError())
```

```
{  
    case ERROR_I/O_PENDING:  
        // Add to the list of pending asynchronous I/O.  
        inFlightHandles[i] = olaps[i].hEvent;  
        break;  
  
    // (Handle other types of errors.)  
}  
}  
}  
  
}  
  
// Phase 2:  
// Go through and process synchronously completed I/O.  
HANDLE hCurrentThread = GetCurrentThread();  
for (int i = 0; i < PACK_COUNT; i++)  
{  
    if (inFlightHandles[i] == NULL)  
    {  
        // Process the results of the synchronous I/O...  
        // bytes[i] and bytesRead[i] contain I/O completion info.  
        inFlightHandles[i] = hCurrentThread;  
    }  
}  
  
// Phase 3:  
// Wait for asynchronous I/O requests, processing as they finish.  
for (int i = 0; i < PACK_COUNT; i++)  
{  
    if (inFlightHandles[i] != hCurrentThread)  
    {  
        DWORD ret = WaitForMultipleObjects(  
            PACK_COUNT, (const HANDLE *)inFlightHandles[0],  
            FALSE, INFINITE);  
        if (ret >= WAIT_OBJECT_0 &&  
            ret < WAIT_OBJECT_0 + PACK_COUNT)  
        {  
            // An asynchronous I/O completed...  
            // bytes[ret] and olaps[ret] contain I/O completion info.  
            inFlightHandles[i] = hCurrentThread;  
        }  
        else  
        {  
            // Error handling ...  
        }  
        i = -1; // Go through the loop again.  
    }  
}
```

```
// Phase 4:  
// Clean up the memory and events we allocated above.  
for (int i = 0; i < PACK_COUNT; i++)  
{  
    delete [] bytes[i];  
    CloseHandle(olaps[i].hEvent);  
}
```

There are four main phases of this code.

First, we allocate the relevant OVERLAPPED structures, BYTE arrays into which data will be copied, and events that will be used to signal completion. We also kick off the asynchronous I/O using `ReadFile`, similar to what has already been shown. We accumulate a list of which operations actually turned into asynchronous I/O versus those that completed synchronously by placing the relevant I/O's event HANDLE into the `inFlightHandles` array in the former case.

In the next phase, we loop through and, for each `inFlightHandles` entry that is `NULL`, we can go ahead and process the I/O right away. It completed synchronously. The relevant information will have been stored into the `bytes[i]` and `bytesRead[i]` arrays during the call to `ReadFile`. We do something that might appear odd after this: we store the current thread's HANDLE into the `inFlightHandles` array where the `NULL` used to be. This is done because it will never become signaled (since the current thread would have to exit). This makes issuing a wait-any style wait a bit easier, which we use in the next phase.

In the third phase, we must wait for asynchronous I/O completions. To do so, we loop through the `inFlightHandles` entries. So long as we see at least one that isn't set to the current thread's HANDLE (meaning it's already finished), we will do a wait-any style `WaitForMultipleObjects`. Once this awakens, we can translate the return into a specific I/O that has finished. The `bytes[ret]` and `olaps[ret]` entries will contain information that we can use to process the completion. We then place the current thread's HANDLE into the `inFlightHandles` array to skip the entry on subsequent waits and restart the loop.

The fourth and final phase is just to delete the buffer memory and close the event handles.

Method #5: APC Callbacks. An alternative that makes the kind of code we just saw simpler is to use APCs as a means to process I/O completions. You saw that `ReadFileEx` and `WriteFileEx` from earlier allow you to pass a callback routine as a `LPOVERLAPPED_COMPLETION_ROUTINE`. As specified, this callback will be executed inside an APC on the thread that initiated the I/O. This can be useful because APCs are generally high performance and don't require that you allocate extra event kernel objects. Compared to the four previous mechanisms, this is often the most efficient technique if you've decided not to use completion ports.

For the completion to be delivered when the I/O finishes, the initiating thread must be in an alertable wait state. It's a good idea to ensure that the code initiating the I/O is also the code that intercepts the APCs. This might seem obvious, but there are easy ways to make mistakes. If you initiate some I/O and then either return control back to a caller, perhaps indirectly due to an exception, or make a call into another API that internally performs an alertable wait, the I/O may finish somewhere else. Strange results may arise. For example, the wait might occur inside a lock or when some thread affine state has been introduced. If an exception is thrown from the completion callback, unexpected results will surely occur. The use of APC completion therefore is constrained to fairly closed scenarios, where code run in between initiating and completing the I/O is tightly controlled.

Here's a version of the wait-any style code shown above that uses APC completion instead.

```
VOID CALLBACK IoCmp(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    LPOVERLAPPED lpOverlapped)
{
    // Process the I/O completion ... gets invoked from an APC.
}

// Elsewhere... file to be used for many asynchronous IOs:
HANDLE hFile = CreateFile(
    "Test.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
```

```
OPEN_EXISTING,
FILE_FLAG_OVERLAPPED,
0);

SetFileCompletionNotificationModes(hFile,FILE_SKIP_SET_EVENT_ON_HANDLE);

const DWORD    PACK_COUNT = 10;
const DWORD    BYTES_PER = 4096;

OVERLAPPED    olaps[PACK_COUNT];
BYTE *        bytes[PACK_COUNT];
DWORD         inFlight = 0;

// Phase 1:
// Initialize primary structs, byte arrays, and events.
// Also kick off the asynchronous I/O operations themselves.
for (int i = 0; i < PACK_COUNT; i++)
{
    ZeroMemory(&olaps[i], sizeof(OVERLAPPED));
    olaps[i].Offset = BYTES_PER * i;
    olaps[i].OffsetHigh = 0;
    olaps[i].hEvent = NULL;
    bytes[i] = new byte[BYTES_PER];

    if (!ReadFileEx(hFile, bytes[i], BYTES_PER, &olaps[i], &IoCmp))
    {
        switch (GetLastError())
        {
            case ERROR_I/O_PENDING:
                inFlight++; // Track number of pending IOs.
                break;

            // (Handle other types of errors.)
        }
    }
    else
    {
        // Process the results of synchronous I/O...
        // bytes[i] and bytesRead[i] contain I/O completion info.
    }
}

// Phase 2:
// Wait for asynchronous I/O requests, processing as they finish.
while (inFlight > 0)
{
    WaitForSingleObjectEx(GetCurrentThread(), INFINITE, TRUE);
    inFlight--;
}
```

```
// Phase 3:  
// Clean up the memory and events we allocated above.  
for (int i = 0; i < PACK_COUNT; i++)  
{  
    delete [] bytes[i];  
    CloseHandle(olaps[i].hEvent);  
}
```

This code looks very similar to the code above, which uses `WaitForMultipleObjects` to wait on an array of event `HANDLEs`. We have simplified it by handling synchronously completed I/O inline. This example also illustrates the trickiness of APC style completion. We must be extremely careful that we do not enter an alertable wait state prior to our call to `WaitForSingleObjectEx`. If we allow an I/O to complete outside of this loop, the `inFlight` counter will not be updated correctly and we may deadlock. A more robust solution would arrange for the APC callbacks themselves to track outstanding I/Os.

Method #6: I/O Completion Ports. If you are building a highly scalable server application or using asynchronous I/O in any serious way, you will probably want to use I/O completion ports as your rendezvous mechanism. In fact, this is the only completion mechanism even exposed in .NET. (Although .NET APIs hide all of the I/O completion usage internally, this section may be interesting for managed developers who want to know “how it all works” under the hood.)

An I/O completion port is like a little miniature scheduler for work items. The work that it schedules takes the form of **I/O completion packets**, and the OS uses logic that attempts to minimize the number of active threads processing packets so as not to oversubscribe processors with too many threads. We saw briefly in Chapter 7, Thread Pools, that the Win32, new Vista, and CLR thread pools each contain a single automatically created I/O completion port per process and manage a set of threads dedicated to processing completion packets from it. These features can be used for any of the kinds of asynchronous I/O we have reviewed in this chapter.

As a brief example, here is code that uses the I/O completion capability of the native thread pool. We initiate a single I/O, and use the thread pool as a way to invoke the callback.

```
VOID CALLBACK IoCmp(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PVOID Overlapped,
    ULONG IoResult,
    ULONG_PTR NumberOfBytesTransferred,
    PTP_IO Io)
{
    // Process the I/O completion ... gets invoked on the thread pool.
}

// Elsewhere... file to be used for many asynchronous IOs:
HANDLE hFile = CreateFile(
    "Test.txt",
    GENERIC_READ,
    FILE_SHARE_READ,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_OVERLAPPED,
    0
);

PTP_IO pIo = CreateThreadpoolIo(
    hFile,
    &IoCmp,
    NULL,
    NULL
);

// Everything else remains similar...
```

We've glossed over coordinating the cleanup of resources such as buffers. Because completions happen on separate threads, it is often necessary to synchronize this cleanup or to have higher level state management put in place.

Digging Deeper into I/O Completion Ports

While the thread pool support for I/O completion ports is incredibly useful—it allows the thread pool to decide when to add or remove threads from the mix and is typically the solution of choice—some circumstances call for a customized solution. Accessing I/O completion ports more directly is certainly possible, but to do so will require a deeper understanding of them.

(You cannot currently create and manage your own I/O completion ports in managed code; they are only available from native code.)

A completion port is just another kind of kernel object that can be created and destroyed. A number of threads may wait on a single completion port. Components may queue completion packets to a specific port when I/O finishes, possibly waking waiting threads. This new work is usually generated by an asynchronous I/O request but can also be queued manually by calling `PostQueuedCompletionStatus`. In any case, once a new packet is queued, the OS decides whether to wake up a thread. If fewer threads than there are processors are actively processing packets, the port will wake one up; otherwise, it makes a more difficult choice. In order to make this decision, the OS keeps omnipresent knowledge of how many threads waited on the port and which ones are actively running. Should a woken thread fail to return to the port for a certain period of time, either because it has blocked or because processing a packet takes some time, the thread will allow additional threads to unblock to process work.

As of Windows Vista and Server 2008, asynchronous I/O completions may borrow one of the threads waiting on a port, instead of forcing a context switch to the thread that issued the asynchronous I/O. This helps to improve scalability and liveness.

There are only three APIs necessary to create and manage I/O completion ports. And one is even optional. The I/O completion ports APIs themselves are strikingly simple, given the vast amount of intelligence they contain within. What makes them seem complicated is the numerous ways of interacting with them indirectly with file APIs, socket APIs, and the like.

The major workhorse is the creation function.

```
HANDLE WINAPI CreateIoCompletionPort(
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort,
    UNLONG_PTR CompletionKey,
    DWORD NumberOfConcurrentThreads
);
```

As with most Win32 creation APIs, this creates a kernel object and returns a `HANDLE` to it. If creation fails, the return value will be `NUL` and `GetLastError` will tell you specifically why it failed. It is common to create a port passing `INVALID_HANDLE_VALUE` for the `FileHandle` and `NULL` for `ExistingCompletionPort`. After doing so, you can then use the same port

to service multiple files, sockets, and /or manually posted packets. Unless there are many, many requests going against a single device HANDLE, having a port dedicated to each one adds unnecessary overhead. Reuse is typically best.

After a port has been created like this, you can then call `CreateIoCompletionPort` and pass a HANDLE to a pre-existing port as the `ExistingCompletionPort` argument. The OS will then use the existing port for the particular file HANDLE (or SOCKET, as we will soon see). The device HANDLE supplied must be one that was opened for overlapped I/O. This is how the legacy thread pool's `BindIoCompletionCallback`, Vista thread pool's `CreateThreadpoolIo`, and the CLR thread pool's `BindHandle` functions are implemented.

The `CompletionKey` is an opaque value that will be supplied to any thread that completes due to I/O completions posted to the particular file (which is irrelevant if a file is not specified). Unfortunately, there's no easy way to supply a callback to run when a thread waiting for work awakens (as with APCs above), but the `CompletionKey` can be a handy way of passing a function pointer that is to be executed by the thread that wakes up. This requires an application specific convention to be established. As you may have guessed, this is exactly how the thread pools work: they have some internal convention for passing completion routines as function pointers and delegates around in the I/O completion registration.

The `NumberOfConcurrentThreads` indicates how many threads the OS should use for servicing packets. Often this should be the number of processors—based on the logic outlined earlier—but doesn't necessarily need to be. For example, if you have many ports in a single process, it may make sense to distribute the number of threads used more evenly. This parameter is ignored if you don't pass `NULL` for the `ExistingCompletionPort`.

So now that you've got a port created, what do you do with one? You'll probably create threads (like the aforementioned thread pools) to wait for packets. Waiting for a completion packet is done with the `GetQueuedCompletionStatus` API.

```
BOOL WINAPI GetQueuedCompletionStatus(
    HANDLE CompletionPort,
    LPDWORD lpNumberOfBytes,
```

```
PULONG_PTR lpCompletionKey,
LPOVERLAPPED * lpOverlapped,
DWORD dwMilliseconds
);
```

This function blocks until a new packet arrives and the thread is selectively unblocked based on the runnable thread throttling logic in the OS. You pass to it the CompletionPort you'd like to wait on, a bunch of arguments into which data associated with the completion packet will be placed, and a dwMilliseconds timeout. The timeout works the same way as those you've seen previously, that is, `INFINITE (-1)` to specify "no timeout," `0` to avoid blocking, or some other number of milliseconds otherwise. The `lpNumberOfBytes` `DWORD` receives the number of bytes associated with the completion, `lpCompletionKey` is set to the key passed to the completion port creation routine, and the `OVERLAPPED` contains additional information about the completion. The API returns `FALSE` if an error or a timeout occurs. To differentiate between the two, call `GetLastError` and look for a return of `WAIT_TIMEOUT`.

Notice that `GetQueuedCompletionStatus` *does not* offer a way to pump for messages or to do an alertable wait. This can cause some problems in systems that use APCs to take back control of threads, for example. In such cases, you may need to rely on timeouts instead.

There is a `GetQueuedCompletionStatusEx` method that was added in Windows Vista and Server 2008, which provides two additional useful features when compared to its counterpart. First, you can receive multiple completion entries at once. This reduces performance overhead, due to fewer kernel transitions and internal locks being taken, and can be useful on heavily loaded server programs that can experience times during which I/Os are finishing faster than they can be processed. Second, you can specify that the wait be alertable.

```
BOOL WINAPI GetQueuedCompletionStatusEx(
    HANDLE CompletionPort,
    LPOVERLAPPED_ENTRY lpCompletionPortEntries,
    ULONG ulCount,
    PULONG ulNumEntriesRemoved,
    DWORD dwMilliseconds,
    BOOL fAlertable
);
```

If multiple completion entries are available on the specified port HANDLE, this function will retrieve up to ulCount of them. It stores the count in ulNumEntriesRemoved and, for each completion entry, an associated structure in the output lpCompletionPortEntries array. When calling this API, you must ensure the array is large enough to store up to ulCount entries since that is the maximum number of records Windows will try to write to the array. The dwMilliseconds argument allows a timeout to be specified, and fAlertable controls the alertability of the wait used internally.

Each entry is represented by a new OVERLAPPED_ENTRY structure.

```
typedef struct _OVERLAPPED_ENTRY
{
    ULONG_PTR lpCompletionKey;
    LPOVERLAPPED lpOverlapped;
    ULONG_PTR Internal;
    DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, * LPOVERLAPPED_ENTRY;
```

Each of these fields (except for Internal, which is reserved for internal use) maps to the respective output parameter for the ordinary GetQueuedCompletionStatus API.

In most cases, completion packets will be posted automatically when Win32 device operations complete. But you can also manually post a completion packet.

```
BOOL WINAPI PostQueuedCompletionStatus(
    HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    ULONG_PTR dwCompletionKey,
    LPOVERLAPPED lpOverlapped
);
```

Posting a packet manually to the CompletionPort specified allows you to generate work for a waiting thread. The waiting thread will awaken with access to the dwNumberOfBytesTransferred, dwCompletionKey, and lpOverlapped structure set in its output arguments. This feature allows you to treat an I/O completion port as if it were a thread pool. In fact, as was mentioned previously, the CLR's thread pool offers the UnsafeQueueNativeOverlapped method for this very purpose. It internally uses PostQueuedCompletionStatus. For more details, refer to Chapter 7, Thread Pools.

Asynchronous Sockets I/O

As with other local devices, the sockets APIs enable asynchronous network operations. The process of using them is similar to asynchronous file I/O, so all of this should sound quite similar. To use a socket asynchronously, you must first open it for overlapped execution using the `WSASocket` function, which can be found in the `Winsock2.h` platform header (and `Ws2_32.lib` and `Ws2_32.dll` Winsock static and dynamic link platform libraries).

```
SOCKET WSASocket(
    int af,
    int type,
    int protocol,
    LPWSAPROTOCOL_INFO lpProtocolInfo,
    GROUP g,
    DWORD dwFlags
);
```

To open for overlapped execution, pass the `WSA_FLAG_OVERLAPPED` flag to `WSASocket` as part of its `dwFlags` argument. Once you have done this, you can use the resulting `SOCKET` asynchronously in any of the following socket functions. Whether asynchronous execution is used or not is solely determined on whether the overlapped structure is `NULL`.

```
BOOL AcceptEx(
    SOCKET sListenSocket,
    SOCKET sAcceptSocket,
    PVOID lpOutputBuffer,
    DWORD dwReceiveDataLength,
    DWORD dwLocalAddressLength,
    DWORD dwRemoteAddressLength,
    LPDWORD lpdwBytesReceived,
    LPOVERLAPPED lpOverlapped
);
int WSASend(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesSent,
    DWORD dwFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
int WSASendTo(
    SOCKET s,
```

```
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
const struct sockaddr * lpTo,
int iToLen,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
int WSARecv(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
int WSARecvFrom(
    SOCKET s,
    LPWSABUF lpBuffers,
    DWORD dwBufferCount,
    LPDWORD lpNumberOfBytesRecvd,
    LPDWORD lpFlags,
    struct sockaddr * lpFrom,
    LPINT lpFromlen,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
int WSAIoctl(
    SOCKET s,
    DWORD dwIoControlCode,
    LPVOID lpvInBuffer,
    DWORD cbInBuffer,
    LPVOID lpvOutBuffer,
    DWORD cbOutBuffer,
    LPDWORD lpcbBytesReturned,
    LPWSAOVERLAPPED lpOverlapped,
    LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
);
BOOL TransmitFile(
    SOCKET hSocket,
    HANDLE hFile,
    DWORD nNumberOfBytesToWrite,
    DWORD nNumberOfBytesToSend,
    LPOVERLAPPED lpOverlapped,
    LPTTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
    DWORD dwFlags
);
```

```
BOOL TransmitPackets(
    SOCKET hSocket,
    LPTRANSMIT_PACKETS_ELEMENT lpPacketArray,
    DWORD nElementCount,
    DWORD nSendSize,
    LPOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

The `AcceptEx` function allows you to asynchronously accept new connections while the other functions allow you to perform asynchronous sends and receives on existing connections. Given the sheer number of arguments for all of these functions, there is a lot of socket specific knowledge you'll need to use them. This book isn't about building network programs per se—there are plenty of good resources on that already—so we'll skip those aspects and focus just on how to use them for asynchronous programming. Doing so is crucial for building scalable sockets applications, particularly on heavily loaded servers.

`WSAOVERLAPPED` has the same structure as `OVERLAPPED`. The completion routine type, `LPWSAOVERLAPPED_COMPLETION_ROUTINE`, is a function pointer to a slightly different signature than the file based completion routines seen earlier.

```
VOID CALLBACK SocketCompletionRoutine(
    DWORD dwError,
    DWORD cbTransferred,
    LPWSAOVERLAPPED lpOverlapped,
    DWORD dwFlags
);
```

If the `lpOverlapped` argument to any of the functions above is non-NULL, the request may complete asynchronously. As with the device functions seen earlier, however, the request may complete synchronously. Asynchronous execution is indicated by a return value of `SOCKET_ERROR` and a subsequent return value of `WSA_IO_PENDING` from `WSAGetLastError`. Otherwise, the call completes the same as any ordinary synchronous I/O, and any pertinent output parameters (such as `lpNumberOfBytesRecv`) will have been set. As with file I/O, if the thread that initiates an asynchronous sockets request exists before that request has completed, that request will be canceled automatically by the OS.

The other completion styles for sockets I/O are basically identical to those for device I/O. Instead of `GetOverlappedResult`, you will use `WSAGetOverlappedResult`.

```
BOOL WSAAPI WSAGetOverlappedResult(
    SOCKET s,
    LPWSAOVERLAPPED lpOverlapped,
    LPDWORD lpcbTransfer,
    BOOL fWait,
    LPDWORD lpdwFlags
);
```

As with `GetOverlappedResult`, passing a value of `TRUE` for `fWait` will block the thread until the specific asynchronous operation finishes. Otherwise, if the function returns `FALSE`, the `WSAGetLastError` function will return `WSA_IO_INCOMPLETE` to indicate I/O is in progress.

To bind a socket to an I/O completion port, you use the same steps seen previously. When you do the binding by calling `CreateIoCompletionPort`, you must cast the `SOCKET` to a `HANDLE` and pass it as the first `FileHandle` argument.

.NET Framework Asynchronous I/O

Asynchronous I/O in .NET is much simpler than in Win32. Just measuring by page count alone, the coverage of managed asynchronous I/O is only a fraction of Win32's. That's because it is entirely based on the asynchronous programming model (APM) that we already reviewed in Chapter 8, Asynchronous Programming Models. This simplicity, on the other hand, means that you'll have vastly less control over the way that I/O is initiated and the way completions happen. This turns out to be one of the few reasons some programmers continue using native code in heavily loaded server programs, such as Web, application, media, and file servers; this additional control can sometimes be used to achieve better throughput. That said, .NET's approach is just right for most developers.

Asynchronous Device (File) I/O

The primary way to achieve asynchronous I/O in .NET is via the `System.IO.Stream` abstract base class. Concrete subclasses like `System.IO.FileStream` and `System.IO.Pipes.Pipestream` override its `BeginRead`,

`EndRead`, `BeginWrite`, and `EndWrite` asynchronous APIs to provide device specific implementations. (Sockets are a separate topic altogether and we will review them shortly.) The completion techniques are the same as those for any `IAsyncResult` APM-based API.

The `System.IO.Stream` class provides four asynchronous methods of interest.

```
public virtual IAsyncResult BeginRead(
    byte[] buffer,
    int offset,
    int count,
    AsyncCallback callback,
    object state
);
public virtual int EndRead(IAsyncResult asyncResult);
public virtual IAsyncResult BeginWrite(
    byte[] buffer,
    int offset,
    int count,
    AsyncCallback callback,
    object state
);
public virtual void EndWrite(IAsyncResult asyncResult);
```

These are used to initiate asynchronous I/O requests. The basic implementations provided by `Stream` are not very interesting, however. They are there so `Stream` implementations for devices that don't natively support asynchronous I/O needn't implement anything special. The default implementation queues thread pool callbacks that `Read` and `Write`, respectively. These are virtual methods, however, so for `Streams` that do support asynchronous I/O, it is quite easy to override this behavior. That's what `FileStream` and `PipeStream` do.

As with `CreateFile`, you must specify at creation time that you'd like to use a `FileStream` for asynchronous execution. With `FileStream`, you do this by passing `true` as the `isAsync` argument to the constructor overloads, which accept it. The stream's `IsAsync` property will subsequently return `true`. If you fail to pass this value, calls to `BeginRead` and `BeginWrite` will succeed. But they will use the base class implementation from `Stream`, which provides none of the benefits of true asynchronous file I/O.

Similarly, when you construct a named pipe stream, you must specify that you'd like to use it for asynchronous execution. Otherwise, the resulting

stream will just use Stream's implementations. Since PipeStream is an abstract class, you'll do this when instantiating one of its concrete subclasses, NamedPipeClientStream or NamedPipeServerStream. Unlike FileStream, which uses a bool, there are overloads that accept a PipeOptions enum value. This enum type supports an Asynchronous value. After constructing a pipe stream in this manner, its IsAsync property will return true.

When constructing these kinds of streams for asynchronous execution, in addition to opening the underlying HANDLE for overlapped I/O, the constructors use ThreadPool.BindHandle to register the HANDLE for I/O completion port completion. For simplicity's sake, the .NET libraries always use an I/O completion port callback; even if you end up waiting on the event returned in the IAsyncResult, setting the event requires an internal callback to be run. This is an implementation detail, but is not always optimal. For those that keep a close eye on performance, where details like this matter, this is worth knowing.

Once you've constructed a stream capable of asynchronous I/O, you can then use its BeginRead, EndRead, BeginWrite, and EndWrite APIs. You can pass an AsyncCallback, poll the IAsyncResult's IsCompleted flag, wait on the resulting event, and so forth. It should now be a little more apparent why IAsyncResult has the strange CompletedSynchronously flag. When set, it means the device I/O completed synchronously (as described earlier) and the callback was invoked on the thread that called BeginRead (or BeginWrite). If you were to keep issuing new calls to asynchronous I/O inside the completion callbacks, you could end up using a lot of stack. The CompletedSynchronously flag can, thus, be used to stop the recursion and avoid stack overflow.

There is a special API for named pipes that supports asynchronous execution. The NamedPipeServerStream allows waiting for a new connection asynchronously, using the BeginWaitForConnection and EndWaitForConnection pair of methods.

```
public unsafe IAsyncResult BeginWaitForConnection(
    AsyncCallback callback,
    object state
);
public unsafe void EndWaitForConnection(IAsyncResult asyncResult);
```

These internally use the ConnectNamedPipe Win32 API shown earlier.

Asynchronous Sockets I/O

The `System.Net.Sockets` library supports asynchronous sockets I/O, just as the native Winsock APIs do (as we saw earlier). The basic usage that has been around since .NET 1.0 is straightforward and looks almost identical to the APM based stream APIs we've seen. Along with .NET 3.5, however, comes a new way of performing asynchronous sockets I/O that allows finer-grained control over the number of asynchronous objects that are created. This is useful for high performance situations and is akin to the way pooling overlapped objects (in native code) can be used to lead to performance improvements.

Let's first look at the classic APM approach. Many of `Socket`'s functions, such as accepting, reading, and writing, have corresponding APM versions that start with `Begin` and `End`. Unlike file I/O, you needn't specify when constructing the `Socket` that you want to use it for asynchronous execution; the class internally ensures that it is bound to an I/O completion port by the time you issue an asynchronous request. You can, however, enforce that only asynchronous operations are used for a particular `Socket` by giving a `SocketInformation` object at construction time with the `SocketInformationOptions.NonBlocking` setting. Because there are so many `Begin/End` methods and overloads on `Socket`, we will only list them by name: `BeginAccept`, `BeginConnect`, `BeginDisconnect`, `BeginRecieve`, `BeginRecieveFrom`, `BeginRecieveMessageFrom`, `BeginSend`, `BeginSendFile`, and `BeginSendTo`.

The `NetworkStream` class also implements the `BeginRead`, `EndRead`, `BeginWrite`, and `EndWrite` methods to use the true asynchronous I/O capabilities of the `Socket` class.

The new pattern introduced in .NET 3.5 brings about a `SocketAsyncEventArgs` class. Each instance of this class represents a possible in-flight asynchronous operation. This was added so that programs can pool and manage these objects much as they would overlapped objects and buffers, minimizing overhead caused per operation by the APM based methods, that is, due to the `IAsyncResult` object allocations and associated state. This provides finer-grained control over the resource usage on highly scalable servers, but comes at a cost: it is entirely up to the application to manage the lifetime of `SocketAsyncEventArgs`, and the API is slightly less convenient to use than the APM methods.

To use this method, you must first allocate an instance of `SocketAsyncEventArgs`.

```
public class SocketAsyncEventArgs : EventArgs, IDisposable
{
    public SocketAsyncEventArgs();

    public event EventHandler<SocketAsyncEventArgs> Completed;

    public void Dispose();
    public void SetBuffer(int offset, int count);
    public void SetBuffer(byte[] buffer, int offset, int count);

    public Socket AcceptSocket { get; set; }
    public byte[] Buffer { get; }
    public IList<ArraySegment<byte>> BufferList { get; set; }
    public int BytesTransferred { get; }
    public int Count { get; }
    public bool DisconnectReuseSocket { get; set; }
    public SocketAsyncOperation LastOperation { get; }
    public int Offset { get; }
    public IPPacketInformation ReceiveMessageFromPacketInfo { get; }
    public EndPoint RemoteEndPoint { get; set; }
    public SendPacketsElement[] SendPacketsElements { get; set; }
    public TransmitFileOptions SendPacketsFlags { get; set; }
    public int SendPacketsSendSize { get; set; }
    public SocketError SocketError { get; set; }
    public SocketFlags SocketFlags { get; set; }
    public object UserToken { get; set; }
}
```

Once you have an instance you'd like to use for an operation, you will want to call the `SetBuffer` method to register the `byte[]` you will use for sends and receives and set the `Completed` event handler to contain a delegate referencing the callback to run upon completion. This callback is the standard `EventHandler<T>` delegate type. Other useful properties can be set, such as `UserToken`, which allows you to flow application state from the point of initiating the asynchronous operation and the callback itself. Some of the APIs require certain properties to have been set and will manipulate them in interesting ways.

Next, you will use an initialized `SocketAsyncEventArgs` to start an asynchronous network operation. This is done with the various `XxAsync` methods on the socket class.

```
public bool ConnectAsync(SocketAsyncEventArgs e);
public bool DisconnectAsync(SocketAsyncEventArgs e);
public bool ReceiveAsync(SocketAsyncEventArgs e);
public bool ReceiveFromAsync(SocketAsyncEventArgs e);
public bool ReceiveMessageFromAsync(SocketAsyncEventArgs e);
public bool SendAsync(SocketAsyncEventArgs e);
public bool SendPacketsAsync(SocketAsyncEventArgs e);
```

All of these methods return a `bool` value, which *must* be checked. If `true` is returned, it means the operation is happening asynchronously and the callback will be invoked when it finishes. The `SocketAsyncEventArgs` passed to the callback will contain the results of the operation. If `false` is returned, however, the operation has completed synchronously. The `SocketAsyncEventArgs` supplied as the argument will contain the results of the computation. The callback will not fire in this case, so the completion activity must be run immediately in the context of the code that initiated the I/O.

The results of an operation are not retrieved with an `End` call, as with the APM, so the `SocketAsyncEventArgs` also serves the purpose of communicating results and errors (in `SocketError`), if any. Consult the SDK documentation for full details about which properties are used by which specific sockets operations.

Most often, there will be a pool of these objects and the completion callback is meant to return them back. But notice that there is also a `Dispose` method to get rid of state in the object (such as overlapped state) once you no longer need a particular instance.

I/O Cancellation

We mentioned several times earlier that when a thread terminates, any outstanding asynchronous IOs that it initiated are automatically canceled by the OS. This capability is also available in user-mode ever before a thread terminates, in case the I/O operations become irrelevant for some application-specific reason. This is **asynchronous I/O cancellation**.

I/O cancellation can also be used to improve application responsiveness. When someone accesses a file over the network through an application's GUI, you might supply a progress indicator to let them know how much time it will take to retrieve. And you might even give them a cancel

button. We saw already in Chapter 13, Data and Task Parallelism, how to cancel CPU-bound activities with polling. But if workers are blocked on I/O, we need another way to interrupt things. As of Windows Vista, this kind of **synchronous I/O cancellation** is supported.

The .NET Framework doesn't currently expose I/O cancellation directly in any way. It turns out that if you P/Invoke to some of the Win32 functions about to be mentioned, the .NET `FileStream` and related classes will at least respond intelligently (by throwing an `OperationCanceledException` from things like `Read` and `Write`). Everything we're about to discuss, however, is limited to native code programming.

Asynchronous I/O Cancellation for the Current Thread

Say we've initiated asynchronous I/O using a non-I/O completion port completion mechanism. If we suddenly lose interest in its results, we could of course just forget about it. In other words, we could just never get around to waiting for it and processing the results, perhaps returning control back to the application. This seems simple enough.

But there are some major drawbacks to this naive approach. Just because we ignored the I/O that was initiated does not mean it has stopped. In fact, it will eventually complete and result in some processing in the kernel. Whatever data structures passed to the `ReadFile` routine are still referenced by the I/O and, when the routine finishes, it might try to write to them. This includes any buffers and `OVERLAPPED` structures involved in the I/O. This writing will race with whatever the program does after "forgetting" about the I/O, and would make it just about impossible to properly free the data structures. If completion is done by an APC, then this APC may get called at some arbitrary point in the program's execution. Or the thread may terminate, automatically canceling the I/O but providing no chance to free the data structures.

The `CancelIo` function allows you to cancel this kind of I/O completely so that racy I/O processing does not happen. This API has been around since the Windows 95 and NT 4.0 days.

```
BOOL WINAPI CancelIo(HANDLE hFile);
```

This API only cancels outstanding I/O issued the `hFile` from the calling thread. If there is no I/O happening asynchronously, or the I/O was

triggered by a separate thread, the call has no effect. Be cautious: when `CancelIo` returns, it only indicates that the I/O has been *marked* for cancellation, not that it *has* been canceled. The I/O will still complete normally, or may have already completed, so there is extra coordination necessary to reclaim resources. All successfully canceled I/Os complete with the error code `ERROR_OPERATION_ABORTED`.

Synchronous I/O Cancellation for Another Thread

Through clever use of asynchronous I/O, we can cancel long running I/O operations that happen synchronously on the current thread. For example, we can begin an asynchronous `ReadFile` and wait on the file handle with `WaitForSingleObject`, using a timeout. If the timeout expires, we can choose to do whatever we please, including responding to a cancellation button, issuing a `CancelIo` in response, and giving back the (previously blocked) thread to the application. The major disadvantage to this approach is that we must choose a somewhat arbitrary timeout interval for checking cancellation. There is a tension between choosing a small interval (to increase responsiveness) and a large interval (to reduce the number of superfluous context switches and number of reissued waits).

Yet another approach is possible. Windows Vista introduces a new `CancelSynchronousIo` function. Calling this on a target thread cancels its current synchronous I/O operation.

```
BOOL WINAPI CancelSynchronousIo(HANDLE hThread);
```

Any synchronous I/O on `hThread` will awaken with the `ERROR_OPERATION_ABORTED` error. Note that this has no effect on asynchronous I/Os that the target thread has issued. (`CancelIoEx` can be used for that; we'll examine that function momentarily.) Also, `CancelSynchronousIo` does not wait for the I/O to be canceled before returning to the caller; it merely marks the I/O for cancellation. The target thread may indeed not even be issuing any I/O at the time a call is made.

While synchronous I/O cancellation appears to be a useful feature, it has many drawbacks.

The first drawback is that it doesn't handle all possible I/O kinds; it only cancels file based operations, including file I/O that is taking place over a

network UNC path. (Canceling network I/O via UNC paths tends to be the most useful capability. It's common to block for seconds when accessing UNC paths, particularly if a server is down, whereas blocking on the local disk is typically measured in micro or milliseconds.) If a thread is blocked on a network socket, however, then you will need to use another mechanism to interrupt a thread. Requests for canceling operations on devices that don't support I/O cancellation will be ignored. Similarly, if a thread is blocked on an event or other kind of synchronization object, you will need to implement your own higher-level cancellation framework to awaken it. Neither Win32 nor .NET currently provide such a unified framework.

But the second drawback is the deal breaker. It's easy to use `CancelSynchronousIo` carelessly and dangerously. Your first inclination might be to misuse it because it's deceptively simple interface says nothing about proper use. For it to be safe, you must ensure that the I/O currently happening on the target thread is *safe* to cancel. If it's running library code that is not expecting the I/O to be canceled, issuing the cancellation could lead to corrupt state. The proper use of this API is to implement synchronization between the code issuing I/O and the code canceling I/O, such that you know specifically which I/O requests will be canceled by calling the API. This typically involves locks and dealing with some tricky race conditions between checking and canceling.

An alternative approach is to use a separate cancellation event that is managed by application code. Whenever cancellation is requested, it is set. Then all code that waits on I/Os must perform wait-any style waits and check upon waking whether it woke because of cancellation. The nice thing is that this same approach can be used for synchronization waits and for devices that don't support cancellation. This is the cleanest approach and is the recommended approach, though it takes up front planning, care, and diligence. Haphazardly calling `CancelSynchronousIo` on random application threads is easier, but the end result will be messy.

Asynchronous I/O Cancellation for Any Thread

There is another technique that can be used to cancel asynchronous I/Os happening on any thread, including the current one. The `CancelIoEx` function takes a different approach than `CancelIo` and runs right up against

many of the same dangerous issues that were just mentioned for synchronous cancellation.

```
BOOL WINAPI CancelIoEx(HANDLE hFile, LPOVERLAPPED lpOverlapped);
```

When invoked on `hFile` with a `NULL` `lpOverlapped`, any outstanding asynchronous I/Os in the process for that particular file will be marked for cancellation. You can also specify a particular `LPOVERLAPPED` structure, which, as you may guess, only cancels those asynchronous I/Os on the target file that pertain to that particular `OVERLAPPED`. If no such I/Os can be found, the function returns `FALSE` and `GetLastError` will return `ERROR_NOT_FOUND`.

Where Are We?

This chapter provided an overview of some of the most important I/O capabilities supported by the Windows OS, with a particular eye on concurrent programming. The most important capability is true first-class support for asynchronous I/O, enabling a device to execute an I/O operation fully asynchronously without needing an OS thread blocked waiting for completion. This takes advantage of the natural asynchrony in the hardware. For highly concurrent programs—particularly server applications with high throughput demands—this can offer a substantial boost to scalability and reduction in memory usage. We saw that files, pipes, and sockets, specifically, each support slightly different variants on the same idea.

We concluded with a look at how to cancel runaway I/O operations whose results are no longer needed. And this was a convenient way to end the chapter. Next we will focus on graphical user interfaces (GUIs) on Windows. Building a responsive GUI almost always involves some kind of interaction with asynchronous I/O, and it is becoming increasingly necessary for applications to provide cancellation capabilities. With that, let's turn to GUIs.

FURTHER READING

- A. Jones, A. Deshpande. Windows Sockets 2.0: Write Scalable Winsock Apps Using Completion Ports. *MSDN Magazine* (2000).
- G. Maffeo, A. Sliwowicz. Win32 I/O Cancellation Support in Windows Vista. *MSDN Developer Center Article* (2005).

Microsoft. I/O Prioritization in Windows Vista: Recommendations for Application, Driver, and Device Developers for Supporting I/O Prioritization in Windows Vista. *Microsoft.com Whitepaper* (2006).

J. Richter, J. D. Clark. *Programming Server-Side Applications for Windows 2000* (MS Press, 2000).

M. Russinovich. Windows Administration: Inside the Windows Vista Kernel: Part 1. *Microsoft TechNet Magazine* (2007).

16

Graphical User Interfaces

GRAPHICAL USER INTERFACES (GUIs) are of special interest to developers writing concurrent programs. Due to the same shared message loop oriented architecture that all Windows GUI frameworks use, concurrency is often an unavoidable necessity to deliver a responsive experience. The reason is subtle. Each window has a special GUI thread whose job is to process messages in its own dedicated message queue. This entails responding to button clicks, repainting the screen, and the like, usually by running application specific event handlers. All events are processed sequentially, one after the other. Code on this thread must be written with great care, however, because any blocking due to I/O or synchronization activity will delay processing the window's messages. If an event handler is called in response to button click, for example, and it loads a file over the network, the application is apt to freeze up while it loads.

To prevent these kinds of problems, high latency and computationally intensive work should never happen on the GUI thread. To make matters slightly more complicated, most GUI frameworks also require that code is running on the GUI thread in order to update UI widgets. This means that even if you manage to marshal work off the thread, you'll need to get back onto it later. Accomplishing all this requires a bit of knowledge about how threading works, and, of course, the various ways in which interthread communication can be implemented. There are many facilities meant specifically to make this easier, particularly in the .NET Framework.

In this chapter, we'll review the GUI threading architecture broadly and then look at specifically how it is surfaced in Windows Forms and the Windows Presentation Foundation (WPF). We'll also look at the mechanisms available for building responsive GUIs, including the `.NET SynchronizationContext`, which unifies GUI threading models on .NET, and the `AsyncOperationManager`, which builds atop the `SynchronizationContext` feature to simplify building higher-level services. Asynchronous patterns like the event driven asynchronous programming model reviewed in Chapter 8, Asynchronous Programming Models, commonly use these features in their implementation.

GUI Threading Models

GUI architectures on Windows have remained fairly consistent for the past two decades. Although there are differences in the details—and in the capabilities and style of programming—USER32, Windows Forms, and WPF all use the same general architecture for reacting to user input and repainting the screen. That architecture can be summed up as “single threaded and message based.” Just a few lines of Petzold style code can be used to succinctly illustrate it.

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

This is called the message loop or, alternatively, the **message pump**. We already had some exposure to this concept during the course of discussing functions like `MsgWaitForMultipleObjectsEx` in Chapter 5, Windows Kernel Synchronization. Notice, however, that this loop is sequential. One `GetMessage` call happens after the other.

To understand the message loop, you need to first understand how GUIs on Windows work. Figure 16.1 illustrates the basic architecture. Each thread that creates at least one window has a message queue, and it is this thread's job to process messages from the queue. The thread is silently

given this responsibility whenever something like `CreateWindow` (USER32) or `Application.Run` (Windows Forms and WPF) is called in an application. All subsequent GUI events, such as user initiated events (e.g., clicks, key-strokes, window close requests, scrollbar dragging), system initiated events (e.g., repainting and resizing), and application specific events for custom components, are processed by posting messages to this hidden message queue.

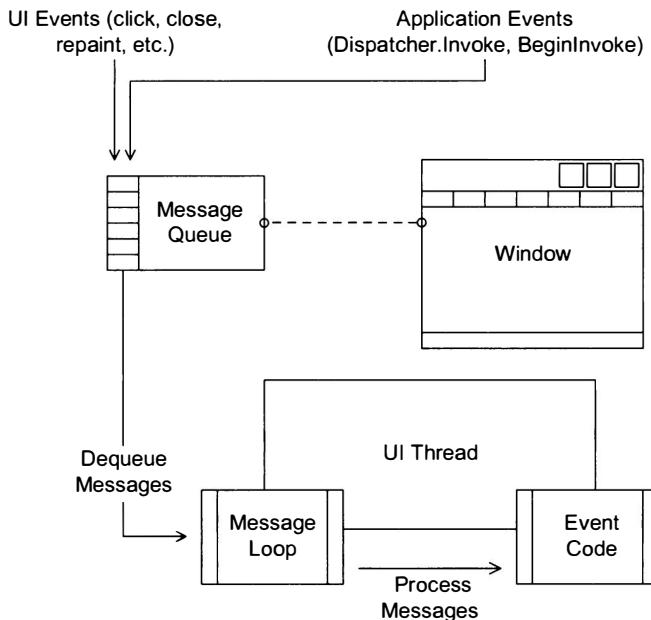


FIGURE 16.1: GUIs on Windows

The days of hand writing USER32 message loops have long passed. Windows Forms and WPF contain their own message loops so that you needn't worry about it. When you call Windows Forms' `Application.Run` method, not only are windows created, but the `Run` method continuously runs the message loop until the program exits. This message loop invokes a **window procedure** that is in turn responsible for processing messages. What this means is that after the call to `Run` a large portion of the work that subsequently happens (if not all of it) is generated by event handlers run in response to GUI events.

There is another aspect to the GUI architecture that is interesting, relevant, and somewhat unfortunate. Any code that directly manipulates GUI elements *must* execute on the GUI thread. Given that we already established the GUI thread's sole purpose is to process messages, repaint the screen, and the like, you might wonder how this is even possible. The answer is that such code runs inside of event handlers that are invoked in response to GUI events. By invoking event handlers on the GUI thread, some complex issues are avoided—such as requiring developers to acquire locks while updating the GUI—in an attempt to provide a more convenient programming model. Additionally, if events could be processed entirely asynchronously, strange glitches could occur due to interleaving multiple handlers and/or the framework deciding to repaint while a handler was in progress.

Given a window handle (`HWND`), you can easily find out the identity of its special thread.

```
DWORD GetWindowThreadProcessId(HWND hWnd, LPDWORD lpdwProcessId);
```

Why is this design choice an issue? Anything the GUI thread does in addition to dequeuing and dispatching messages from its queue prevents it from processing additional messages. If the thread is running a user supplied event handler and that event handler does some lengthy operation (such as a network I/O), subsequent messages will get clogged in the queue waiting for the GUI thread to return to its message loop. The fact that code inside of event handlers automatically runs on the GUI thread leads developers down this path by default, often without knowing it.

Let's take a simple example. In response to a button click, let's say that your application fires off a network request to download a file. It does this on the GUI thread. Now imagine that this could take some time, maybe 1 second. The application is frozen and cannot repaint for 1 second. If the user of your application tries to resize, close, or maximize the window, for example, they will see visual artifacts (such as a blank white screen) because the GUI thread can't retrieve those messages and properly repaint. But 1 second is a fairly brief delay; it will be slightly annoying, but not terrible. But now imagine that the network connection drops out, and instead of 1 second, we must wait for a 30 second network timeout to occur. What an awful user

experience. The Windows shell will slap a Not Responding onto the end of your application's title bar, and a user is apt to need to resort to killing the process in Task Manager unless they are incredibly patient.

This fact is also the motivation for things such as `MsgWaitForMultipleObjects`, which runs the message loop while a thread waits. You have far less control over this in managed code. And reentrancy is often a tricky issue anyway; for example, how can you be sure that when an event handler blocks, it is actually safe to dispatch other arbitrary GUI events in response to messages? A better solution is to architect your program so that the only code running on the GUI thread is actively manipulating controls. Any data or computations required to update the GUI in an appropriate way should be done elsewhere and not on the GUI thread. Typically that means offloading work to the thread pool and then marshaling results back when they are available. There are several facilities available to run callbacks back on the GUI thread in this manner. We'll explore the Windows Forms and WPF specific ones in addition to some more generic features like `SynchronizationContext` and `BackgroundWorker` later.

Finally, you might wonder why an apparently flawed, single threaded architecture has persisted for many years. The main reason is that providing anything else is incredibly difficult. Single threaded is simple. One of WPF's original goals was to replace this architecture with a so-called rental model (see Further Reading, Anderson). But due to numerous issues around compatibility, performance, glitches, and user education around threading in general, this plan was eventually abandoned (see Further Reading, Kramer). In summary, synchronization was suddenly thrust into the forefront of development of GUI applications, and yet most such developers aren't completely familiar with the associated concurrency issues. The result would have been misuse, possibly resulting in a worse set of issues than the single threaded GUI problems. Perhaps some new GUI framework in the future will undertake the goal of doing away with the single threaded GUI architecture, but with today's technologies we must cope.

Single Threaded Apartments (STAs)

You'll see the term single threaded apartment (STA) used to describe GUI and COM architectures alike. The term is informal, and comes from COM's

threading models. Understanding a bit about COM's threading models and how they relate to GUI programming will come in very handy, so we'll spend a moment reviewing them.

- **Single Thread Apartment (STA).** There is a single thread that runs in a given STA, and there can be any number of STAs in a particular process. Any apartment threaded COM objects created by code running on this thread are **affine** to it. This is similar to the way in which GUI controls are affine to the single GUI thread owning the windows on which those controls reside. In fact, a lot of the plumbing beneath STAs explicitly uses the same windows message queue mechanisms. Each STA has a hidden USER32 window, and when a cross apartment call is made, it results in a new GUI message. Each STA thread must therefore run a message pump in order to dispatch these messages. Failure to pump can lead to deadlocks rather than responsiveness issues, meaning `MsgWaitForMultipleObjects` is even more important in COM programs.
- **Multithreaded Apartment (MTA).** Any number of threads can run in the MTA, and there is only one of them per process. All threads not affinitized to a STA effectively run free threaded in the MTA with free access to all of the COM objects within. Any interaction between threads in an MTA and objects in the STA is regulated by sending messages between apartment threads. Apartment-threaded objects created inside the MTA are also affine to it, but since there are multiple threads in the MTA, this only means STA to MTA access must be regulated in a similar fashion.
- **Neutral Apartment (NA).** This kind of apartment has no threads associated with it. Free threaded COM objects live here and require no marshaling to access.

There are myriad other interesting COM synchronization concepts, but they are beyond the scope of this book (see Further Reading, Box; Grimes).

You may be wondering what dictates whether an STA, MTA, or NA is used. In native code, you just use the `CoInitialize` and `CoInitializeEx` COM functions to join a thread with a particular kind of apartment.

```
HRESULT CoInitialize(void * pvReserved);
HRESULT CoInitializeEx(void * pvReserved, DWORD dwCoInit);
```

`CoInitialize` joins the STA, and the `dwCoInit` parameter to `CoInitializeEx` can be used to specify `COINIT_APARTMENTTHREADED` (STA) or `COINIT_MULTITHREADED`.

In the .NET Framework, you can use the `Thread.SetApartmentState` function to achieve the same thing (and `Thread.GetApartmentState` to query for the current status).

```
public void SetApartmentState(ApartmentState state);
public ApartmentState GetApartmentState();
```

These APIs deal in terms of the `ApartmentState` enum.

```
public enum ApartmentState
{
    STA,
    MTA,
    Unknown
}
```

There is an inherent race condition if multiple components wish to join an apartment, so it is common protocol to ensure only one component per thread takes the responsibility for joining.

There are also two attributes available in the .NET Framework: `STAThreadAttribute` and `MTAThreadAttribute`. When applied to the entry point for a program, the CLR will ensure the resulting thread joins the correct apartment. For example:

```
class MyProgram
{
    [STAThread]
    public static void Main(string[] args)
    {
        /*... running in the STA ...*/
    }
}
```

These are interesting because there's been a historically close relationship between things like OLE32 and USER32. For example, the Windows clipboard uses OLE32 and is often used from GUI programs. They have a symbiotic relationship. This, combined with the similarities in threading

models, means that you'll frequently see STAs and GUIs mentioned together. The two can strictly be teased apart, but most developers can reasonably consider them to be the same abstraction. This is the reason you'll see that the Visual Studio project templates for Windows Forms and WPF automatically tack a `STAThreadAttribute` onto your project's entry point.

Responsiveness: What Is It, Anyway?

The term **responsiveness** is familiar to most developers, at least intuitively. A responsive application is one that responds to input promptly and doesn't leave the user hanging (pun intended). The perceived response time is the delay between the initiation of an action and the results of that action being readily available to the user. Often the results entail the full set of computations that are to occur as a result, but responsiveness can be vastly improved for long running computations by providing an early acknowledgement and optionally progress updating.

Predictability is also an important quality of responsive GUIs. This is one of the reasons the network scenario given previously is so terrible. The program will work just fine under most circumstances, but occasionally hangs. I'm sure you've had this experience before, and it's not a pleasant one. And one of the most frustrating aspects is the sheer unpredictability of when it will happen. If clicking a certain button always causes a 5-minute hang, the user would know to avoid clicking it or at least use the opportunity to grab a coffee. And it would be more likely to show up and get fixed while initially testing the application.

Most poorly responding applications are a result of developers not understanding the GUI threading architecture. A distant second is common in large organizations: developers provide reusable libraries that may block under some circumstances, and those libraries are then used by developers working on the GUI components without realizing the potential for blocking. There is often expensive processing that needs to be done in response to GUI events, but it's the responsibility of an application developer to identify the cost and appropriately decide to arrange for that work to happen in a way that still provides a great user experience. How you actually go about that is what the rest of this chapter is all about.

.NET Asynchronous GUI Features

To implement the aforementioned ideas, you need to know how to marshal work between threads. By now, it should be evident how to get work *off* the thread by using one of the many asynchronous .NET APIs or by calling the thread pool directly. The next obvious question is how you are supposed to get work back *onto* the GUI thread to update the visuals. Both Windows Forms and WPF give you specific mechanisms to marshal work onto the GUI thread. Although similar, they offer different abstractions for this purpose.

Despite the different APIs available for Windows Forms and WPF, there is a common infrastructure beneath it all. This hinges on `SynchronizationContext`, which is an abstract representation for thread and synchronization affinity of the kind that GUI frameworks employ (as well as COM STAs). On top of this, `AsynchronousOperationManager` provides some simple abstractions to make it easier to manage the lifetime of individual `AsynchronousOperations` that use said contexts. Finally, a convenient codification of using said things to build responsive GUIs—including cancellation and progress reporting—is available in the form of the `BackgroundWorker` component.

These common abstractions are useful for several purposes. First, by providing common infrastructure, reusable libraries can be developed that expose asynchronous operations that will work across GUI frameworks. Second, having commonalities makes transitioning between frameworks easier. Many developers need to use multiple frameworks for different purposes or, at the very least, welcome not having to learn completely new APIs to accomplish the same functionality between different frameworks.

.NET GUI Frameworks

Let's take a look at the framework specific APIs to interact with the GUI thread. After this we will look at the common infrastructure that ties them all together.

Windows Forms

The way Windows Forms surfaces the ability to marshal back to the GUI thread should look very familiar to you. It is largely based on the APM

that was discussed back in Chapter 8, Asynchronous Programming Models.

Marshaling Calls with ISynchronizeInvoke. Callbacks are represented using delegates. Assuming you are triggering the callback on an external (non-GUI) thread, you must decide whether to block waiting for the callback to finish running (synchronous) or instead simply queue the callback to run at some point in the future (asynchronous). The Windows Forms APIs support both.

The support is provided through the `ISynchronizeInvoke` interface, in the `System.ComponentModel` namespace. The `System.Windows.Forms.Control` class implements this interface, so all controls inherit these capabilities automatically.

```
public interface ISynchronizeInvoke
{
    IAsyncResult BeginInvoke(Delegate method, object[] args);
    object EndInvoke(IAsyncResult result);
    object Invoke(Delegate method, object[] args);
    bool InvokeRequired { get; }
}
```

In addition to those methods, the `Control` class also provides some convenience methods.

```
public IAsyncResult BeginInvoke(Delegate method);
public object Invoke(Delegate method);
```

These are used for methods that don't require any arguments.

Notice that `BeginInvoke` and `EndInvoke` are reminiscent of the APM. In fact, they follow the same programming model except that they've been written to be general purpose. You provide any kind of `Delegate` as the method argument for `BeginInvoke`, and the arguments to it are captured in the untyped `object[] args` parameter. The `BeginInvoke` method will marshal the delegate over to the GUI thread owning the window to which the target control belongs (internally using the Win32 `PostMessage` API), an `IAsyncResult` is returned as a handle to the result, and the results will be made available through the `EndInvoke` method. The implementation lazily allocates the kernel event object so to avoid unnecessary resource allocation overhead. Calling `EndInvoke` will block until the callback finishes running.

Alternatively, you can call the `Invoke` method to run the code synchronously, which is effectively equivalent to saying `BeginInvoke` immediately followed by `EndInvoke`.

Each of these mechanisms automatically captures and flows the `ExecutionContext`. You can use `ExecutionContext`'s `SuppressFlow` method to prevent the context from flowing across threads. For full trust applications that needn't worry about security problems like elevation of privilege, this can provide some efficiency gains.

Identifying Calls that Need to Marshal. If you're already running code on the GUI thread, marshaling is unnecessary. In fact, if `Control`'s implementation wasn't intelligent enough, doing a synchronous `Invoke` from the GUI thread could lead to deadlock. Thankfully it detects these cases for both `BeginInvoke` and `Invoke` and runs the callback inline without interacting with the message queue. You can check this yourself by reading the `InvokeRequired` property. It returns `false` if you are already running code on the GUI thread associated with the target `Control`. A return of `true` means you should use either `Invoke` or `BeginInvoke` to transfer control before calling a method. This is implemented using the `GetWindowThreadProcessId` method we reviewed earlier.

In addition to all of the features for marshaling work between threads, Windows Forms 2.0 has introduced automatic checking to guard against illegal cross thread GUI control accesses. Prior to 2.0, accesses may succeed or fail somewhat sporadically. It depends on race conditions and the nature of the particular API in question. As of 2.0, however, Windows Forms will behave differently when run under a debugger. Most accesses to controls will throw an `InvalidOperationException` with an error message of "Control <X> accessed from a thread other than the thread it was created on." This will not be thrown (in some cases) for deployed applications because the runtime checks can be costly; when a debugger is attached, however, it will always be thrown. To disable this check (for compatibility reasons), you can set a control's `CheckForIllegalCrossThreadCalls` property to `false`.

Running the Message Loop Mid-stack. Occasionally, a GUI thread will do something that means it can't run its message loop for an extended period of time. This is common when showing a modal dialog such as the `OpenFileDialog` in Windows Forms where the call to `ShowDialog` blocks until a

user selection has been made. It may also be common if some lengthy computation must occur on the GUI thread, such as manipulating a large number of controls.

The `CommonDialog` from which things like `OpenFileDialog` derive automatically runs the so-called modal message loop when `ShowDialog` is waiting. This ensures that GUI messages are processed, so that, for example, the window can still repaint while the modal dialog is moved around on top of it, among other things. You can also run the message loop explicitly in your program with a call to the `Application.DoEvents` static method. Doing so processes all of the messages currently in the window's queue and then returns. Notice that this is explicitly allowing reentrancy because event handlers may run on the current thread. It's fairly common for this API to be misused; one common example is to mask improperly written code that should have marshaled work to a separate thread (as noted above). Be on the lookout for this.

Windows Presentation Foundation (WPF)

Just as Windows Forms provides a consistent way of marshaling work to the GUI thread via `ISynchronizeInvoke`, WPF also provides a way of doing this across all controls, albeit in its own (different) way. All visual types in WPF derive directly or indirectly from the same base class, `System.Windows.Threading.DispatcherObject`. This and related classes offer support for marshaling between threads. Many visual types also extend the `System.Threading.Freezable` base class (which inherits from `DependencyObject`, which inherits from `DispatcherObject`), providing dynamic immutability. A frozen object may be safely shared among threads without worry that concurrent updates will be observed.

The DispatcherObject and Dispatcher Classes. The `DispatcherObject` class itself is small and simple.

```
public class DispatcherObject
{
    protected DispatcherObject();

    public bool CheckAccess();
    public void VerifyAccess();

    public Dispatcher Dispatcher { get; }
}
```

Because all visual classes in WPF derive from `DispatcherObject`, they all have these same instance members.

The `CheckAccess` and `VerifyAccess` methods are meant to determine whether the calling thread may freely manipulate the target control. A return value of `true` from `CheckAccess` means that code is already running on the GUI thread, whereas a return value of `false` means marshaling is required. Similarly, `VerifyAccess` just returns if code is already running on the GUI thread but throws an `InvalidOperationException` otherwise. This method is used throughout WPF to verify that properties and methods are only accessed from the proper thread. If you try, you'll see this exception.

Each WPF GUI thread has a single `System.Windows.Threading.Dispatcher` associated with it. Once you've retrieved a reference to one, either by the `Dispatcher` property on a specific `DispatcherObject` or by calling the `CurrentDispatcher` static property on `Dispatcher` itself, you can use it to marshal calls to the GUI thread.

The `Dispatcher` class is fairly feature rich when compared to Windows Forms.

```
public sealed class Dispatcher
{
    // Methods
    public DispatcherOperation BeginInvoke(
        DispatcherPriority priority,
        Delegate method
    );
    public DispatcherOperation BeginInvoke(
        DispatcherPriority priority,
        Delegate method,
        object arg
    );
    public DispatcherOperation BeginInvoke(
        DispatcherPriority priority,
        Delegate method,
        object arg,
        params object[] args
    );

    public object Invoke(
        DispatcherPriority priority,
        Delegate method
    );
    public object Invoke(
        DispatcherPriority priority,
```

```
        Delegate method,
        object arg
    );
    public object Invoke(
        DispatcherPriority priority,
        Delegate method,
        object arg,
        params object[] args
    );
    public object Invoke(
        DispatcherPriority priority,
        TimeSpan timeout,
        Delegate method
    );
    public object Invoke(
        DispatcherPriority priority,
        TimeSpan timeout,
        Delegate method,
        object arg
    );
    public object Invoke(
        DispatcherPriority priority,
        TimeSpan timeout,
        Delegate method,
        object arg,
        params object[] args
    );
}

public bool CheckAccess();
public void VerifyAccess();

public void BeginInvokeShutdown(DispatcherPriority priority);
public void InvokeShutdown();

public DispatcherProcessingDisabled DisableProcessing();
public void PushFrame(DispatcherFrame frame);

// Static Methods
public static void ExitAllFrames();
public static Dispatcher FromThread(Thread thread);
public static void Run();
public static void ValidatePriority(
    DispatcherPriority priority,
    string parameterName
);

// Properties
public static Dispatcher CurrentDispatcher { get; }
public bool HasShutdownFinished { get; }
public bool HasShutdownStarted { get; }
```

```
public DispatcherHooks Hooks { get; }
public Thread Thread { get; }

// Events
public event EventHandler ShutdownFinished;
public event EventHandler ShutdownStarted;
public event DispatcherUnhandledExceptionEventHandler
    UnhandledException;
public event DispatcherUnhandledExceptionFilterEventHandler
    UnhandledExceptionFilter;
}
```

Here is an overview of some of **Dispatcher**'s features.

- **BeginInvoke** and **Invoke** are meant for marshaling work.
- **CheckAccess** and **VerifyAccess** are equivalent in behavior to the like named methods found on **DispatcherObject**.
- You can get the CLR **Thread** object from a given **Dispatcher** via the **Thread** property and vice versa with the **FromThread** method.
- The **DispatcherHooks** class, available via the **Hooks** property, provides several events that you can use to get notified when new operations are posted to a particular **Dispatcher**.
- You can also shutdown the **Dispatcher** so that it will no longer process events with the **InvokeShutdown** and **BeginInvokeShutdown** methods. The **HasShutdownStarted** and **HasShutdownFinished** properties can be used to inquire about pending shutdowns, and **ShutdownStarted** and **ShutdownFinished** can be used to hook these events. Note that when a **Dispatcher** is shutdown, pending messages in its queue are dropped.
- The **UnhandledException** and **UnhandledExceptionFilter** events allow you to trap exceptions coming from messages run in the target **Dispatcher**. They even enable you to “catch” them, even if they were technically unhandled in the callback code itself. This is more useful for logging kinds of scenarios.

Though quite useful, most of the features available on **Dispatcher** are for very advanced scenarios. We will turn our attention to one of them: marshaling callbacks to the GUI thread and synchronizing with their completion using the **BeginInvoke** and **Invoke** methods.

Marshaling Calls with Dispatcher. Having just reviewed Windows Forms' support for marshaling work to the GUI thread, the usage of the `BeginInvoke` and `Invoke` methods is probably obvious. There are some interesting differences, however, in addition to some useful new features lacking in the Windows Forms model.

The `BeginInvoke` method enqueues any kind of `Delegate` callback for execution in the target `Dispatcher`'s message queue. It then returns a `DispatcherOperation` object that can be used to interact with the pending operation, including waiting on it. Notice that this object takes the place of an `EndInvoke` method.

The `Invoke` method executes the callback synchronously on the GUI thread. In other words, it also enqueues the method to run in the target `Dispatcher`, but then goes ahead and blocks waiting for it to return. There are also overloads of `Invoke` that accept a timeout argument in the form of a `TimeSpan`. If it is exceeded before the operation has finished running, a value of `null` will be returned. This works by internally waiting on the `DispatcherOperation` that is created.

Although it's not obvious from the signatures, the implementation dynamically checks to see if the `Delegate` you supply is of the `DispatcherOperationCallback` kind.

```
public delegate object DispatcherOperationCallback(object arg);
```

If not, it then goes ahead and dynamically checks to see if your callback has a return value. In either case, it will make the returned object available to you. In the `Invoke` method, it is simply conveyed as the return value itself. With `BeginInvoke`, it will get stored in the `Result` property on the returned `DispatcherOperation` object.

`BeginInvoke` and `Invoke` automatically capture and flow the `ExecutionContext`. As with Windows Forms, you can suppress flowing with `ExecutionContext`'s `SuppressFlow` method, so long as you are running in full trust.

If you're going to use `BeginInvoke` at all, you'll want to do things with the `DispatcherOperation`. We've already seen one reason. This class is a lot like an `IAsyncResult` in its capabilities, but exposes them in very different ways.

```
public class DispatcherOperation
{
    // Methods
    public bool Abort();
    public DispatcherOperationStatus Wait();
    public DispatcherOperationStatus Wait(TimeSpan timeout);

    // Properties
    public Dispatcher Dispatcher { get; }
    public DispatcherPriority Priority { get; }
    public object Result { get; }
    public DispatcherOperationStatus Status { get; }

    // Events
    public event EventHandler Aborted;
    public event EventHandler Completed;
}
```

Using this class, you can `Abort` the operation, which prevents it from running if it has not yet been started. It returns `true` to indicate success, or `false` if the operation already began. Other priorities allow you to query about certain aspects of the operation. For example, `Dispatcher` and `Priority` retrieve information about how it was created.

The most commonly used aspect of the `DispatcherOperation` class is the `Wait` method. It waits for the operation to finish running and then returns. This is how the synchronous `Invoke` method is implemented internally and, as you can see, there is an overload that accepts a `TimeSpan` timeout. Both overloads return a `DispatcherOperationStatus` enum value indicating the current status of the operation. The `Status` property also allows you to query the current status anytime without needing to wait.

```
public enum DispatcherOperationStatus
{
    Pending,
    Aborted,
    Completed,
    Executing
}
```

The `Pending` status means that the operation has been placed into the dispatcher's queue, but has not yet begun running. `Executing`, on the other

hand, means that it is actively running. The two final states, `Aborted` and `Completed`, indicate whether the operation was aborted before running or whether it has finished successfully, respectively.

`DispatcherOperation` also provides `Aborted` and `Completed` event handlers. As you might imagine, these are fired when the respective completion occurs. No guarantees are made about where specifically they are run. `Abort`, for example, runs them synchronously before returning, which may or may not be on the GUI thread itself.

There is one last thing to do with `BeginInvoke` and `Invoke` that we skipped. You can specify a `DispatcherPriority` for any work item, which allows you to rank items among each other. The `Dispatcher` internally maintains a priority queue data structure containing all of the callbacks that must be run at any given time. When selecting the next callback to dispatch, it will prefer those with higher priority.

```
public enum DispatcherPriority
{
    Invalid = -1,
    Inactive = 0,
    SystemIdle = 1,
    ApplicationIdle = 2,
    ContextIdle = 3,
    Background = 4,
    Input = 5,
    Loaded = 6,
    Render = 7,
    DataBind = 8,
    Normal = 9,
    Send = 10
}
```

The first thing that's strikingly obvious is that these aren't your typical priorities. They are declaratively named. This is because WPF itself internally uses priorities extensively for GUI events. For example, when user input is available for processing, such as a button click, the message is enqueued at priority `Input`; when repainting the screen is necessary, it happens at `Render` priority; changes in data that require refreshing the display uses `DataBind`; and so on. This capability allows you to step aside if you don't wish to interfere with certain kinds of responsiveness events or to get ahead of them if you believe your work item is of higher priority.

Synchronization Contexts

The `System.Threading.SynchronizationContext` class is a common abstraction of a synchronization point for marshaling between threads. Asynchronous APIs such as those that use the event-based programming model will use the `SynchronizationContext` to run asynchronous computations and to post results back to the original thread when appropriate. There is a default implementation that just executes asynchronous callbacks and completion events on the thread pool; but components such as Windows Forms, WPF, and ASP.NET provide their own implementations with customized behavior. In addition to asynchronous transfer of control, the `SynchronizationContext` can also be used to hook synchronization waits.

An Overview of the SynchronizationContext API

The basic `SynchronizationContext` API is fairly compact.

```
public class SynchronizationContext {
    // Constructors
    public SynchronizationContext();

    // Instance Methods
    public virtual SynchronizationContext CreateCopy();
    public bool IsWaitNotificationRequired();
    public virtual void OperationCompleted();
    public virtual void OperationStarted();
    public virtual void Post(SendOrPostCallback d, object state);
    public virtual void Send(SendOrPostCallback d, object state);
    protected void SetWaitNotificationRequired();
    public virtual int Wait(
        IntPtr[] waitHandles,
        bool waitAll,
        int millisecondsTimeout
    );
    // Static Properties
    public static SynchronizationContext Current { get; }

    // Static Methods
    public static void SetSynchronizationContext(
        SynchronizationContext syncContext
    );
    protected static int WaitHelper(
        IntPtr[] waitHandles,
        bool waitAll,
        int millisecondsTimeout
    );
}
```

There is a notion of a “current” context, which is accessible with the `Current` property and settable with `SetSynchronizationContext`. There’s no capability to chain contexts together, so when a component replaces the current one, it must consider what that means for a context that already existed and also take care to revert to the old context (as appropriate) at a later time. More often than not, a single context is established per thread, such as with the GUI thread in Windows Forms and WPF programs.

As its name implies, `CreateCopy` can be used to create a copy of a context, usually for purposes of flowing to another thread. The ability to create a copy is used primarily by `ExecutionContext`. The `ExecutionContext` holds things like the `SecurityContext` and `LogicalCallContext` for a particular managed thread, but also considers the current `SynchronizationContext` part of its overall state too. When an `ExecutionContext` is captured for purposes of flowing via its `Capture` method, a copy of the `SynchronizationContext` is made. When the `Run` method is subsequently used, the code run in the context will also see the newly copied `SynchronizationContext`.

Creating a new managed `Thread` and queueing work to the thread pool with `ThreadPool.QueueUserWorkItem` explicitly suppress flowing of the `SynchronizationContext`, even though the other aspects of the `ExecutionContext` still flow. This was a design decision made by the CLR team that avoided some compatibility issues in ASP.NET, which, remember, has implemented its own context.

All of this is well and good, but probably isn’t very interesting until you understand precisely what a `SynchronizationContext` object itself can be used for. The major “workhorse” methods are the `Post`, `Send`, and, sometimes, `Wait` methods. Both `Post` and `Send` take as input a `SendOrPostCallback` delegate and a separate `state` object and invoke the delegate in a certain way. The delegate is defined very simply as follows.

```
public delegate void SendOrPostCallback(object state);
```

The `Post` method performs an asynchronous invocation of the callback, and `Send` performs a synchronous invocation of the callback. These callbacks execute within the target “context.” The default implementations of these methods on `SynchronizationContext` are very simple.

```
public virtual void Post(SendOrPostCallback d, object state)
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(d.Invoke), state);
}

public virtual void Send(SendOrPostCallback d, object state)
{
    d(state);
}
```

As we'll see, the GUI oriented subclasses use the `Post` and `Send` methods as an opportunity to marshal work back to the GUI thread responsible for instantiating that particular `SynchronizationContext` object. This uses the facilities reviewed earlier. In that sense, you can just think of them analogues to the USER32 `PostMessage` and `SendMessage` APIs.

The `OperationStarted` and `OperationCompleted` methods are specific to the `AsyncOperationManager` we're about to review momentarily. They contain empty implementations in `SynchronizationContext` itself but can be overridden to perform any kind of book keeping that is necessary to track number of outstanding operations and the like.

Finally, the `Wait` method can be overridden to hook blocking calls. It has a signature much like the Win32 native `WaitForMultipleObjects` function: it takes an array of `HANDLEs` (in the form of an `IntPtr[]` array), a boolean `waitAll` parameter to specify the kind of wait, and a timeout in milliseconds (or `-1`, a.k.a. `Timeout.Infinite`, to specify no timeout). The central blocking routine in the CLR will invoke this method on your context, but only if has set the `IsWaitNotificationRequired` property to `true`. This is done by calling `SetWaitNotificationRequired` in the subclass, typically from within its constructor. At that point, the CLR will call out to your type for all blocking calls occurring on threads whose `Current` context is yours. The protected static method `WaitHelper` is the CLR's default implementation, in case you decide in the callback that you needn't do anything special.

A few things are worth calling out.

- Writing a custom `Wait` method is highly susceptible to stack overflows. If you stop to think about it, this should be obvious. If any code in the callback blocks, it will just get rerouted out to your custom `Wait` method, and so on. Because there's so much hidden blocking in .NET

Framework code, and since it often only happens conditionally (like contentious lock acquires), it can be incredibly difficult to determine whether you've written the `Wait` method correctly.

- The code inside of `Wait` is the wait itself. If you return, then whatever code was blocking will assume the API is being honest and truthful. Clearly this can be used to accidentally (or maliciously even) make code run without the proper protection of locks, and the like, so it should be used with extreme care.
- The wait objects are represented as an `IntPtr[]`, which means you really can't correlate them back to the original synchronization objects from which they came. For example, waiting on a `Monitor`, `EventWaitHandle`, and so forth will all route through this method, but you can't easily map the `IntPtr` back.
- The CLR doesn't always call this method for waits. The reason is that the callout stems from deep inside the CLR VM itself. Some waits may occur while a GC is in progress, for example, at which point it's wholly illegal to invoke any managed code. The CLR just reverts to its default wait logic in such cases.

Installing your own `SynchronizationContext` is tricky and should only be done when you own the thread. As noted, there is no compositional mechanism to support multiple contexts on the same thread, so there is an inherent race anytime multiple components want to install their own. You can chain contexts together, but this only works in some limited circumstances, such as when you just need to wrap calls to add some kind of logic such as tracing.

For example, here's a general purpose implementation that passes through all `CreateCopy`, `Post`, `Send`, `OperationStarted`, and `OperationCompleted` method calls. It overrides `Wait`, however, to enable wrapping waits in arbitrary pre and post delegates.

```
using System;
using System.Threading;

delegate object PreWaitNotification(
    IntPtr[] waitHandles,
    bool WaitAll,
```

```
    int millisecondsTimeout
);
delegate void PostWaitNotification(
    IntPtr[] waitHandles,
    bool WaitAll,
    int millisecondsTimeout,
    int ret,
    Exception ex,
    object state
);

class BlockingNotifySynchronizationContext : SynchronizationContext
{
    private SynchronizationContext m_captured;
    private PreWaitNotification m_pre;
    private PostWaitNotification m_post;

    public BlockingNotifySynchronizationContext(
        PreWaitNotification pre, PostWaitNotification post) :
        this(SynchronizationContext.Current, pre, post)
    {
    }

    public BlockingNotifySynchronizationContext(
        SynchronizationContext captured,
        PreWaitNotification pre, PostWaitNotification post)
    {
        m_captured = captured;
        m_pre = pre;
        m_post = post;

        // Make sure we get notified of blocking calls.
        SetWaitNotificationRequired();
    }

    public override SynchronizationContext CreateCopy()
    {
        return new BlockingNotifySynchronizationContext(
            m_captured == null ? null : m_captured.CreateCopy(),
            m_pre, m_post
        );
    }

    public override void Post(SendOrPostCallback cb, object s)
    {
        if (m_captured != null)
            m_captured.Post(cb, s);
        else
            base.Post(cb, s);
    }
}
```

```
public override void Send(SendOrPostCallback cb, object s)
{
    if (_captured != null)
        _captured.Send(cb, s);
    else
        base.Send(cb, s);
}

public override void OperationCompleted()
{
    if (_captured != null)
        _captured.OperationCompleted();
    else
        base.OperationCompleted();
}

public override void OperationStarted()
{
    if (_captured != null)
        _captured.OperationStarted();
    else
        base.OperationStarted();
}

public override int Wait(
    IntPtr[] waitHandles, bool waitAll,
    int millisecondsTimeout)
{
    // Invoke the pre callback.
    object s = _pre(waitHandles, waitAll, millisecondsTimeout);

    // Now perform the wait.
    int ret = 0;
    Exception ex = null;
    try
    {
        if (_captured != null)
            ret = _captured.Wait(
                waitHandles, waitAll, millisecondsTimeout);
        else
            ret = base.Wait(
                waitHandles, waitAll, millisecondsTimeout);
    }
    catch (Exception e)
    {
        ex = e;
        throw;
    }
    finally
}
```

```
{  
    // Invoke the post callback.  
    m_post(  
        waitHandles, waitAll, millisecondsTimeout, ret, ex, s);  
}  
  
return ret;  
}  
}
```

What you would use such functionality for is entirely up to you. For example, you might decide to log information such as how long waits took on average. This could be done by returning a timestamp from the predelegate, which is then passed in as the state object to the postdelegate. You would have to be careful that the tracing framework you use doesn't acquire locks internally. Another example of a possibly useful feature using `SynchronizationContext` would be to force the addition of timeouts to all waits. If a timeout of 5 seconds was exceeded, you might fire an exception or `FailFast` to help track down possible deadlocks. This would be a convenient debugging mechanism and not something you'd necessarily want to rely on at runtime.

Implementations of SynchronizationContext in the .NET Framework

The whole reason for of `SynchronizationContext` is to abstract away all of this functionality beneath an interface common among many programming models. So if you look at the subclasses of `SynchronizationContext` that ship with the .NET Framework, you'll see some application model specific marshaling techniques being used instead of the very simplistic implementations the base type offers. In fact, these same techniques map closely to those we saw earlier for marshaling work to and from GUI threads. Let's look at a few of them.

Windows Forms has its own `WindowsFormsSynchronizationContext` in `System.Windows.Forms` that is automatically installed on the GUI thread when it is set up (`Application.Run`) and which uses the `Application.ThreadContext` class internally to capture the thread responsible for the message loop. From there, it can grab the control that it can use to marshal to and from the GUI thread. Imagining this control is stored in an `m_control` variable, pseudo-code looks like this.

```
public override void Post(SendOrPostCallback d, object state)
{
    m_control.BeginInvoke(d, new object[] { state });
}

public override void Send(SendOrPostCallback d, object state)
{
    m_control.Invoke(d, new object[] { state });
}
```

The context uses `BeginInvoke` and `Invoke` to implement asynchronous post and synchronous send, respectively, both of which we reviewed earlier. In reality, there's a bit more going on in the implementation—things such as validating that the target GUI thread is still running (since it may have since exited) and so on—but this is immaterial to the discussion.

WPF also has its own `DispatcherSynchronizationContext` in the `System.Windows.Threading` namespace. Its implementation looks nearly identical to the Windows Forms one, except that it uses a `Dispatcher` object instead of a `Control` for invoking callbacks.

```
public override void Post(SendOrPostCallback d, object state)
{
    m_dispatcher.BeginInvoke(DispatcherPriority.Normal, d, state);
}

public override void Send(SendOrPostCallback d, object state)
{
    m_dispatcher.Invoke(DispatcherPriority.Normal, d, state);
}
```

In addition to implementing `Post` and `Send`, WPF also overrides the `Wait` method to suppress the CLR's automatic message pumping and alertable wait logic in key areas of WPF's internal logic where reentrancy would cause serious problems.

Both ASP.NET and the Windows Communication Foundation (WCF) have their own internal `SynchronizationContext` implementations that aren't public. They both have to do with internals of the respective systems. For example, ASP.NET sometimes invokes callbacks under a lock and also tracks the number of outstanding callbacks. And WCF has its own `ComPlusSynchronizationContext` that marshals work across COM apartments.

Asynchronous Operations

If you need to use the `SynchronizationContext` facilities for posting and sending, you'll need to deal with some boilerplate to capture the current context, check whether it's null or not (since the runtime doesn't automatically place one there), flow it around properly, and so on. Instead of doing that, you can use the `AsyncOperationManager` class, which automates all of this for you. It resides in the `System.ComponentModel` namespace. The amount of boilerplate this saves you is minuscule (a half dozen lines of code can be expressed in a couple), but given that the point of `SynchronizationContext` is to allow a simple and common way of marshaling work across the .NET Framework—to enable things like the event-based APM—it makes usage convenient enough to reach the tipping point.

The `AsyncOperationManager` just offers two static members.

```
public static class AsyncOperationManager
{
    public static SynchronizationContext { get; set; }
    public static AsyncOperation CreateOperation(
        object userSuppliedState
    );
}
```

The `SynchronizationContext` property offers an accessor that lazily initializes a default context if none exists at the time of the call. Its setter just passes the value you supply to the `SynchronizationContext.SetSynchronizationContext` method. (The fact that `SynchronizationContext` is static definitely makes these classes a whole lot less useful. But you'll typically not need to change it.) And the `CreateOperation` method is just a factory for `AsyncOperation` objects, passing the state you've supplied so that it's available. Each such object represents an operation that can be used to issue posts. This is the only way to construct one.

```
public sealed class AsyncOperation
{
    ~AsyncOperation();

    // Methods
    public void OperationCompleted();
    public void Post(SendOrPostCallback d, object arg);
    public void PostOperationCompletedCallback(
```

```
        SendOrPostCallback d,
        object arg
    );

    // Properties
    public SynchronizationContext SynchronizationContext { get; }
    public object UserSuppliedState { get; }
}
```

Each `AsyncOperation` is meant to have a single asynchronous action posted to it with the `Post` method. When the operation finishes, the callback should invoke `OperationCompleted` or, if there is some action associated with the completion of said operation, it should invoke `PostOperationCompletedCallback`. This internally calls `OperationCompleted` and acts doubly as a way to mark completion and to queue an asynchronous completion activity back onto the `SynchronizationContext` that created the operation.

When the context is for a GUI framework, this makes it very easy to “get back” to the GUI thread to update some part of the screen. If no completion is explicitly signaled, `AsyncOperation`’s finalizer will do it. (Explicitly marking completion suppresses finalization on the object, because completion may only be done once per asynchronous operation; subsequent attempts will throw.) `Post` and `PostOperationCompletedCallback` both rely on the underlying context’s `Post` method. When constructed with `AsyncOperationManager.CreateOperation`, the context’s `OperationStarted` method is called, and when any of the completion mechanisms are used, the `OperationCompleted` method is called.

A Convenient Package: `BackgroundWorker`

Everything we’ve discussed so far is targeted at low-level library code. Very few application developers will want to use `SynchronizationContext` directly; it requires too much boilerplate. Even `AsyncOperationManager` and `AsyncOperation` only raise the level of abstraction slightly to the point where it’s easier to write library components that fully support asynchronicity. The `BackgroundWorker`, also a member of the `System.ComponentModel` namespace, builds on top of these facilities and codifies some of the most common uses of asynchronous operations in GUI programs. This class is meant to provide a low barrier to entry into asynchronous

programming and specifically targets higher-level application developers. Here is an overview of the API.

```
public class BackgroundWorker : Component
{
    // Constructor
    public BackgroundWorker();

    // Methods
    public void CancelAsync();
    public void ReportProgress(int percentProgress);
    public void ReportProgress(int percentProgress, object userState);
    public void RunWorkerAsync();
    public void RunWorkerAsync(object argument);

    // Events
    public event DoWorkEventHandler DoWork;
    public event ProgressChangedEventHandler ProgressChange;
    public event RunWorkerCompletedEventHandler RunWorkerCompleted;

    // Properties
    public bool CancellationPending { get; }
    public bool IsBusy { get; }
    public bool WorkerReportsProgress { get; set; }
    public bool WorkerSupportsCancellation { get; set; }

    // Protected members
    protected virtual void OnDoWork(DoWorkEventArgs e);
    protected virtual void OnProgressChanged(
        ProgressChangedEventArgs e
    );
    protected virtual void OnRunWorkerCompleted(
        RunWorkerCompletedEventArgs e
    );
}
```

BackgroundWorker provides several key features.

- The basic model entails providing an event handler for a work function. It is named **DoWork**. At some point, often in response to a button click, you will kick off the asynchronous work by calling **RunWorkerAsync**. You may optionally provide a state parameter. The implementation handles marshaling work to another thread and eventually (if you so choose) firing additional events back on the GUI thread via the **RunWorkerCompleted** event handler.

- The `IsBusy` property changes to `true` while an asynchronous operation is actively running and is automatically reverted back to `false` when it finishes.
- Incremental progress can be responded to by hooking the `ProgressChanged` event handler. The asynchronous work is responsible for setting `WorkerSupportsProgress` to `true` if it supports this and must periodically call the `ReportProgress` method, which in turn causes `BackgroundWorker` to run the event code on the GUI thread. Progress is reported with a number between `0` and `100`, and state can be attached to it.
- Cancellation is supported in a first-class way. Setting `WorkerSupportsCancellation` to `true` indicates that the asynchronous code will periodically check the `CancellationPending` property and, if `true`, voluntarily quit and cleanup whatever work was in progress. Cancellation is then initiated with a call to the `CancelAsync` method.
- Because `BackgroundWorker` implements the `IComponent` interface, it offers nice Visual Studio IDE integration. You can drag and drop it onto the designer surface and wire up all of the interesting event handlers without having to write any code.
- If it's not evident, all of this is built on top of the `AsyncOperationManager` and, therefore, `SynchronizationContext`. By calling `RunWorkerAsync`, a new `AsyncOperation` is created, and a work item is explicitly queued to the CLR thread pool. This work invokes the `DoWork` event handler, catches exceptions to marshal back (if any), and eventually calls `PostOperationCompleted` on the underlying `AsyncOperation`. This transfers control back to the GUI thread, allowing the `RunWorkerCompleted` event to execute. Any calls to report progress directly use `Post`. All of this is done internally so you can remain unaware of it, but it's a good example of using all of the machinery we just reviewed to provide a nice, simple abstraction.

A subtlety around `BackgroundWorker`'s use is that each worker may only represent a single asynchronous operation. If you try to use it for more than one simultaneously, an `InvalidOperationException` will be generated by `RunWorkerAsync`. You will need to specifically have code to prevent this

from happening, such as disabling any buttons meant to initiate asynchronous work while an outstanding request is running, or by generating multiple `BackgroundWorkers` and tracking them in a list of some sort.

Each of the events has its own `EventHandler` type, each with its own `EventArgs` class.

```
// DoWork event

public delegate void DoWorkEventHandler(
    object sender, DoWorkEventArgs e
);

public class CancelEventArgs : EventArgs
{
    public CancelEventArgs();
    public CancelEventArgs(bool cancel);

    public bool Cancel { get; set; }
}

public class DoWorkEventArgs : CancelEventArgs
{
    public DoWorkEventArgs(object argument);

    public object Argument { get; }
    public object Result { get; set; }
}

// ProgressChanged event

public delegate void ProgressChangedEventHandler(
    object sender, ProgressChangedEventArgs e
);

public class ProgressChangedEventArgs : EventArgs
{
    public ProgressChangedEventArgs(
        int progressPercentage, object userState
    );

    public int ProgressPercentage { get; }
    public object UserState { get; }
}

// RunWorkerCompleted event

public delegate RunWorkerCompletedEventHandler(
    object sender, RunWorkerCompletedEventArgs e
);
```

```
public class RunWorkerCompletedEventArgs : AsyncCompletedEventArgs
{
    public RunWorkerCompletedEventArgs(
        object result, Exception error, bool cancelled
    );

    public object Result { get; }
    public object UserState { get; }
}
```

Let's review each briefly in turn.

`DoWorkEventArgs` derives from `CancelEventArgs` and adds `Result` and `Cancel` properties. `Result` is used to marshal any kind of result from the background work back to the GUI thread. The `Cancel` flag's purpose is to let the completion handler know the work quit voluntarily in response to seeing a `CancellationPending` of `true` due to a `CancelAsync` call. The `RunWorkerCompletedEventArgs` object passed to the completion handler copies the `Result` (if any) and the `Cancelled` flag— inherited from `AsyncCompletedEventArgs`—based on the `DoWorkEventArgs` object's properties that were set by the callback.

`ProgressChangedEventArgs` is straightforward and marshals the input passed to `ReportProgress` to the GUI thread so it can update whatever state is appropriate, often involving things such as a progress bar control.

Finally, `RunWorkerCompletedEventArgs` offers `Error` and `UserState` properties in addition to the `Result` and `Cancel` properties mentioned already. If the `DoWork` code throws an unhandled exception, it will be caught and stored in the `Error` property.

Where Are We?

In this chapter, we've reviewed the fundamental architecture shared among all Windows GUI frameworks, including USER32, Windows Forms, and WPF. We saw why the single threaded nature of this architecture poses challenges to building responsive systems, and why marshaling callbacks off and onto the special GUI thread is often necessary.

We've also reviewed the mechanisms used in Windows Forms and WPF to enable this kind of marshaling, to inquire about when marshaling

is necessary, and a little about how message loops are run in both systems. We then moved on to see that .NET 2.0 introduced the `SynchronizationContext` as a common shared infrastructure beneath these models, and how it has enabled higher-level abstractions such as the `AsyncOperationManager` and `BackgroundWorker`. We also saw that `BackgroundWorker` is a great way to easily add asynchrony to your GUI applications, and that it has built in support for many common tasks.

This was the last chapter of the book. At this point, you should be fully equipped to build real-world concurrent programs, ranging from low-level parallel algorithms, data structures, and systems software on up to high-level responsive GUI applications. Good luck, and have fun.

FURTHER READING

- C. Anderson. *Essential Windows Presentation Foundation (WPF)* (Addison-Wesley, 2007).
- D. Box. *Essential COM*. (Addison-Wesley, 1998).
- C. Brumme. Apartments and Pumping in the CLR. Weblog article,
<http://blogs.msdn.com/cbrumme/archive/2004/02/02/66219.aspx> (2004).
- D. Duis, J. Johnson. Improving User Interface Responsiveness Despite Performance Limitations. In *Proceedings of the IEEE Computer Society International Conference*, 1990).
- J. Duffy. Application Responsiveness: Using Concurrency Can Enhance User Experiences. *Dr. Dobb's Journal* (2006).
- G. H. Forman. Obtaining Responsiveness in Resource-Variable Environments. PhD dissertation (University of Washington, 1998).
- I. Griffiths. Give Your .NET-based Applications a Fast and Responsive UI with Multiple Threads. *MSDN Magazine* (2003).
- R. Grimes. Synchronization Domains. *Dr. Dobb's Journal* (2004).
- N. Kramer. Threading Models. Windows Presentation Foundation, Weblog essay,
<http://blogs.msdn.com/nickkramer/> (2006).
- C. Sells, M. Weinhardt. *Windows Forms 2.0 Programming*, Second Edition (Addison-Wesley, 2006).

PART V

Appendices

A

Designing Reusable Libraries for Concurrent .NET Programs

AS THE INDUSTRY at large grows up with concurrency as a first-class design concept, the reusable libraries and larger frameworks that developers use to build complex systems and applications must increasingly cope with **pervasive concurrency**. Although this book has spent a great deal of time expanding on the mechanisms, concepts, and best practices of concurrent programming, this appendix presents several important ideas in a single, consolidated place.

Pervasive concurrency may sound revolutionary at first, but the industry-wide transformation from sequential to concurrent won't take place overnight. Early adoption will occur in applications, while libraries and frameworks will evolve slowly and carefully over time. The core platform components have only begun this shift, and a full evolution of the software stack will necessarily follow suit and take longer to occur. While the guidance here will also evolve along with the platform, the advice can be used when writing code today.

Although most of the contents of this appendix are worded in a .NET specific way, a large portion of it can be generalized to building C++ libraries.

The 20,000-Foot View

There are several major themes library developers must focus on in their design and implementation in order to prepare for pervasive concurrency. These are not concrete rules that can be easily followed, but rather general high-level themes of focus.

- The level of reliability developers demand of .NET libraries is increasing over time. Yet the introduction of more concurrency leads to subtle timing bugs—such as races and deadlocks—which will now occur with an increasing probability. Those rare races that would have required obscure multistep sequences of context switches at very specific lines of code on single processor machines, for example, will start surfacing regularly for applications running on 8-core desktop machines. Library authors have gotten better at finding and fixing these types of bugs before shipping, but nobody catches them all. Fixing more of them will require intense concurrency oriented testing and aggressive adoption of best practices that statistically reduce the risk.
- Many libraries assume that the identity of the OS thread remains constant over time in a number of places—a problem called **thread affinity**—preventing user-mode scheduling: specifically, (1) multiple pieces of work can't share the same OS thread stack, and (2) a user-mode or continuation based scheduler can't readily move work between OS threads as resources permit. Windows GUIs are notorious for their reliance on thread affinity in addition to COM STAs. While fibers aren't the solution for user-mode scheduling, it's probable that something like them will be necessary to achieve scale.
- Scaling due to parallelism will become just as important for many kinds of problems as single threaded sequential performance. This not only means using parallelism internally for compute-bound APIs but also not getting in the way of higher-level application concurrency. If a developer's application is massively concurrent, you have to assume he or she will notice if you take an overly coarse-grained lock, block the thread unexpectedly, or acquire

thread affinity such that work can't remain agile. Faced with such issues, developers will have no recourse other than to refactor, rewrite, and/or avoid the use of certain APIs. And worse, they'll learn all of this through trial and error.

- APIs often utilize operations with variable latency as an implementation detail. If a developer is trying to build a scalable application or a responsive GUI, it's imperative that they avoid blocking. If some high latency operation is inevitable, either because of an API or architectural design choice, developers should be made aware of this fact. This will at least allow them to call the API in an appropriate way, for example by offloading it from the GUI thread. A better option is to provide them the choice to use an alternative asynchronous version of the API—such as one of the asynchronous patterns from Chapter 8, Asynchronous Programming Models—which can often use the platform's rich intrinsic asynchronous file and network I/O capabilities.

These are all dense and complex issues and are intertwined. Many concerns can be teased apart and mitigated by following a set of best practices. This is not to say they are all easy to achieve. These guidelines will evolve as the community at large learns more. And I am hopeful that they will be reinforced with library and tool support over time.

The Details

Now that we've seen some of the high level themes that .NET library developers should keep in mind, let's look at some detailed best practices. All of these have been touched on in one way or another throughout the book. References are included where appropriate.

Locking Models

1. Static state access must be thread safe.

Any library code that accesses shared state must be done thread safely. For most libraries, this means that objects reachable through a static variable (that the library itself places there) must be

protected by a lock. The lock has to be held over the entire invariant under protection (for multistep operations) to ensure that other threads don't witness state inconsistencies in between updates. Protecting invariants spanning multiple fields requires that lock granularity is large enough, but not so big that it leads to scalability problems. Read-modify-write bugs are also a common mishap here; for example, if you're updating a `static` counter, it must be done with an `Interlocked.Increment` operation, done under a lock or be protected by some other synchronization mechanism.

Reads and writes to `static` variables whose data types are not word size (i.e., 32 bits or 4 bytes on 32-bit, 64 bits or 8 bytes on 64-bit) also need to happen under a lock or with the appropriate `Interlocked` method. Otherwise, threads can observe "torn values. for example, while one thread writes a 64-bit value, `0aaaaaaaaaaaaaaaaaaaa` to a field— involving two individual 32-bit writes in the object code—another thread may run and see a garbage value, say, `0aaaaaaaaaaaaaaaaaaaa`, because the high 32-bit word was written first. Similar problems can happen to GUID fields on all architectures because GUIDs are 128 bits wide. `Int64s` (`longs`) on 32-bit machines also fall into this category, as do value types built out of said data types.

This responsibility doesn't extend to instance field accesses, even if the library objects end up getting stored in `static` variables by the developers using the library. In other words, only if the library makes state accessible through a `static` variable does the library need to protect it with synchronization. Everything else is up to the developers using the library. In some cases, a library author may choose to make a stronger guarantee—and clearly document it—but it should certainly be the exception rather than the default choice. A good example is a library that is specifically targeting concurrent programs.

2. Instance state access needn't be thread safe. In most cases, it should not be.

As an extension of the previous point, protecting library instance state with locks introduces performance overhead that is often ill

justified. The granularity of such locks is typically too small for any application operation of interesting size. And if the granularity could be wrong you'll need to expose implementation locking details or it was a waste of time. Claiming an object performs thread safe reads/writes to instance fields can even give users a false sense of safety because they might not understand the subtleties around locking granularity.

.NET still has numerous types that claim: "This type is thread safe" in the MSDN documentation, but this is typically limited to simple, immutable value types.

As an example of where this went wrong in the past, the .NET Framework V1.0 included synchronizable versions of most of its collections. These used coarse-grained locking, meaning they didn't exploit the natural concurrency of certain container types (as we saw in Chapter 12, Parallel Containers). To deal with the improper granularity problem, they exposed a `SyncRoot` property. In retrospect, this whole scheme turned out to be a bad idea: customers were frequently plagued by race conditions they didn't understand, and, for those who kept a collection private to a single thread or used higher-level synchronization rather than the collection's lock, the performance overhead was substantial and prohibitive. The new V2.0 generic collections left this part out.

3. Use isolation and immutability where possible to eliminate races.

If you don't share and mutate data, it doesn't need lock protection. CLR strings and most built in value types, for example, are immutable. Isolation can also be used to hide intermediate state transitions, although typically also requires that multiple copies are maintained and periodically synchronized with a central version to eliminate staleness. This approach can be used to improve scalability, particularly for highly shared state. Many CRT `malloc/free` implementations will use a per thread pool of memory and occasionally rendezvous with a central process-wide pool to eliminate contention, for example. You are encouraged to think about exposing isolation and immutability in your public API surface area.

4. Document your locking model.

Most library code has a simple locking model: static state manipulation is thread safe and everything else is not (see #1 and #2 above). But if your internal locking schemes are more complex, you should document those using asserts (see below), good comments, and detailed design documents with information about what locks protect what data. Of course all of this must be carefully verified with testing. If any of these subtleties are surfaced to users of your class then those must also be explained in product documentation and, preferably, reinforced with some form of tools and analysis support. COM/GUI STAs, for example, have esoteric threading schemes, where synchronization leaks heavily into the programming model. As a community, we would be best served if there are no new invented instances of such specialized models.

Using Locks**5. Use the C# lock and VB SyncLock statements for all synchronized regions.**

Following this guidance ensures that locks will be released even in the face of asynchronous thread aborts, leading to fewer deadlocks (statistically speaking). These statements generate code such that the corresponding `Monitor.Exit` will always be run in the finally block if the `Monitor.Enter` succeeded. This still doesn't protect code from rude AppDomain unloads—requiring more intricate techniques that won't be discussed here—but this is not something most library developers have to worry about: tolerating rude AppDomain unloads is only necessary when protecting cross AppDomain state in a sophisticated CLR host like SQL Server.

6. Avoid making calls to someone else's code while you hold a lock.

This applies to most virtual, interface, and delegate calls while a lock is held—as well as ordinary statically dispatched calls—into subsystems you aren't familiar with. The more you know about the code being run while you hold a lock, the better off you will be. If you follow this approach, you'll encounter far fewer deadlocks, hard to reproduce reentrancy bugs, and surprising dynamic composition

problems, all of which can lead to hangs when your API is used on the UI thread, reliability problems, and frustration for your customer. Locks don't compose very well; ignoring this and attempting to compose parts of your components that use them in this way is fraught with peril.

7. Avoid blocking while you hold a lock.

This is self explanatory. Admittedly, it is sometimes unavoidable. Trying to acquire a lock is an operation that can block under contention, so by definition, if you need to hold more than one lock at once, you will be violating this advice. But what's more important, blocking on high or variable latency operations such as I/O will effectively serialize any other thread trying to acquire that lock behind your I/O request. If that other thread trying to acquire the lock is on the UI thread, you may have just indirectly caused a user visible hang. The developer may not understand the cause of this hang if the lock is buried inside of your library, and it may be tricky and error prone to work around. At the very least, extending lock hold times like this can cause convoys.

Aside from having scalability impacts, blocking while a lock is held can lead to deadlocks and invariants being broken. Any time you block on an STA thread, the CLR uses it as a chance to run the message loop. When run on pre-Windows 2000 that means running custom `MsgWaitForMultipleObjects` pumping code, and OLE's `CoWaitForMultipleHandles` post-Windows 2000. While this style of pumping processes only a tiny subset of GUI messages, it can dispatch arbitrary COM to CLR interop calls. These calls include cross thread/apartment `SendMessage` calls, such as an MTA to STA call through a proxy. If this happens while a lock is held, that newly dispatched work also runs under the protection of the lock. If the same object is accessed, this can lead to surprising bugs where invariants are still broken inside the lock.

Try to minimize the time you hold a lock and move all blocking and communication across apartments, threads, processes outside the edges of those lock acquisition/releases. All libraries should strive to only acquire locks at the leaves of callgraphs to the extent that it is possible.

8. Assert lock ownership.

Races often result when some leaf-level code assumes a lock has been taken at a higher level on the call stack, but the caller has forgotten to acquire it. Or maybe the owner of that code recently refactored it and didn't realize the implicit pre-condition that was broken in the process. This may go undetected in test suites unless the race actually occurs in the world.

All new locks in the .NET Framework provide APIs to test if the lock is held. `Monitor` currently lacks an `Isheld` API, so if you want to heed this advice with `Monitor` you'll need to maintain the extra state yourself. `Isheld`-like functionality should never be used to dynamically influence lock acquisition and release at runtime, for example avoiding recursion and taking or releasing based on its value. It is meant as a debugging aid only.

9. Avoid lock recursion in your design. Use a non recursive lock where possible.

Recursion is one of the problems highlighted in Chapter 11, Concurrency Hazards, that can lead to reliability and reentrancy problems. Lock recursion is typically an indication of an oversimplified synchronization policy. For instance, many designs use lock recursion as a way to avoid splitting functions into those that take locks (nonrecursively) and those that assert that locks are already taken. This can lead to a reduction in code size, but usually results in a more brittle design in the end. For this reason, most new .NET locks are nonrecursive by default and only offer it as an opt in setting.

Recursive lock acquires are redundant and add unnecessary performance overhead. But worse, depending on recursion can make it more difficult to understand the synchronization behavior of your program, in particular at what boundaries invariants are supposed to hold. Usually we'd like to say that the first line after a lock acquisition represents an invariant "safe point" for an object, but as soon as recursion is introduced this statement can no longer be made confidently. This in turn makes it more difficult to ensure correct and reliable behavior when dynamically composed.

10. Don't build your own lock.

Most locks are built out of simple principles at the core. There's a state variable, a few interlocked instructions (exposed to managed code through the `Interlocked` class), and some form of spinning and possibly waiting on an event when contention is detected. Given this, it may look straightforward to build your own. This is deceptively difficult.

CLR locks have to coordinate with hosts so that they can perform deadlock detection and sophisticated user-mode scheduling for hosted customer authored code. Some of .NET's locks (`Monitor`) make higher reliability guarantees so that they can be safely used during AppDomain teardown. Real locks are tuned to use an ideal mixture of spinning and waiting across many OS SKUs, CPU architectures, and cache hierarchy arrangements. Such spinning must be written to work correctly with Intel HyperThreading and to avoid priority induced starvation. Locks must mark critical regions of code so that would-be thread aborts will be performed correctly while sensitive shared state manipulation is under way. And the C# and VB languages offer the "lock" and "SyncLock" keywords (as highlighted earlier) whose code generation pattern ensures that code won't orphan locks in the face of asynchronous thread aborts. To get all of this right requires a lot of hard work, time, and testing.

With that said, .NET may not currently have every lock you could ever want. Spin locks are a popular request that can help with performance scalability of highly concurrent and leaf-level code, as demonstrated in Chapter 14, Performance and Scalability. It's best to make do with what is available out-of-the-box and to look for third party locks only if necessary.

11. Don't call `Monitor`. Enter on AppDomain agile objects (Types and Strings).

Instances of some `Type` objects are shared across AppDomains. The most notable are `Types` for domain neutral assemblies (such as `mscorlib.dll`) and cross assembly interned `Strings`. While it may look innocuous, locks taken on these things are visible across all

AppDomains in the process. As an example, two AppDomains executing this same code will interfere with each other.

```
lock (typeof(System.String)) { ... }
```

This can cause severe reliability problems should a lock get orphaned in an add-in or hosted scenario, possibly causing cross AppDomain deadlocks stemming (seemingly inexplicably) from deep within your library. The resulting code also leads to false contention between code running in different domains and, therefore, can impact scalability in a way that is very difficult for customers (and library authors) to reason about.

12. Don't use a machine- or process-wide synchronization primitive when AppDomain-wide would suffice.

The **Mutex** and **Semaphore** types in the .NET Framework should only be used for legacy, interoperability, cross AppDomain, and cross process reasons. They are heavier weight—several orders of magnitude slower than a **CLR Monitor**, as mentioned in Chapter 6, Data and Control Synchronization—and they introduce reliability and affinity problems. They can be orphaned, out of process **denial of service** attacks can be mounted, and they can introduce scalability bottlenecks. Moreover, they are associated with the OS thread and, therefore, impose thread affinity.

13. A race condition or deadlock in library code is always a bug.

This seems like it should be obvious. But it's not always cut and dried. Race conditions and deadlocks can be very difficult to fix. Sometimes fixing one requires refactoring a lot of mostly working code to make some (seemingly) corner case and obscure sequence of events work correctly. It's tempting to rearrange things to narrow the window of the race or reduce the likelihood of a deadlock. But *never* lose sight of the fact that, no matter how narrow the likelihood, a race or deadlock is a severe correctness problem.

Sometimes fixing a bug like this requires making breaking changes. Sometimes you may not have enough time to fix the bug in time to ship your product. In either case, this is something that

should be measured and explicitly decided based on the quality bar for the product at the time the bug is found. Remember that as higher degrees of concurrency are used in the hardware, the probability of these bugs resurfacing becomes higher. A race condition that reproduces only once in a while on high-end machines in 2008 could begin happening routinely on middle-of-the-line machines just a couple years later. If you decided in 2008 to ship as is, you may pay for that decision in 2010 when support costs demand that you supply a costly servicing fix.

Reliability

14. **Every lock acquisition might throw an exception. Be prepared for it.**

Most locks lazily allocate a kernel event object if a lock acquisition encounters contention, including CLR monitors. This allocation can fail during low resource conditions, causing OOMs originating from the entrance to the lock. (A typical nonblocking spin lock cannot fail with OOM, which allows it to be used in some resource constrained scenarios where normal locks might be off-limits.) Thread interruptions can lead to `ThreadInterruptedExceptions`. And SQL Server can perform deadlock detection and even break those deadlocks by throwing a `System.Runtime.InteropServices.COMException`.

Often there isn't much that can be done in response to such an exception, except for letting it unwind the stack. This unwind should be done cleanly so that the process doesn't deadlock or crash. Reliability and security sensitive code that must deal with failure robustly should consider this possible point of failure and may need to take special action like reverting partially made updates.

15. **Lock leveling should be used to avoid deadlocks.**

Lock leveling is a scheme in which a relative number is assigned to all locks, and a strict ordering among them is enforced. This discipline guarantees deadlock freedom, as was described in Chapter 11, Concurrency Hazards.

Without using something such as lock leveling, libraries are usually subject to dynamic composition and reentrancy induced

deadlocks, causing users trying to write even moderately reliable code a lot of frustration. This frustration only becomes worse as library usage is woven throughout a highly concurrent application. All that said, there are two problems that will surely get in the way of adopting lock leveling today.

First, there is no standard leveled lock type in the .NET Framework today. While Chapter 11 contains a sample for one, most library developers will not start adopting lock leveling in any serious way without an official .NET base class and associated guidelines. It's also difficult to be successful building libraries that use lock leveling without good tooling support.

That last statement ties into the second problem: lock leveling is a very onerous discipline. The CLR uses it internally for the parts of the system that are relatively closed, but lock leveling doesn't apply so well when dynamic composition is used. Levels are represented using numbering schemes that are arbitrarily chosen on a per assembly basis. You can develop schemes to extend levels across assemblies, and possibly even cook up some native interoperability story, but these are all features that would have to be built on top of the base lock leveling scheme. Again, without standard library support, having to build all this yourself as a library developer is often a nonstarter.

Lock leveling is one of the more promising techniques that we have for avoiding deadlocks. An alternative to lock leveling is to use only nonrecursive locks and closed lock regions. This is a good practice to follow wherever possible.

16. Restore sensitive invariants in the face of an exception before the first pass executes up the stack.

This is in part a security concern as well as a reliability concern. The CLR exception model is the two-pass model inherited from Windows SEH. The first pass runs before `finally` blocks execute, meaning that the locks held by the thread at the time of a throw are still held when up stack exception filters are run. IL supports filters, although most C# developers are unaware because the language itself doesn't expose syntax for them (VB and VC++ do). Code inside of filters runs with locks held and can recursively acquire them.

If you're using .NET security APIs, CAS asserts and impersonation cannot leak in this way, but anything custom can. You can stop the first pass and ensure your lock is released or sensitive state reverted by wrapping a `try/catch` around the sensitive operation and rethrowing the exception.

```
try
{
    lock (...)
    {
        try
        {
            // S0: Break invariants.
            // S1: Possibly throw an exception...
        }
        finally
        {
            // S2: Restore invariants.
        }
    } // S3: Release the lock.
}
catch
{
    // Just break the 1st pass and repropagate.
    throw;
}
```

In this example, we ensure both statements S2 (which restores invariants) and S3 (which releases the lock) execute before running the first pass. This is only something you should consider if security and reliability requirements dictate it. Also keep in mind that doing so hampers debuggability.

17. If class constructors are required to have run for code inside of a lock, consider eagerly running the constructor with a call to `RuntimeHelpers.RunClassConstructor`.

Reentrancy involving cctors can be difficult to reason about because behavior is nondeterministic based on whether a class has been constructed already. Anyplace your code accesses `statics` is an opportunity for the CLR to run a cctor on the current thread. Aside from the fact that running the cctor could cause an exception (much like an asynchronous exception), it could also recursively acquire a

lock that the current thread holds. If that lock protects some state that is now inconsistent, broken invariants can be seen. You can consider calling `Runtime.RunClassConstructor` before acquiring a lock to eagerly hoist the cctor's execution, avoiding such reentrancy issues.

18. Don't use Windows asynchronous procedure calls (APCs) in managed code.

APCs pollute the OS thread to which they are tied and are a strange form of thread affinity. They can fire at arbitrary alertable blocking points in the code, including after a thread pool thread has been returned to the pool, after the finalizer thread has gone on to invoke `Finalize` other objects in the process, or even at some random blocking point deep within the EE (perhaps while we aren't ready for it, as in during a garbage collection). If an APC raises an exception, the state of affairs at the time of the crash is likely to be confusing. The stack certainly will be. APCs also represent possible security threats and can also introduce many subtle reentrancy and reliability problems of the kind already outlined. There are no .NET APIs to interact with APCs, and this is for a good reason; resist the temptation to P/Invoke to access them.

19. Don't change a thread's priority.

Unless a library owns the thread, it has no business changing its priority; even if it is owned, priority must be used with extreme care because it opens up a host of new liveness hazards of which to be aware. These hazards include priority inversion and priority induced starvation (requiring the Windows balance set manager to guarantee forward progress). Similar problems include preventing the CLR's finalizer thread (which itself runs at high priority) from making forward progress, which can increase resource consumption. Testing for these kinds of problems in isolation will tend not to be overly successful. Instead, application developers trying to compose libraries into their programs will discover them.

20. Always test and retest a wait condition inside of a lock.

A common mistake when writing control synchronization code is to improperly retest a condition each time a thread wakes up. If you're

using an `EventWaitHandle` or `Monitor.Wait/Pulse/PulseAll`, for example, you typically need to double-check that the state is in the expected condition when waking, and you probably need to do it under the proper data synchronization. For example:

```
void Put(T obj)
{
    lock (myLock)
    {
        myQueue.Enqueue(obj);
        Monitor.PulseAll(myLock);
    }
}

T Get()
{
    lock (myLock)
    {
        while (myQueue.Count == 0)
            Monitor.Wait(myLock);
        return myQueue.Dequeue();
    }
}
```

Notice that `Get` loops around testing if the queue is empty and waits when it is. If this were a simple if-check, there would be horrendous race condition. Another thread may take the element from the queue before the awakened thread wakes up and reacquires the lock; the result is that the queue is empty by the time the thread reaches S2. The call to `myQueue.Dequeue` will likely throw an exception in response. Fixing this is generally easier with condition variables because they combine control and data synchronization. Raw events are more error prone because the lock must be separately managed.

Scheduling and Threads

21. Don't write code that depends on the OS thread ID or HANDLE. Use `Thread.Current` or `Thread.Current.ManagedThreadId` instead. When code depends on the identity of the actual OS thread, the logical task running that code is bound to the thread. This leads to thread affinity problems as mentioned earlier. If running on a system where threads are migrated between OS threads using some form of

user-mode scheduling this can break if user-mode switches happen at certain points in the code. Library code should strive to be maximally flexible and specifically not get in the way of such things.

Be on the lookout: many Win32 and Framework APIs may imply thread affinity when used. GUI APIs typically require that they are called from a thread that owns the message queue for the GUI element in question. Historically, some Microsoft components like the Shell, MSHTML.DLL, and Office COM APIs have also abused this practice. The situation on the server is much better, but still isn't perfect. Some APIs we design with the client in mind end up being used on the server, often with less than desirable results.

22. Mark regions of code that do depend on the OS thread identity with `Thread.BeginThreadAffinity/EndThreadAffinity`.

The corollary to the previous rule is that if you must have code that depends on the OS identity, you must tell the CLR (and potential host) about it. That's what the `Thread.BeginThreadAffinity` and `EndThreadAffinity` methods do. Demarking and entering such a region halts OS thread migration altogether. This is unfortunate, but at least code will remain robust.

23. Always access TLS through the .NET Framework mechanisms: `ThreadStaticAttribute` or `Thread.GetData/SetData` and related members.

The implementation of these APIs abstract away the dependency on the OS thread allowing you to store state associated with the logical piece of work. Although they sound thread specific, these store state based on whatever user-mode scheduling mechanism is being used, and, therefore, you don't take thread affinity when you use them.

24. Always access the security/impersonation tokens or locale information through the `Thread` object.

As with the previous item, the CLR abstracts away the storage of this information on the `Thread` object, via the `Thread.CurrentCulture`, `Thread.CurrentUICulture`, and `Thread.CurrentPrincipal` properties. This information is flowed across logical async points as required, and, therefore, using them doesn't imply any sort of hard OS thread affinity.

25. Always access the “last error” after an interop call via `Marshal.GetLastWin32Error`.

If you mark a P/Invoke signature with `[DllImportAttribute(..., SetLastError=true)]`, then the CLR will store the Win32 last error on the logical CLR thread. This ensures that, even if a cooperative scheduling switch happens after the P/Invoke but before you can check its value, your last error will be preserved. The Win32 APIs `GetLastError` and `SetLastError`, on the other hand, store this information in the TEB. If you are P/Invoking to get at the last error information, you are apt to be surprised if you are running in an environment that permits thread migration because the error may change before you get a chance to access it. You can avoid this by always using the safe `Marshal.GetLastWin32Error` function.

26. Avoid P/Invoking to other Win32 APIs that access data in the Thread Environment Block (TEB).

Security and locale information is something Win32 stores in the TEB that .NET offers APIs to access safely. That is easy. But many Win32 APIs access data from the TEB without necessarily saying so, or will look for and possibly lazily create some thread affine data structure (e.g., a window message queue in USER32), leading to silent thread affinity. While there is no good list of which APIs acquire or depend on thread affinity, it’s good to be aware of this issue.

Scalability and Performance

27. Consider using a reader/writer lock for read-only synchronization.
- Concurrent access to shared state often consists of a high read-to-write ratio. Given this, using exclusive synchronization (such as CLR monitors) can hurt scalability in situations with a large numbers of concurrent readers. While starting off with a reader/writer lock could be a premature optimization, many situations warrant using one, particularly very hot read regions of code.

There has been a lot of negative press about .NET’s ReaderWriterLock. In particular, the performance is at about 6 times that of successful `Monitor.Enter` calls. Unfortunately, this has (in the past)

prevented many library developers from using reader/writer locks altogether. This is the primary motivation that the `ReaderWriterLockSlim` type was added in .NET 3.5.

28. Avoid lock free code for all but the most critical performance needs.

Compilers and processors reorder reads and writes to get better performance, but in doing so make it harder to write concurrent code without locks. The CLR memory model gives a base level of guarantees that we preserve across all hardware platforms. Chapter 10, Memory Models and Lock Freedom, went into detail about when and how to exploit the memory model. When in doubt, however, avoid it if at all possible.

The reason? Lock free code is extraordinarily complicated to write, maintain, and debug for most developers, even those who have been doing it for years. This is the type of code whose proliferation will lead to poor robustness in the face of adding more and more processors. Use of `volatile` fields and calls to `Thread.MemoryBarrier` should be viewed with great suspicion, as it probably means somebody is trying to be cleverer than is required.

29. Avoid hand-coded spin waits. If you must do it, do it right.

Sometimes it is tempting to put a busy wait in very tightly synchronized regions of code. Unless written properly, however, this technique won't work well. It's often simpler to use locks or events (such as `Monitor.Wait/Pulse/PulseAll`) for this type of cross thread communication. These internally employ some reasonable amount of spinning versus waiting automatically for you. If you think spin waiting is appropriate for your situation, please consult Chapter 14, Performance and Scalability, where an overview is provided along with details of proper spin wait algorithms.

30. When yielding the current thread's time slice, use `Thread.Sleep(1)` (eventually).

Calling `Thread.Sleep(0)` doesn't let lower priority threads run. If a user has lowered the priority of their thread and uses it to call your API, this can lead to priority induced starvation. Eventually issuing

a `Thread.Sleep(1)` is the best way to avoid this problem, perhaps starting with a 0 timeout and falling back to the 1 millisecond timeout after a few tries. Particularly if you come from a Win32 background, it might be tempting to P/Invoke to `SwitchToThread` because it is cheaper than a sleep. This is because sleeps on .NET are always alertable, which incurs somewhat expensive checks for APCs. If you do so, you must realize that P/Invoking to `SwitchToThread` currently bypasses important thread scheduling hooks that call out to a would-be host.

31. Consider using spin locks for high traffic leaf-level regions of code.

A spin lock avoids giving up the timeslice on MP systems and can lead to better scalability when used correctly. Context switches in Windows are anything but cheap, ranging from 1,000s to 10,000s of cycles on average. Forfeiting the time-slice also means that you're possibly giving up data in the cache, depending on the data intensiveness of the work that is scheduled as a replacement on the processor. And any time you have cross thread causality, it can cause a rippling effect across many threads, effectively stalling a pipeline of parallel work. That said, spin locks can fare less well under extreme contention, and can cause real problems if lock hold times are lengthy.

32. You must understand every instruction executed while a spin lock is held.

Related to the previous item, spin locks are powerful but dangerous. You must ensure the time the lock is held is very small, and to ensure this you must also ensure that the entire set of instructions run is completely under your control. Virtual method calls and blocking operations are completely out of the question. Because a spin lock spins rather than blocking under contention, a deadlock will manifest as a spiked CPU and system-wide performance degradation and, therefore, is a much more serious bug than a typical hang.

33. Consider a low lock data structure for hot queues and stacks.

Windows has a set of "S-List" APIs that provide a way to do "lock free" pushes and pops from a stack data structure. This can lead to

highly scalable, nonblocking algorithms, much in the same way that spin locks do, because expensive context switches are usually avoided. We looked at a corresponding .NET class in Chapter 10, Memory Models and Lock Freedom, that can be used. Similarly, Chapter 12, Parallel Containers, took a look at several other scalable container classes that can be used in these situations.

34. Always use the CLR thread pool to introduce fine-grained concurrency.

The CLR's thread pool is optimized to ensure scalability across an entire process. It even load balances between multiple AppDomains. When many components that are performing concurrent fine-grained operations are loaded into a process, and they all use the thread pool, they will not compete with one another. Alternative designs where each component managed its own pool of threads would lead to sub-optimal usage of processors, and overcreation of threads and their associated resources, resulting in unsatisfactory machine load.

Blocking

35. Document latency expectations for your users.

There is no consistent way to describe the performance characteristics of managed APIs as a contract, aside from documentation. When writing concurrent software, however, it's very important for developers to understand and reason about the performance of the dependencies they choose to take, particularly when this code is run inside critical regions. This includes things such as knowing the probability of blocking—and, therefore, whether to try and mask latency by transferring work to a separate thread, overlapping I/O, and so forth—as well as the compute and memory intensiveness of the internal operations. Library documentation should explain expected behavior.

36. Use the asynchronous programming model (APM) to supply async versions of blocking APIs.

Particularly if you are building a feature that performs I/O or otherwise uses an API that offers an asynchronous programming model (APM) variant, you should consider also exposing an APM

variant of your own API. For example, if your API would spend a good portion of its execution time blocked waiting for synchronous I/O, those same customers who'd use asynchronous file I/O APIs will want some way to turn your library's I/O into asynchronous I/O. The only way they can do that is if you provide the APM variant, as described further in Chapter 8, Asynchronous Programming Models.

37. Always block using one of these existing APIs: Lock acquisition, `WaitHandle.WaitOne`, `WaitAny`, `WaitAll`, `Thread.Sleep`, or `Thread.Join`.

The CLR doesn't block in a straightforward manner. Blocking is an opportunity to run the message loop on STA threads, for example. Hosts are also notified, such that they can do necessary bookkeeping. P/Invoking to a blocking API completely bypasses this machinery, and the CLR will not have a chance to hook the call. If this API blocks but doesn't pump messages on an STA, for instance, cross apartment deadlocks, among other problems, could occur. Other infrastructure is likely to rely on the central wait routine to do other useful things. All library code ought to block using one of the officially supported mechanisms.

FURTHER READING

Brumme, C. AppDomains ("application domains"). Blog article, <http://blogs.msdn.com/cbrumme/archive/2003/06/01/51466.aspx> (2003).

Cwalina, K., Abrams, B. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Addison-Wesley, 2005).

Duffy, J. Atomicity and Asynchronous Exceptions. Blog article, <http://www.bluebytesoftware.com/blog/2005/03/19/AtomicityAndAsynchronousExceptionFailures.aspx> (2005).

Duffy, J. Broken Variants of Double-checked Locking. Blog article, <http://www.bluebytesoftware.com/blog/2006/01/26/BrokenVariantsOnDoublecheckedLocking.aspx> (2006).

Duffy, J. No more hangs: Advanced techniques to avoid and detect deadlocks in .NET apps. *MSDN Magazine* (2006).

- Duffy, J. Application Responsiveness: Using Concurrency to Enhance User Experiences. *Dr. Dobb's Journal* (2006).
- Olukotun, K., Hammond, L. The Future of Microprocessors. *ACM Queue*, Vol. 3, No. 7 (2005).
- Sutter, H., Larus, J. Software and the Concurrency Revolution. *ACM Queue*, Vol. 3, No. 7 (2005).

B

Parallel Extensions to .NET

MICROSOFT'S NEW PARALLEL Extensions to the .NET Framework technology aims to evolve concurrent programming substantially by providing four major pillars of new concurrency functionality to .NET.

1. A collection of task oriented APIs, called the task parallel library (TPL), enabling you to manage lightweight tasks that are efficiently scheduled by a runtime that uses work stealing techniques of the kind alluded to earlier in this book. A rich task object model is available, in addition to helper classes with common imperative data parallel operations like parallel for loops.
2. A data parallel implementation of .NET's language integrated Query (LINQ). The Parallel LINQ (PLINQ) query provider takes any LINQ-to-Objects query over in memory data structures and auto-parallelizes it by indirectly using TPL.
3. A rich collection of synchronization primitives that encapsulate common coordination patterns. These extend the basic condition variable and events provided by the platform, as discussed back in Chapter 6, Data and Control Synchronization.
4. A set of concurrent collections, of the kind we reviewed in Chapter 12, Parallel Containers. These are the `System.Collections.Generic` equivalent for concurrent .NET programs.

Because Parallel Extensions is currently in “preview” status, everything shown in this chapter is apt to change. The content is roughly based on the June 2008 Community Technology Preview (CTP). The latest available release can be downloaded from <http://msdn.microsoft.com/concurrency/>.

Let’s look at each of these four pillars in more depth.

Task Parallel Library

The unit of concurrency in TPL is a `Task` object. This class offers many useful capabilities and, like most of TPL’s other classes, can be found in the `System.Threading.Tasks` namespace.

```
public class Task : TaskBase, IAsyncResult,
    IDisposable, ISupportsCancellation
{
    // Constructors

    public Task(Action action);
    public Task(Action<object>, object state);
    public Task(Action action, TaskManager taskManager);
    public Task(Action action, TaskCreationOptions options);
    public Task(
        Action action,
        TaskManager taskManager,
        TaskCreationOptions options
    );
    public Task(
        Action<object> action,
        object state,
        TaskManager taskManager,
        TaskCreationOptions options
    );

    // Static factory methods

    public static Task StartNew(Action action);
    public static Task StartNew(Action<object>, object state);
    public static Task StartNew(Action action, TaskManager taskManager);
    public static Task StartNew(
        Action action,
        TaskCreationOptions options
    );
}
```

```
public static Task StartNew(
    Action action,
    TaskManager taskManager,
    TaskCreationOptions options
);
public static Task StartNew(
    Action<object> action,
    object state,
    TaskManager taskManager,
    TaskCreationOptions options
);
// Methods

public void Cancel();
public void CancelAndWait();
public bool CancelAndWait(int millisecondTimeout);
public bool CancelAndWait(TimeSpan timeout);

public Task ContinueWith(Action<Task> action);
public Task ContinueWith(
    Action<Task> action,
    TaskContinuationKind kind
);
public Task ContinueWith(
    Action<Task> action,
    TaskContinuationKind kind,
    TaskCreationOptions options
);
public Task ContinueWith(
    Action<Task> action,
    TaskContinuationKind kind,
    TaskCreationOptions options,
    bool executeSynchronously
);
public void Dispose();

public void Start();

public void Wait();
public bool Wait(int millisecondsTimeout);
public bool Wait(TimeSpan timeout);

public static void WaitAll(params Task[] tasks);
public static bool WaitAll(Task[] tasks, int millisecondsTimeout);
public static bool WaitAll(Task[] tasks, TimeSpan timeout);
public static void WaitAny(params Task[] tasks);
public static bool WaitAny(Task[] tasks, int millisecondsTimeout);
```

```
public static bool WaitAny(Task[] tasks, TimeSpan timeout);

// Properties

public static Task Current { get; }

public Exception Exception { get; }
public int Id { get; }
public bool IsCanceled { get; }
public bool IsCancellationRequested { get; }
public bool IsCompleted { get; }
public Task Parent { get; }
public TaskStatus Status { get; }
public TaskCreationOptions TaskCreationOptions { get; }

}
```

The first aspects to Task you'll notice are the constructors and static `StartNew` factory methods. Both offer the same overloads; the `StartNew` methods are just shortcuts for the common operation of constructing a new task and immediately invoking its `Start` method. This is what most people will do: creating and starting a task as two independent operations is not nearly as common as doing both at once.

There are four parameters that show up in the overloads.

- An `action` must be given for every new task. This is a delegate that will be run once the task actually gets run. Some overloads accept an `Action` delegate—which has a `void` return type and accepts no parameters—while others accept an `Action<object>` delegate—which has a `void` return type but accepts a single parameter of type `object`.
- Optionally, an `object state` argument can be supplied. This is for those overloads that take an `Action<object>` and, as you probably guessed, the value is passed through to the delegate as its sole argument.
- A `TaskManager` object may be supplied. We'll save the discussion of `TaskManagers` for a few pages. In a nutshell, they offer the ability to isolate tasks generated by different components in the same process from one another, and also allow different policies to be applied. If one is not explicitly supplied, the default per `AppDomain` `TaskManager` is used.

- The `TaskCreationOptions` enum offers ways to change the default behavior of a task. This is a flags enum, so any of these options can be combined together: `None` (the default), `SuppressExecutionContextFlow`, `RespectParentCancellation`, `SelfReplicating`, `Detached`, and `UnhandledExceptionsAreFatal`. The `SuppressExecutionContextFlow` flag is much like the thread pool's `UnsafeQueueUserWorkItem`, in that it will prevent flowing of the `ExecutionContext` (and hence `SecurityContext`); this saves a bit of overhead for programs that only run in full trust. We will encounter the specific meaning of the other options throughout this appendix.

When a task is started, it is made available for execution. There is no guarantee when it will run. This is much like the thread pool's `QueueUserWorkItem` method. Underlying TPL is a very sophisticated scheduler that does a better job than the CLR's thread pool at managing resources intelligently, particularly for newer architectures and NUMA memory hierarchies. This includes using more scalable work stealing queues to manage tasks. This improves scalability because a lock free container type (such as the one shown in Chapter 12, Parallel Containers) can be used for tasks queued from scheduler threads. For tasks queued from nonscheduler threads, they go into a roughly-FIFO global queue protected by traditional locking. When a scheduler thread finishes running a task, it can consult its local task queue first: this avoids memory and global queue lock contention; if that fails, the scheduler thread tries stealing from surrounding queues; only if that also fails will the global queue be consulted. The preference for going to its own queue leads to roughly LIFO task dispatch ordering.

The static `Current` property can be accessed from within the delegate to retrieve the currently executing `Task` object. If there is none, it returns `null`. The `Id` instance property generates a unique identifier and returns it and can be useful in debugging and diagnostics. Finally, the `Status` property fetches a snapshot of what the task is currently doing. The returned value will be one of these enum values: `Created`, `WaitingToRun`, `Running`, `Blocked`, `WaitingForChildrenToComplete`, `RanToCompletion`, `Canceled`, or `Faulted`. All tasks begin life as `Created` and move into `WaitingToRun`.

once `Start` is called. If you use the `StartNew` factory method, you'll only see tasks created in the `WaitingToRun` state. When the task begins executing (usually because a scheduler thread has awakened and begun running it), the task moves into the `Running` state; if it blocks by doing a wait of any sort, it will be moved into the `Blocking` state and then transition back to `Running` when it wakes back up (similar to `Thread`'s `WaitSleepJoin` state). The `WaitingForChildrenToComplete` state will make more sense below when we discuss **structured tasks**. The last three states are final: `RanToCompletion` means the task's delegate executed to completion, `Canceled` means a cancellation request was successful (more on that later), and `Faulted` means the task's delegate threw an unhandled exception. The `IsCanceled` property is just a shortcut for checking for `Canceled`, and `IsCompleted` is a shortcut for checking for any of the final three states.

Once you've created a task, there may come a point where you need to wait for it to complete. Perhaps this is because the task is creating a value of interest, and the program has reached a point where it can make no more useful progress until that value is known. Whatever the case, the `Task` class provides the instance `Wait` method, and the static `WaitAll` and `WaitAny` methods for this purpose. Their functionality is self explanatory: `Wait` waits for a single `Task` to enter into a final state, `WaitAll` waits for all of the `Task` objects in an array to do the same, and `WaitAny` waits for a single `Task` in the supplied array (returning an index into the one which completed). All offer `int` and `TimeSpan` based timeout overloads.

Interestingly, a call to wait on a task might not block, even if that task hasn't finished running. The reason is that under some circumstances (such as running on a scheduler thread), TPL can manually dequeue the task and **inline** it. That means the task is run on the current thread inside the call to wait on it. For recursive divide and conquer style problems this is great; otherwise, you'd need to be very precise about when you switch over to sequential recursion in order to avoid creating a ridiculous number of blocked threads. From the task's point of view, it is being run on a scheduler thread and it generally can't tell that it was inlined. The one thing to be careful about is TLS and thread-affinity at the point of a call to wait on a task: for example, if a CLR monitor is held when a call to wait is made, the inlined task may freely acquire it recursively. This will undoubtedly lead to some surprises.

Most of the other APIs available on the `Task` class are described in detail later. Each family of methods is sufficiently interesting to warrant its own section.

Unhandled Exceptions

TPL automatically catches all unhandled exceptions thrown from task delegates. A task with an unhandled exception enters into the `Faulted` final state, and its `Exception` property provides access to the exception that tore it down. Any waits on the `Task` will be immediately satisfied, and the exception will be repropagated by the call to `Wait`. If a task fails in this way and the exception goes unobserved—in other words, nobody accesses the `Exception` property or calls `Wait` on the task—something unpleasant will happen: TPL will rethrow the exception on your finalizer thread, crashing it. The debugging experience for this is not ideal, because the exception will appear to have originated from a finalizer that TPL controls. But this situation indicates a severe bug in the program. An unhandled exception that is never witnessed is a severe error that may indicate state corruption and that the program is failing; it should never be ignored, and TPL ensures this is so.

This behavior is meant to provide a sequential programming-like appearance for exception handling. In most structured parallelism cases (which we'll discuss more soon), functions create and wait on tasks inside of a well defined scope; preserving exception propagation across asynchronous points in this manner can be useful. In other cases, however, a task will be created and forgotten: this is sometimes called **fire and forget**. Similarly, many tasks have been written so that no unhandled exceptions are expected. To improve debugging, you may pass the `UnhandledExceptionsAreFatal` flag when creating your task. This suppresses TPL's automatic marshaling of exceptions.

Because the definition of concurrency implies multiple things are happening at once, it also means that multiple things may fail at once. This fundamentally impacts the way exceptions are treated in TPL and the entire Parallel Extensions library. We saw this in Chapter 13, Data and Task Parallelism. The practical implication is that all exceptions are exposed as `AggregateException` objects, each of which is a collection of one or more

other exceptions. `AggregateException` is a basic exception class with three unique aspects:

- The `InnerExceptions` property returns a `ReadOnlyCollection<Exception>` containing each of the unhandled exception objects.
- Because of recursive concurrency, the individual exceptions within this collection can themselves also be `AggregateException` objects. This can lead to an unmanageable amount of nesting. Calling the `Flatten` method will return a new `AggregateException`, which recursively “flattens” the whole tree. For each exception, it pulls out the `InnerExceptions` recursively, until there are no aggregates left. You are left with a single `AggregateException` that has no other aggregates within.
- This kind of aggregation fundamentally changes exception handling. No longer can you catch a specific exception. Instead, you catch `AggregateException`, look for certain kinds of exceptions within, and repropagate if you can’t handle them all. The `Handle` method encapsulates this common pattern. It accepts a `Func<Exception, bool>`; it iterates over all `InnerExceptions`, runs the predicate against each, and, if the function returned `true` for all of them, will return. If there was a single `false`, a new `AggregateException` is created (containing all exceptions for which the function returned `false`), and this is thrown out of the `Handle` method.

Imagine we have a function `f` that calls another function `g` sequentially. The function `g` may throw a `FooException`, and `f` knows how to handle it. If any other kind of exception were thrown out of `g`, however, `f` would let it go unhandled. We would write this as:

```
void f()
{
    try
    {
        g();
    }
    catch (FooException fe)
    {
        // S(fe) handles the exception.
        // We then swallow it.
    }
}
```

```
void g()
{
    if (...) throw new FooException();
    ...
}
```

If we were to instead invoke `g` from within a TPL task and `f` waited on it, we would need to do something special for exception handling. The call `f` makes to `Wait` will now result in an `AggregateException` if an exception were thrown. We'd write this as follows.

```
void f()
{
    try
    {
        Task.StartNew(() => g()).Wait();
    }
    catch (AggregateException ae)
    {
        ae.Handle(e =>
        {
            FooException fe = e as FooException;
            if (fe != null)
            {
                // S(fe) handles the exception.
                return true;
            }
            return false;
        });
    }
}

void g()
{
    if (...) throw new FooException();
    ...
}
```

Parents and Children

By default, tasks created from within other tasks will form parent/child trees. A task `B` that is created within another task `A` will become `A`'s child (and similarly `A` becomes `B`'s parent). The `Parent` property retrieves this information at runtime and comes in handy for debugging. There is no equivalent property to fetch the list of running children. For example, this code snippet illustrates this particular situation.

```
Task taskA = Task.StartNew(delegate
{
    Task taskB = Task.StartNew(...);
    // assert(taskB.Parent == Task.Current);
    ...
});
```

We say that such tasks are **structured** because TPL enforces the hierarchy. This means that TPL will not consider a parent finished until all of its outstanding children have also finished. It's as if a parent always implicitly waits on its children before completing. (This also means that when you wait on a parent of a structured task tree, you're also implicitly waiting on all of its children.) This snippet illustrates a simplistic implementation of this idea.

```
Task taskA = Task.StartNew(delegate
{
    try {
        Task taskB = Task.StartNew(...);
        ...
    } finally {
        taskB.Wait(); // Imaginary (implicit).
    }
});
```

Things are more complicated than this due to unhandled exceptions (as we'll see soon), but as a mental model, this isn't too far from reality. Structured tasks are useful because having a well defined scope where concurrency begins and ends, as mentioned in Chapter 1, Introduction, can help reduce the occurrence of hazards such as race conditions. This approach also guarantees that exceptions from children are always propagated up the ancestor hierarchy such that a thread that waits on the topmost task will see them all. As the exceptions make their way up the hierarchy, the aggregation can become deep. This is an example of why `AggregateException`'s `Flatten` method can be very useful.

That said, **unstructured concurrency** is sometimes necessary, and TPL provides this capability. In this model, children are permitted to survive their parent task. Unstructured tasks are opt in instead of being the default: pass the `Detached` option at task creation time.

```
Task taskA = Task.StartNew(delegate
{
    Task taskB = Task.StartNew(..., TaskCreationOptions.Detached);
    // assert(taskB.Parent != Task.Current);
    ...
});
```

In this example, task A will not automatically wait for B to finish, and B's Parent property will return null as though it were created in a situation where there was no active task.

Cancellation

TPL offers first class cancellation through the Cancel and CancelAndWait functions. When called on a task, the runtime first checks to see if it has begun running. If not, the task will never run: it is effectively removed from the scheduler's queue, and its state immediately transitions to the final Canceled state. Otherwise, the task's IsCancellationRequested flag is set to true. The point of this flag is to enable cooperative cancellation if a task begins running and is then asked to cancel itself, as we saw in Chapter 13, Data and Task Parallelism.

If a task is canceled, any calls to Wait will awaken with an AggregateException containing a single TaskCanceledException. This is a basic exception class that also offers a Task property to indicate which particular task was canceled.

Another useful aspect to using structured parallelism is that cancellation requests may be automatically flowed through a hierarchy of tasks. By default, this does not occur, but by specifying the RespectParentCancellation flag at task creation time, a child task will inherit its parent cancellation flag. (Note that detached tasks do not flow the cancellation flag, no matter whether the option is specified or not.) This feature is opt in because any task that can be canceled must be treated specially: all Wait call sites must be hardened to be correct in the face of unexpected cancellation exceptions. For systems that need cancellation (most notably GUI driven applications), the ability to flow cancellation this way can be a great feature.

Futures

Tasks run actions, but the programming model doesn't require that they produce a result. It's somewhat common for a task's "result" to be the set of side effects that it performs. But it's also common for a task to produce a real value and for other tasks in the system to need to consume this value. In this case, extra storage and synchronization is needed with the basic Task APIs in order to communicate the resulting value to interested parties.

The Future<T> class offers intrinsic support for this commonly needed capability: an instance is merely a task that produces a value of type T.

```
public sealed class Future<T> : Task
{
    // Constructors

    public Future();
    public Future(Func<T> valueSelector);
    public Future(Func<T> valueSelector, TaskManager taskManager);
    public Future(Func<T> valueSelector, TaskCreationOptions options);
    public Future(
        Func<T> valueSelector,
        TaskManager taskManager,
        TaskCreationOptions options
    );

    // Static factory methods

    public static Future<T> StartNew();
    public static Future<T> StartNew(Func<T> valueSelector);
    public static Future<T> StartNew(
        Func<T> valueSelector,
        TaskManager taskManager
    );
    public static Future<T> StartNew(
        Func<T> valueSelector,
        TaskCreationOptions options
    );
    public static Future<T> StartNew(
        Func<T> valueSelector,
        TaskManager taskManager,
        TaskCreationOptions options
    );

    // Methods

    public Task ContinueWith(Action<Future<T>> action);
    public Task ContinueWith(
```

```
Action<Future<T>> action,
TaskContinuationKind kind
);
public Task ContinueWith(
    Action<Future<T>> action,
    TaskContinuationKind kind,
    TaskCreationOptions options
);
public Task ContinueWith(
    Action<Future<T>> action,
    TaskContinuationKind kind,
    TaskCreationOptions options,
    bool executeSynchronously
);
public Future<U> ContinueWith<U>(Func<Future<T>, U> func);
public Future<U> ContinueWith<U>(
    Func<Future<T>, U> func,
    TaskContinuationKind kind
);
public Future<U> ContinueWith<U>(
    Func<Future<T>, U> func,
    TaskContinuationKind kind,
    TaskCreationOptions options
);
public Future<U> ContinueWith<U>(
    Func<Future<T>, U> func,
    TaskContinuationKind kind,
    TaskCreationOptions options,
    bool executeSynchronously
);
// Properties

public Exception Exception { get; set; }
public T Value { get; set; }
}

public static class Future
{
    public static Future<T> StartNew<T>();
    public static Future<T> StartNew<T>(Func<T> valueSelector);
    public static Future<T> StartNew<T>(
        Func<T> valueSelector,
        TaskManager taskManager
    );
    public static Future<T> StartNew<T>(
        Func<T> valueSelector,
        TaskCreationOptions options
    );
    public static Future<T> StartNew<T>(

```

```
    Func<T> valueSelector,
    TaskManager taskManager,
    TaskCreationOptions options
);
}
```

There isn't much to a `Future<T>` besides what it inherits from the `Task` base class. It has some constructors (which look a lot like `Task`'s), and there are a lot of new static factory methods. The primary difference is that instead of `Action` delegates, these accept `Func<T>` delegates: this is typed as returning a value of type `T`. There is also a nongeneric `Future` class to make type inference based creation easier. For example, in C# 3.0 and beyond you can create a new `Future<T>` without having to explicitly state the type argument for `T`.

```
var myFuture = Future.Create(() => int.MaxValue);
```

In the above snippet, the `myFuture` variable ends up correctly typed as a `Future<int>`.

When a `Future<T>` finishes, the value returned from its delegate ends up accessible from the `Value` property. Any accesses to retrieve this value will block waiting for it to be bound (if it hasn't been already) and then return the value. Much like the `Wait` API, any unhandled exceptions will be repropagated during accesses to `Value`.

You may have noticed a few strange things here: there is a constructor (and corresponding `StartNew` overloads) that doesn't accept any `Func<T>`. Moreover, the `Exception` and `Value` properties have public set methods. This is a feature often called a **promise style future**, because the future itself is a promise for a value, but there is no tie-in to the scheduler itself. You cannot `Start` such a future. Some thread must later explicitly set the appropriate property (`Exception` if something wrong happens, or `Value` otherwise), and it will behave just as if the scheduler were responsible for doing so. In other words, task state transitions will occur as expected, threads waiting for results will be awoken, and so forth.

Continuations

The `ContinueWith` methods on `Task` and `Future<T>` are meant to offer an alternative to waiting. Instead of waiting (which can block a thread), you can

instead register an action to be performed once the target task enters a final state. This “promise” to invoke an action later on itself manifests as yet another task, meaning you can wait on it and so on. This task is not necessarily started when returned, however; the TPL continuation implementation will call `Start` on it sometime later. (`ContinueWith` handles the race condition in which a task completed before the call to `ContinueWith`; in this case, it is possible for the continuation task to have already been started, or even begun running, before it is returned.) A wonderful thing about this is that you can create a string of continuations that are dependent on one another, and at the end of doing so you will have a single `Task` handle to the whole chain.

The relatively obscure parameter `executeSynchronously` controls whether the continuation should be run asynchronously in the scheduler (the default) or synchronously whenever the task completes. The only purpose for this is to avoid overhead when the continuation is a very quick action, like setting a flag or event, for instance.

By default, a task’s continuation will fire no matter the final state of the task. You can, however, specify a `TaskContinuationKind` flags enum value to limit the final states in which the continuation will become active: `OnRanToCompletion`, `OnCanceled`, or `OnFaulted`. (The default is equivalent to `OnRanToCompletion | OnCanceled | OnFaulted`.) If the task eventually transitions into a final state that wasn’t part of the continuation’s activation criteria, the continuation `Task` object will be canceled. This may cause continuations of that continuation (registered with `OnCanceled`) to fire, and so on.

The `Future<T>` class also provides some unique overloads of `ContinueWith` that enable you to access the future’s value inside the callback, and / or return another `Future<U>` object. This allows for some very simple chaining of dataflow operations. For example:

```
Future<string> fs =
    Future<int>.StartNew(...).
    ContinueWith<DateTime>(v => ... v.Value ...).
    ContinueWith<string>(v => ... v.Value ...);
...
string realValue = fs.Value;
```

Notice that the `ContinueWith` callbacks access the `Value` property of the future. This ensures that exceptions will propagate through the entire

continuation chain. If any of the futures in the chain fails, then the eventual call to `fs.Value` will propagate the exception(s).

Task Managers

As was mentioned in Chapter 7, Thread Pools, one of the weaknesses of traditional thread pools is that they offer no way to assign policy and establish some degree of isolation between different components in the same process. Recall that the Windows Vista thread pool now offers a solution to this, by enabling you to manage multiple pools. Well, TPL's `TaskManager` abstraction is meant to do precisely this. By instantiating and creating tasks that are bound to different task managers, you have explicit control over policy and isolation; the underlying scheduler semi-fairly services all managers in the process, so you know that one chatty component can't unfairly starve another component that only occasionally generates work.

The `TaskManager` and related `TaskManagerPolicy` classes are simple.

```
public class TaskManager : IDisposable
{
    public TaskManager();
    public TaskManager(TaskManagerPolicy policy);

    public void Dispose();

    public static TaskManager Current { get; }
    public static TaskManager Default { get; }

    public TaskManagerPolicy Policy { get; }
}

public class TaskManagerPolicy
{
    public TaskManagerPolicy();
    public TaskManagerPolicy(int maxStackSize);
    public TaskManagerPolicy(int minProcessors, int idealProcessors);
    public TaskManagerPolicy(
        int minProcessors,
        int idealProcessors,
        int idealThreadsPerProcessor
    );
    public TaskManagerPolicy(
        int minProcessors,
        int idealProcessors,
        ThreadPriority threadPriority
    );
}
```

```
public TaskManagerPolicy(
    int minProcessors,
    int idealProcessors,
    int idealThreadsPerProcessor,
    int maxStackSize,
    ThreadPriority threadPriority
);

public int IdealProcessors { get; }
public int IdealThreadsPerProcessor { get; }
public int MaxStackSize { get; }
public int MinProcessors { get; }
public ThreadPriority ThreadPriority { get; }
}
```

The `TaskManager` class can be constructed with no-arguments or with a specific `TaskManagerPolicy` object. The former uses the default policy settings. The static `Current` property retrieves the active `TaskManager`, and `Default` retrieves the default AppDomain-wide manager, which will be used if not overridden at task creation time. Aside from creating a new one and accessing its `Policy` object, you can call `Dispose` on it. This call synchronously shuts down the scheduler and waits for it to complete. This may take some time because scheduler resources can only be freed once all current tasks finish executing.

The `TaskManagerPolicy` class provides several interesting settings and a lot of constructor overloads for common combinations of settings.

- `IdealProcessors`: This instructs the scheduler how many processors it should attempt to maximize usage of. The default is equal to the number of processors on the machine (i.e., `Environment.ProcessorCount`).
- `IdealThreadsPerProcessor`: This tells the scheduler how many threads per processor it should optimize for. The default is 1; in other words, it is optimized for compute-bound workloads. If the task manager is meant to run workloads that frequently block, however, it is a good idea to experiment with values greater than 1.
- `MinProcessors`: This tells the scheduler what the minimum number of processors to utilize is. Because the scheduler contains intelligent resource management algorithms, it may otherwise have decided to use fewer than these processors. But if you want to increase the fairness among long running pieces of work, specifying a value here can be useful.

- **MaxStackSize:** By default, just as with thread creation, scheduler threads will be created with the default stack size inherited from the executable. (See Chapter 4, Advanced Threads.) If you specify a value here, however, threads will be created with *at least* the MaxStackSize you have specified.
- **ThreadPriority:** Threads in the scheduler run with a normal priority. This is usually what you want. But if you'd prefer to run threads with lower priority (because, for example, tasks in this particular manager are meant to do "background" work) or higher priority (which is dangerous, for all the reasons outlined in Chapter 11, Concurrency Hazards), you may override the policy.

Once you've got a fully constructed `TaskManager`, you can pass it as an argument to many interesting APIs. That mostly means the various constructors and `StartNew` methods on `Task`, `Future<T>`, and `Future`.

Putting it All Together: A Helpful Parallel Class

Being able to use tasks directly is wonderful. The TPL task abstraction offers some very rich capabilities. However, there are some common patterns of structured usage that are also provided, raising the level of abstraction dramatically. We saw in Chapter 13, Data and Task Parallelism, that data parallelism is a common way of attaining improved performance on parallel processors. We also saw that fork/join structured parallelism is extremely common. Hand coding these with the `Task` class is possible, but there is a simpler way.

The static `Parallel` class in the `System.Threading` namespace offers implementations of three common operations: `for` loops with the `For` method (which supports both 32-bit and 64-bit indices), `foreach` loops with the `ForEach` method (over `IEnumerable<T>` objects), and fork-join with the `Invoke` method.

```
public static class Parallel
{
    public static ParallelLoopResult For(
        int fromInclusive,
        int toExclusive,
        Action<int> body
```

```
);

public static ParallelLoopResult For(
    int fromInclusive,
    int toExclusive,
    int step,
    Action<int, ParallelState> body,
    TaskManager taskManager,
    TaskCreationOptions options
);

public static ParallelLoopResult For<TLocal>(
    int fromInclusive,
    int toExclusive,
    int step,
    Func<TLocal> threadLocalInit,
    Action<int, ParallelState<TLocal>> body,
    Action<TLocal> threadLocalFinally,
    TaskManager taskManager,
    TaskCreationOptions options
);

// Many overloads of For omitted.

public static ParallelLoopResult For(
    long fromInclusive,
    long toExclusive,
    Action<long> body
);
public static ParallelLoopResult For(
    long fromInclusive,
    long toExclusive,
    long step,
    Action<long, ParallelState> body,
    TaskManager taskManager,
    TaskCreationOptions options
);
public static ParallelLoopResult For<TLocal>(
    long fromInclusive,
    long toExclusive,
    long step,
    Func<TLocal> threadLocalInit,
    Action<long, ParallelState<TLocal>> body,
    Action<TLocal> threadLocalFinally,
    TaskManager taskManager,
    TaskCreationOptions options
);

// Many overloads of For64 omitted.

public static ParallelLoopResult ForEach<TSource>(  
}
```

```
    IEnumerable<TSource> source,
    Action<TSource> body
);
public static ParallelLoopResult ForEach<TSource>(
    IEnumerable<TSource> source,
    Action<TSource, int, ParallelState> body,
    TaskManager taskManager,
    TaskCreationOptions options
);
public static ParallelLoopResult ForEach<TSource, TLocal>(
    IEnumerable<TSource> source,
    Func<TLocal> threadLocalInit,
    Action<TSource, int, ParallelState<TLocal>> body,
    Action<TLocal> threadLocalFinally,
    TaskManager taskManager,
    TaskCreationOptions options
);
// Many overloads of ForEach omitted.

public static void Invoke(params Action[] actions);
public static void Invoke(
    Action[] actions,
    TaskManager manager,
    TaskCreationOptions options
);
}
```

Each of these APIs offers several overloads to accommodate slightly different ways in which they can be used. For example, each of the different APIs offers a way to plug in a custom `TaskManager` and set of `TaskCreationOptions`. Many, many overloads have been omitted to save space; instead, the simplest and most general purpose are shown. All of these APIs are structured, however, meaning that the tasks they generate internally will have completed before the time the API returns. This ensures that any exceptions thrown from actions invoked within are propagated correctly out of the call to the specific method.

The goal of the `For` API is to allow easy replacement of existing `for` loops, and similarly with `ForEach`, to allow easy replacement of existing `foreach` loops. They take a simple `Action<T>` delegate, where `T` is `int` for the 32-bit overloads, `long` in the case of the 64-bit overloads, and `TSource` in the case of `ForEach<TSource>`.

For example, given some existing sequential code with a few loops in it:

```
for (int i = 0; i < N; i++) A(i);
for (int j = 0L; j < M; j++) B(j);
List<T> lst = ...;
foreach (T e in lst) C(e);
```

We can easily transform this into the corresponding parallelized version.

```
Parallel.For(0, N, i => A(i));
Parallel.For(0L, M, j => B(j));
List<T> lst = ...;
Parallel.ForEach(lst, e => C(e));
```

The use of C# 3.0 lambda syntax makes the transformation from sequential to parallel elegant and helps to minimize the differences. Of course, as we discussed in previous chapters, the fact that you can parallelize a loop such as this doesn't imply that you should. Functions A, B, and C, for example, must be able to tolerate being called in parallel. In fact, in the extreme, all iterations will be running in parallel. In practice, the realized parallelism will be limited by the machine's resources and current activity.

Each loop API provides an overload that accepts a `ParallelState` object as an argument to the action delegate. This can be used to voluntarily terminate the loop early, as with the `break` statement in ordinary `for` and `foreach` loops.

```
public class ParallelState
{
    public void Break();
    public void Stop();
    public bool ShouldExitCurrentIteration { get; }
}
```

Calling `Break` instructs the `Parallel` machinery to terminate the current loop once all previous iterations have finished. Unlike sequential loops, because other threads may be barging ahead, there is no guarantee that *subsequent* iterations have not run. They might have, although `Parallel` will try to cooperatively stop them from doing so. Multiple calls to `Break` will lead to the lowest iteration winning. Similarly, `Stop` halts the loop, but unlike `Break` it attempts to do so as soon as possible without regard for which iterations may have already run. Both methods use cooperative techniques to

shut down similar to those used for cooperative cancellation; in other words, there is no thread abort or interruption nonsense going on.

For and ForEach each return a ParallelLoopResult structure as their result. This contains information about whether a stop or break occurred, and if so, which iteration the break happened on.

Each of the kinds of loop APIs also offers a generic variant for having per thread state: For<TLocal> and ForEach<TSource, TLocal>. Because the loop will automatically replicate across the available hardware, multiple threads will be used. Sometimes thread local state is necessary due to the introduction of parallelism. Doing a TLS lookup in each loop iteration, however, is apt to have terrible performance. Instead, these overloads can be used: you provide an initialization routine that returns a TLocal object and, optionally, a finally routine that is meant to clean up. The body then has access to the TLocal via the ThreadLocalStorage property of the ParallelState<TLocal> object.

This feature can be used to isolate obviously thread unsafe things, such as database connections between parallel loop iterations, but can also be used to do clever tricks like implementing an efficient reduction procedure. Here's an example Sum API that does just that.

```
int Sum(int[] numbers) {
    int final = 0;
    Parallel.ForEach<int, int>(
        numbers,
        () => 0,
        (e, ps) => ps.ThreadLocalStorage += e,
        s => Interlocked.Add(ref final, s)
    );
    return final;
}
```

The Invoke API makes running a series of statements in parallel much easier, much like our fictional CoBegin API back in Chapter 13. For example, given a series of statements:

```
A();
B();
C();
```

We can easily transform this from sequential to parallel.

```
Parallel.Invoke(  
    () => A(),  
    () => B(),  
    () => C()  
)
```

As with the loops, this looks nice and elegant (again, thanks to C# lambdas) and should also be treated carefully because A, B, and C may run in parallel with one another.

Self-Replicating Tasks

The last TPL feature we'll explore is called self replication. You may have wondered how the `Parallel` class automatically scales to use up all of the available processors. It exploits the inexpensive recursive queueing nature of the work stealing queues by having the internal tasks recursively generate multiple copies of themselves. If one of these so called replicas happens to be stolen because a processor is free, it will be scheduled, queue its own replica, and continue finishing the operation. Once any one of the replicas quits, replication stops. This capability is not a common one but is mind bending enough that TPL provides a `SelfReplicating` option that can be specified at task creation time.

You could use this to create your own `While` loop API. For example:

```
public static void While(Func<bool> predicate, Action body)  
{  
    Task root = Task.Create(() =>  
    {  
        if (!predicate()) return;  
        body();  
    },  
    TaskCreationOptions.SelfReplicating);  
}
```

This particular example of course assumes several things. It assumes both `predicate` and `body` are thread safe. It may also continue to execute other replicas after `predicate` has returned `false` for the first time. Moreover, if `predicate` doesn't return `false` every subsequent time after it has

returned `false` once, there is no guarantee subsequent iterations will stop. But nevertheless, this illustrates the basic self-replicating functionality: the `While` loop will automatically scale to use as many processors as there are free via replication.

Parallel LINQ

Language integrated query (LINQ) allows developers to write declarative queries, either through a series of API calls to the `System.Linq.Enumerable` class, or by using the language comprehension syntax supported by languages like C# and VB. These queries can include powerful set based operations much like SQL: projections, filters, sorts, joins, groupings, searches, and more. Several different query providers are offered, including LINQ-to-Objects, an implementation that works over in-memory data structures such as arrays and lists. LINQ-to-XML allows querying of XML documents and builds on top of LINQ-to-Objects. A detailed overview of LINQ is outside of the scope of this book, but understanding LINQ to some level of detail is a prerequisite to understanding **parallel LINQ** (PLINQ).

The wonderful thing about LINQ is that it's **declarative**, meaning that the specification of the computation of results is sufficiently high level that the individual steps taken to produce the output are immaterial to you. This allows PLINQ to step in and automatically parallelize.

PLINQ works by analyzing the query, and arranging for different pieces to run in parallel with one another on multiple processors. It does this ultimately by using TPL under the covers. The complexity of the analysis done by PLINQ varies dramatically from query to query, and not every query will see a scalability gain when run under PLINQ versus LINQ. This depends on the complexity of the query, size of input data, and cost of the individual operations. For example, to do a join between two data sources, PLINQ must go out of its way to partition data specially; sorts do not scale linearly and will be a limiting factor; and so on.

Using PLINQ is actually very simple once you know how to use LINQ, so this section will be very light indeed. To use PLINQ, you make calls through the `System.Linq.ParallelEnumerable` class (instead of `Enumerable`). PLINQ supports all of the LINQ operators, and the only difference you will notice is that these operators accept `IParallelEnumerable<T>` rather than `IEnumerable<T>`.

objects. To produce an `IParallelEnumerable<T>`, you will use the `AsParallel` extension method on the `System.Linq.ParallelQuery` class.

```
public static IParallelEnumerable AsParallel(this IEnumerable source);
public static IParallelEnumerable<TSource> AsParallel<TSource>(
    this IEnumerable<TSource> source
);
public static IParallelEnumerable<TSource> AsParallel<TSource>(
    this IEnumerable<TSource> source,
    TaskManager taskManager
);
```

Notice there is also an overload for nongeneric `IEnumerable` objects. And there is also an overload of `AsParallel` that accepts a TPL `TaskManager`. This directs PLINQ to queue the resulting `Task` objects that it creates into that manager. The `AsParallel` API works nicely with comprehensions, so you don't need to explicitly call the `ParallelEnumerable` interface at all. If you turn your `IEnumerable<T>` into an `IParallelEnumerable<T>` and use extension methods or comprehensions, PLINQ will be chosen over LINQ. Here is an example of a LINQ query, written three ways.

```
IEnumerable<T> source = ...;

// Variant 1:
IEnumerable<U> q1 = Enumerable.Select<T, U>(
    Enumerable.Where<T>(source, x => p(x)),
    x => f(x)
);

// Variant 2:
IEnumerable<U> q2 = source.
    Where<T>(x => p(x)).
    Select<T, U>(x => f(x));

// Variant 3:
var q3 = from x in source where p(x) select f(x);
```

Now here are those same three variants written to use PLINQ.

```
IEnumerable<T> source = ...;

// Variant 1:
IParallelEnumerable<U> q1 = ParallelEnumerable.Select<T, U>(
    ParallelEnumerable.Where<T>(
        ParallelEnumerable.AsParallel<T>(source), x => p(x)),
    x => f(x)
);
```

```
);  
  
// Variant 2:  
IParallelEnumerable<U> q2 = source.AsParallel().  
    Where<T>(source, x => p(x)).  
    Select<T, U>(x => f(x));  
  
// Variant 3:  
var q3 = from x in source.AsParallel() where p(x) select f(x);
```

Although it's simple to use PLINQ, it must be done with care, as with `Parallel.For` and other parallel APIs, your operators are run in parallel, meaning any accesses to shared state from the delegates passed into PLINQ may result in race conditions.

There are also corresponding `AsMerged` methods that turn an `IParallelEnumerable<T>` back into an `IEnumerable<T>`. This can be used to force a portion of a PLINQ query to go through LINQ in case that portion relies on shared state or where parallelism has a negative performance impact. In addition to that, `AsMerged` allows you to control the kind of buffering used by PLINQ. We'll explore buffering and merging next.

Buffering and Merging

When you create a query as shown above with `q1`, `q2`, and `q3`, it has not yet begun running. Execution of queries is *lazy* and will be deferred until you actually begin consuming the output. That occurs on demand when you `foreach` over the query, upon the first call to `MoveNext` on the result of `GetEnumerator`, or if you use a LINQ API like `ToDictionary`, `ToDictionary`, and so forth. Any exceptions that occur during the execution of your query will, therefore, be thrown only when you've begun consuming the output of the query. As with TPL, PLINQ exceptions are aggregated using the same `AggregateException` type.

The enumerator used to access the results of a query's execution needs to perform interthread coordination to get results from the concurrently running tasks. This is called **merging** and is the opposite of **partitioning**, which is what the query does initially to feed different portions of the input to different tasks. PLINQ goes out of its way to make sure these two operations are as efficient as possible since they are largely the only parts that internally require a lot of synchronization.

(and, hence, can become scalability bottlenecks). For example, PLINQ will do a far better job partitioning `IList<T>` objects because they support random access; given any other `IEnumerable<T>`, PLINQ needs to serialize some portion of access to a shared enumerator. One technique PLINQ uses to make the merge phase more efficient is to buffer elements as much as possible by default.

Three kinds of merges are possible. You can control which is chosen by passing a `ParallelMergeOptions` value to the `AsMerged` API.

1. `AutoBuffered`, a.k.a. pipelined with automatic buffering. In this mode, which is the default for most queries, the thread consuming elements from the enumerator run concurrently with the query. As elements are generated by the query, they are handed over to the enumerator. To amortize the associated synchronization overhead, PLINQ will use some amount of buffering. This also increases the latency for an element to be handed to the consumer, however, which could cause troubles if low latency is desired.
2. `NotBuffered`, a.k.a. pipelined with no buffering. This mode is similar to the first in that the consumer runs concurrently with the query. But unlike the first mode, elements are not buffered. This reduces latency for an element to reach the consumer, but at the expense of more synchronization overhead. For queries in which the cost of per element production is high, this can be appropriate.
3. `FullyBuffered`, a.k.a. stop-and-go. This mode allows PLINQ to avoid per element (or per buffer) synchronization when handing off elements to the consumer. When execution of the query is triggered, the query will only return once the full output is available. The calling thread is used to run part of the query. This increases the latency to retrieve the first result, but is the most efficient mode PLINQ offers in terms of execution time. This mode can increase memory usage, however, because the full output needs to be held in memory.

For most uses of PLINQ, sticking to the default is wise. That usually means `AutoBuffered`, but some things may trigger PLINQ to switch over to `FullyBuffered`. This happens if PLINQ would only be able to return the

first element once the full output was known anyway, which includes the `OrderBy` operator and APIs like `ToArray`.

Order Preservation

Because PLINQ runs in parallel, the elements fed into a query may become scrambled during execution. The symptom of this is that order among elements in the output may not directly correspond to the elements in the input. As a simple example of this, there is no guarantee that `a` and `b` will be equal after the following snippet is run

```
int[] a = new int[] { 0, 1, 2, 3, 4, 5 };
int[] b = (from x in a.AsParallel() select x).ToArray();
```

On one hand, this seems absurd. The query maps the identity function against all elements in the array. But if you stop to think about all of the partitioning and merging going on in order to do that mapping in parallel, it would require PLINQ to expend a considerable amount of effort in order to preserve the input ordering.

For many problems this is acceptable. In fact, because of LINQ's set oriented and SQL-like nature, many people don't expect order to be preserved by LINQ itself. But if this does matter to your problem, you can force PLINQ to preserve the ordering in its output with the `AsOrdered` API. As noted above, this comes at some expense, which is why it is opt in.

```
public static IParallelEnumerable<T> AsOrdered<T>(
    this IParallelEnumerable<T> source
);
```

The only legal position for `AsOrdered` is when immediately preceded by an `AsParallel`. The API will throw an exception otherwise. So if we wanted to force order preservation on our example above, it would look like this:

```
int[] a = new int[] { 0, 1, 2, 3, 4, 5 };
int[] b = (from x in a.AsParallel().AsOrdered() select x).ToArray();
```

There is also an `AsUnordered` API that can be used in the middle of a query to turn off ordering for a particular set of operators. This can be used with operators like `Take` that have a deeply ingrained notion of order. For instance, if your query contains `Take(1000)`, you presumably care about it taking the

first 1,000 elements. That requires use of `AsOrdered`. But perhaps once you've taken those 1,000 elements, you don't want to pay the cost of order preservation for all subsequent operators; this is particularly true of the merge step, whose performance order preservation can impact dramatically.

Synchronization Primitives

Parallel Extensions provides several useful synchronization primitives to support common data and control synchronization needs. Several of these will be familiar to you if you've read the whole book up to this point.

ISupportsCancellation

The `System.Threading.ISupportsCancellation` interface indicates that some class supports object level cancellation. Canceling such an object will immediately wake up all threads that are blocked on it. This is useful when some thread participating in an operation fails to reach a synchronization point or in support of responsive GUIs that need to be able to tear down potentially lengthy parallel computations at the request of the end user.

The interface itself is very straightforward.

```
public interface ISupportsCancellation
{
    void Cancel();
    bool IsCanceled { get; }
}
```

You'll notice that TPL's `Task` class implements this interface, as do many of the types we're about to see. Though simple, this interface allows general purpose cancellation frameworks to be built that operate on a number of different kinds of cancellable things.

CountdownEvent

An extremely common pattern in parallel programming is **fork/join**, where a thread may spawn a certain number of activities and must later wait for them to complete. That's the purpose of `System.Threading.CountdownEvent` type. We saw this in Chapter 13, Data and Task Parallelism, and wrote a few code samples that relied on such a primitive (e.g., to implement parallel `for` loops and the like).

```
public class CountdownEvent : ISupportsCancelation, IDisposable
{
    // Constructor

    public CountdownEvent(int count);

    // Methods

    public void Cancel();

    public bool Decrement();
    public bool Decrement(int count);

    public void Dispose();
    protected virtual void Dispose(bool disposing);

    public void Increment();
    public void Increment(int count);
    public bool TryIncrement();
    public bool TryIncrement(bool count);

    public void Reset();
    public void Reset(int count);

    public void Wait();
    public bool Wait(int timeoutMilliseconds);
    public bool Wait(TimeSpan timeout);

    // Properties

    public int CurrentCount { get; }
    public int InitialCount { get; }
    public bool IsCanceled { get; }
    public bool IsSet { get; }
    public WaitHandle WaitHandle { get; }
}
```

The basic usage of `CountdownEvent` looks something like this:

```
using (CountdownEvent c = new CountdownEvent(N))
{
    for (int i = 0; i < N; i++)
        ThreadPool.QueueUserWorkItem(delegate
    {
        try
        {
            // something interesting ...
        }
    });
}
```

```
        finally
        {
            c.Decrement();
        }
    });
}

c.Wait();
}
```

A new event is constructed with an initial count (retrievable with the `InitialCount` property), and its current count is initialized to that (also retrievable afterward, with the `CurrentCount` property). Then threads call `Decrement` to subtract one from the current count. Any number of threads can wait, and they will be blocked until the event's count reaches 0. At that point, `IsSet` will report back true. You can `Reset` the event, which (by default) unsignals the event and changes its current count to the initial count (or the count specified as an argument to `Reset` if you so choose). The event is backed by a lazily allocated Windows kernel event, so it is a good idea to call `Dispose` on it when you're done.

LazyInit<T>

As we saw in Chapter 10, Memory Models and Lock Freedom, lazy initialization of program data is a common need that is often solved by the double-checked locking pattern. This pattern is not completely obvious and has been subject to a lot of misunderstanding in the past due to the weaker .NET ECMA memory model. And at the very least, it turns out to be complete boilerplate. The `System.Threading.LazyInit<T>` value type is a really simple, lightweight data structure that abstracts away all of these things.

```
public struct LazyInit<T> : IEquatable<LazyInit<T>>, ISerializable
    where T : class
{
    // Constructors

    public LazyInit();
    public LazyInit(Func<T> valueSelector);
    public LazyInit(LazyInitMode mode);
    public LazyInit(Func<T> valueSelector);
    public LazyInit(Func<T> valueSelector, LazyInitMode mode);
```

```

// Methods

public bool Equals(LazyInit<T> other);

// Properties

public LazyInitMode Mode { get; }
public bool IsInitialized { get; }
public T Value { get; }
}

public enum LazyInitMode
{
    AllowMultipleExecution,
    EnsureSingleExecution,
    ThreadLocal
}

```

The basic usage of `LazyInit<T>` is to use it as a field of an object. Then when the value is required, you will invoke the `Value` property; it internally handles lazily initializing upon the first access. If you don't wish to force initialization, you can first check `IsInitialized`. The common way to specify the initialization routine is to provide a `Func<T>` at construction time. If you opt not to do that, then `T` must define a no-arguments constructor and `Activator.CreateInstance` will be used to invoke it instead. Notice also that `T` is constrained to being a reference type.

For example, say we need a `ManualResetEvent` field on an object. Because this is a heavyweight kernel object, it'd be unfortunate to allocate and subsequently have to close it if it isn't even ever needed. We can use a `LazyInit<T>` for the field instead.

```

private LazyInit<ManualResetEvent> m_event =
    new LazyInit<ManualResetEvent>(() => new
ManualResetEvent(false));

```

`LazyInit<T>` is a value type to reduce its overhead: it truly is just a handful of bytes in size. But this means you'll need to be careful that you don't copy it. Doing so can lead to multiple initialization calls for the same original value.

As we saw back in Chapter 10, Memory Models and Lock Freedom, there are several variants of lazy initialization. The `LazyInit<T>` class offers a `LazyInitMode` enum that enables you to choose the appropriate

flavor for your scenario. The default is `AllowMultipleExecution`; this means that multiple objects could be created if threads are racing to access `Value`, but only one will be published. In the case that `T` implements `IDisposable`, any garbage objects will be automatically disposed. Alternatively, if the risk of creating multiple objects is too great—because it'd lead to correctness or performance problems—you can specify `EnsureSingleExecution` instead. This uses a lock internally to guarantee that only one object gets created.

Finally, the `ThreadLocal` mode is quite different from the rest. It ensures that each individual thread that accesses `Value` gets its own copy. The initialization routine will be run once per unique thread access. This can ease the common pattern of needing to check for `ThreadStatic` lazy initialization upon every access by eliminating a lot of boilerplate.

ManualResetEventSlim

The previous `LazyInit<T>` example for `ManualResetEvent` was timely. The need for a one way latch that can either be signaled or unsignaled is perhaps the most common synchronization primitive used in concurrent programs. Windows offers manual reset event kernel objects for this purpose, but they are heavyweight. The CLR offers condition variables, but they are not “sticky” and thus can’t be used in the same kinds of scenarios. This often leads developers to build custom ad hoc solutions that shadow the event’s state in user-mode, spin wait before blocking, and lazy initialize the event object only when waiting is truly needed.

This is precisely what `System.Threading.ManualResetEventSlim` does. It contains a single field that represents the state of the event. Only if the field indicates the event is not set, waiters will force allocation of a kernel object to wait on. But subsequent operations still check the field first before falling back to costly kernel-mode transitions.

```
public class ManualResetEventSlim : IDisposable
{
    // Constructors
    public ManualResetEventSlim();
    public ManualResetEventSlim(bool initialState);
    public ManualResetEventSlim(bool initialState, int spinCount);
```

```
// Methods
public void Dispose();
protected virtual void Dispose(bool disposing);
public void Reset();
public void Set();
public void Wait();
public bool Wait(int millisecondsTimeout);
public bool Wait(TimeSpan timeout);

// Properties
public bool IsSet { get; }
public int SpinCount { get; }
public WaitHandle WaitHandle { get; }
}
```

The usage of `ManualResetEventSlim` is nearly identical to `ManualResetEvent`. You initialize the event and optionally provide its `initialState` (true for signaled, `false` for unsignaled—the default). You then `Set`, `Reset`, and/or `Wait` on the event. You can check the user-mode state of the event by calling `IsSet`. For interoperability with things such as `WaitHandle.WaitAny` and `WaitAll`, you can grab the `WaitHandle` directly, which forces allocation. Finally, it's a good idea to call `Dispose` on the object when you're through with it, as this will dispose of the underlying event if it got lazy allocated.

SemaphoreSlim

`System.Threading.SemaphoreSlim` is to `Semaphore` as `ManualResetEventSlim` is to `ManualResetEvent`. It keeps state in user-mode and only allocates a kernel object when it needs to block. The internal algorithm performs spin waiting and is generally far more efficient than using the kernel semaphore directly.

```
public class SemaphoreSlim : IDisposable, ISupportsCancellation
{
    // Constructors
    public SemaphoreSlim(int initialCount);
    public SemaphoreSlim(int initialCount, int maxCount);

    // Methods
    public void Cancel();
    public void Dispose();
    protected virtual void Dispose(bool disposing);
```

```
public int Release();
public int Release(int releaseCount);
public void Wait();
public bool Wait(int millisecondsTimeout);
public bool Wait(TimeSpan timeout);

// Properties
public WaitHandle AvailableWaitHandle { get; }
public int CurrentCount { get; }
public bool IsCanceled { get; }
}
```

Everything here is straightforward. When you initialize the semaphore, you provide a current count and, optionally, the maximum count. (`Int32.MaxValue` is chosen as the maximum if you do not specify one.) You then call `Wait` to decrement the semaphore count, and `Release` to increment it. You can access the count via the `CurrentCount` property. There is also an `AvailableWaitHandle` property, which gives you an event that you can use for `WaitAny` and `WaitAll` style waits. Note that this event, when set, does not modify the semaphore's count; any thread using it for waiting must call `Wait` on the semaphore object after waking up to decrement the count. It is merely an indication that the semaphore is available.

A unique aspect to `SemaphoreSlim` is that it supports cancellation by implementing the `ISupportsCancellation` interface. By calling `Cancel` on it, any threads waiting will be immediately awoken with an `OperationCanceledException`.

SpinLock

Building a proper spin lock isn't as straightforward as you'd assume, as we saw in Chapter 14, Performance and Scalability. But for leaf-level locks that are meant to be held for very short periods of time, experience low degrees of contention, and where you'd like to minimize overhead and resource usage impact, they can be quite useful. Parallel Extensions includes a `System.Threading.SpinLock` type that can be used for such circumstances.

```
public struct SpinLock
{
    // Constructors
    public SpinLock();
    public SpinLock(bool enableThreadOwnerTracking);
```

```
// Methods
public void Enter(ref bool taken);
public bool TryEnter(ref bool taken);
public bool TryEnter(TimeSpan timeout, ref bool taken);
public bool TryEnter(int timeoutMilliseconds, ref bool taken);
public void Exit();
public void Exit(bool useMemoryBarrier);

// Properties
public bool IsHeld { get; }
public bool IsHeldByCurrentThread { get; }
public bool IsThreadOwnerTrackingEnabled { get; }
}
```

Notice that `SpinLock` is a value type. Its size is 4 bytes total, but you'll need to be very careful that you don't copy it around, since the copies won't enjoy mutual exclusion with respect to one another. Using it is probably relatively obvious: `Enter` is used to acquire the lock (or `TryEnter` if you'd like to use a timeout), which spins until available if it's taken, and `Exit` is used to release the lock. You might wonder why every overload accepts a `ref bool taken` argument. This is to enable their use in reliable situations, where asynchronous exceptions might otherwise lead to orphaned locks.

The regular pattern of usage is:

```
SpinLock slock = ...;
...
bool wasTaken = false;
try
{
    slock.Enter(ref wasTaken);
    // Critical region body
}
finally
{
    slock.Exit();
}
```

An overload of `Exit` allows you to control if a full memory fence is used to release the lock. This is true by default, but does mean the cost of acquiring and releasing is two interlocked operations instead of one. This is done to prevent subsequent code from moving inside the critical region. If you know this cannot happen, or it is safe, you can pass `false`.

When thread owner tracking is enabled, which it is by default and if you pass true for the enableThreadOwnerTracking constructor argument, the lock will use the calling thread's identity to mark lock ownership (when the lock is acquired). The IsThreadOwnerTrackingEnabled property indicates whether the lock was created in this way. This aids debuggability at the expense of some performance. When the lock is owned there is no way to find out what particular thread is holding it without this feature. By turning it on, Enter will throw exceptions instead of spin indefinitely when a thread tries to recursively acquire a lock, Exit will validate that the exiting thread is indeed the owning thread, and IsHeldByCurrentThread will accurately report back status based on the current thread.

It's common to turn this on debug builds, but to turn it off in release builds.

```
SpinLock slock = new SpinLock(  
    #if DEBUG  
        true  
    #else  
        false  
    #endif  
);
```

SpinWait

As we also saw in Chapter 14, Performance and Scalability, coming up with a good general purpose spin waiting algorithm is tricky. Parallel Extensions comes with a super simple `SpinWait` value type that is just four bytes in size. This logic is used by the entire library whenever it needs to spin, including the waiting performed by `SpinLock`. Anytime you need to spin wait for a brief period of time, you can use this type.

```
public struct SpinWait  
{  
    // Constructors  
    public SpinWait();  
  
    // Methods  
    public void SpinOnce();  
    public void Reset();
```

```
// Properties  
public int Count { get; }  
public bool NextSpinWillYield { get; }  
}
```

The `SpinOnce` method performs the spin and alters its logic based on how many times it has been called. It does this by keeping a count internally, which is also exposed via the `Count` property. You can call `Reset` if you want to reset this count back to 0. Internally, this type performs some ratio of busy spins to yields with different Win32 APIs (i.e., `SwitchToThread`, `Sleep(0)`, and `Sleep(1)`). You can use the `NextSpinWillYield` property to tell you if the next call to `SpinOnce` will forfeit the current timeslice. For uses that eventually fall back to true waiting, this can be a cue that it's time to stop spinning, as the following code snippet illustrates.

```
SpinWait sw = ...;  
while (!P)  
{  
    if (sw.NextSpinWillYield)  
        // Do true wait  
    else  
        sw.SpinOnce();  
}
```

This is what `ManualResetEventSlim` does internally inside its `Wait` method. If the user-mode state indicates the event is unsignaled, a loop very much like the one above is used; if `NextSpinWillYield` reports back `true`, the kernel object is lazily allocated and waited on.

Concurrent Collections

The last major pillar of functionality provided by Parallel Extensions is concurrent containers. These are some commonly used collections types that are useful for concurrent programs, including a producer/consumer blocking and bounded collection, and a lock free queue and stack. All of these collections classes can be found in the `System.Collections.Concurrent` namespace.

BlockingCollection<T>

We saw in Chapter 12, Parallel Containers, that producer/consumer situations often call for **blocking** and **bounded queues**. These are queues that block consumers on dequeue when the queue is empty and that block producers on enqueue when the queue is full. Parallel Extensions comes with such a collection out of the box, called `BlockingCollection<T>`, which supports both. Additionally, it abstracts away the underlying storage mechanism, so that any of the other kinds of concurrent collections offered (or more specifically any implementation of the `IProducerConsumerCollection<T>` interface) can be plugged in for the underlying storage. It, by default, uses a concurrent queue if one is not specified.

```
public class BlockingCollection<T> :  
    IEnumerable<T>, ICollection, IEnumerable, IDisposable  
{  
    // Constructors  
    public BlockingCollection();  
    public BlockingCollection(int boundedCapacity);  
    public BlockingCollection(  
        IProducerConsumerCollection<T> collection  
    );  
    public BlockingCollection(  
        IProducerConsumerCollection<T> collection,  
        int boundedCapacity  
    );  
  
    // Methods  
    public void Add(T item);  
    public bool TryAdd(T item);  
    public bool TryAdd(T item, int millisecondsTimeout);  
    public bool TryAdd(T item, TimeSpan timeout);  
  
    public T Take();  
    public bool TryTake(out T item);  
    public bool TryTake(out T item, int millisecondsTimeout);  
    public bool TryTake(out T item, TimeSpan timeout);  
  
    public void CompleteAdding();  
    public void CopyTo(T[] array, int index);  
    public void Dispose();  
    public IEnumerable<T> GetConsumingEnumerable();  
    public T[] ToArray();
```

```
// Static methods

public static int AddAny(
    BlockingCollection<T>[] collections,
    T item
);
public static int TryAddAny(
    BlockingCollection<T>[] collections,
    T item
);
public static int TryAddAny(
    BlockingCollection<T>[] collections,
    T item,
    int millisecondsTimeout
);
public static int TryAddAny(
    BlockingCollection<T>[] collections,
    T item,
    TimeSpan timeout
);

public static int TakeAny(
    BlockingCollection<T>[] collections,
    out T item
);
public static int TryTakeAny(
    BlockingCollection<T>[] collections,
    out T item
);
public static int TryTakeAny(
    BlockingCollection<T>[] collections,
    int millisecondsTimeout,
    out T item
);
public static int TryTakeAny(
    BlockingCollection<T>[] collections,
    TimeSpan timeout,
    out T item
);

// Properties
public int BoundedCapacity { get; }
public int Count { get; }
public bool IsAddingCompleted { get; }
public bool IsCompleted { get; }
}

public interface IProducerConsumerCollection<T> :
    IEnumerable<T>, ICollection, IEnumerable
```

```
{  
    bool Add(T item);  
    bool Take(out T item);  
    T[] ToArray();  
}
```

When you construct a new `BlockingCollection<T>`, you may optionally specify the underlying collection and the bounding size. Aside from that, the class's surface area is quite large, but basically boils down to the `Add` and `Take` methods used to add and remove elements, respectively, with the bounding and blocking behavior. There are also `TryAdd` and `TryTake` overloads that can be used if you wish to avoid blocking, or wish to bound the amount of maximum time spent blocking based on a timeout value.

Similarly, there are a set of static methods: `AddAny`, `TryAddAny`, `TakeAny`, and `TryTakeAny`, each of which accepts an array of `BlockingCollection<T>` objects and will add or remove from the first collection in the list which is unblocked. The index in the supplied array is returned so that you know which collection was affected. The timeout variants return `-1` as a value when timeout occurs.

In typical producer/consumer situations, the consumers will continue taking elements until the producers are done. This is what the `CompleteAdding` method is for; it signals to consumers that, once the collection becomes empty, no additional elements are to be expected. After this has been called, `IsAddingCompleted` returns `true`. The `IsCompleted` property returns `true` so long as this property returns `true` and the underlying collection has been emptied. A typical usage will look something like this:

```
BlockingCollection<T> c = ...;  
  
// Producer:  
while (...)  
{  
    c.Add(...);  
}  
c.CompleteAdding();  
  
// Consumer:  
T elem;
```

```

while (c.TryTake(Timeout.Infinite, out elem))
{
    ...
}

```

To make this common pattern of consumption simpler, you can use the `GetConsumingEnumerable` method. It returns an `IEnumerable<T>` that removes elements from the collection as it enumerates, and will only quite once `CompleteAdding` has been called by a producer.

```

// Consumer:
foreach (T elem in c.GetConsumingEnumerable())
{
    ...
}

```

ConcurrentQueue<T>

The `ConcurrentQueue<T>` class is an implementation of the lock free FIFO queue algorithm explained back in Chapter 12, Parallel Containers. There is no guarantee that it will be lock free, but it just so happens to be today. The implementation uses a linked list internally. It has a very basic public surface area, and is the default collection used by `BlockingCollection<T>` if an alternative is not provided.

```

public class ConcurrentQueue<T> :
    IProducerConsumerCollection<T>, IEnumerable<T>, ICollection,
    IEnumerable, ISerializable, IDeserializationCallback
{
    // Constructors
    public ConcurrentQueue();
    public ConcurrentQueue(IEnumerable<T> collection);

    // Methods
    public void CopyTo(T[] array, int index);
    public void Enqueue(T element);
    public T[] ToArray();
    public bool TryDequeue(out T result);
    public bool TryPeek(out T result);

    // Properties
    public int Count { get; }
    public int IsEmpty { get; }
}

```

As you might imagine, `Enqueue` places an element at the head of the queue, and `TryDequeue` takes an element off the tail of the queue. There is no `Dequeue` method provided because in concurrent situations you must always deal with the fact that the queue's contents are constantly changing. Similarly, there is a `TryPeek` method that examines the tail of the queue but does not actually dequeue it. The `Count` property computes the count (at some expense—it is an $O(N)$ operation) and `IsEmpty` quickly tells you whether it is empty.

ConcurrentStack<T>

Much like `ConcurrentQueue<T>`, the `ConcurrentStack<T>` type is an implementation of the lock free FIFO stack algorithm examined back in Chapter 10, Memory Models and Lock Freedom. The implementation is also a linked list.

```
public class ConcurrentStack<T> :  
    IProducerConsumerCollection<T>, IEnumerable<T>, ICollection,  
    IEnumerable, ISerializable, IDeserializationCallback  
{  
    // Constructors  
    public ConcurrentStack();  
    public ConcurrentStack(IEnumerable<T> collection);  
  
    // Methods  
    public void Clear();  
    public void CopyTo(T[] array, int index);  
    public void Push(T item);  
    public T[] ToArray();  
    public bool TryPeek(out T result);  
    public bool TryPop(out T result);  
  
    // Properties  
    public int Count { get; }  
    public bool IsEmpty { get; }  
}
```

The design philosophy behind this type is nearly equivalent to the queue data type. You use `Push` to add elements to head of the stack and `TryPop` to take elements off the head off the stack. There is also a `TryPeek` that returns the current head element without actually modifying it. The stack also supports an efficient $O(1)$ `Clear` method that clears its contents.

FURTHER READING

- J. Duffy, E. Essey. Parallel LINQ: Running Queries on Multi-Core Processors. *MSDN Magazine* (2007).
- D. Leijen, J. Hall. Parallel Performance: Optimize Managed Code for Multi-Core Machines. *MSDN Magazine* (2007).
- Microsoft Parallel Extensions Team. What's New in the June 2008 CTP of Parallel Extensions. Weblog article, <http://blogs.msdn.com/pfxteam/archive/2008/06/02/8567093.aspx> (2008).

Index

A

- ABA problem, 536–537
- Abandoned mutexes, 217–219
- `AbandonedMutexException`, 205
- Abort API, 109–110
- Aborts, thread, 109–113
- Account identifiers, lock levels, 583–584
- Acquire fence, 512
- `AcquireReaderLock`, 300
- `AcquireSRWLockExclusive`, 290
- `AcquireSRWLockShared`, 290
- `AcquireWriterLock`, 300
- Actions, TPL, 890
- Actual concurrency, 5
- Add method, dictionary, 631
- `AddOnPrerenderCompleteAsync`, 420–421
- Affinity. *See* CPU affinity
- Affinity masks, 172–173, 176–178
- Agents
 - concurrent program structure, 6
 - data ownership and, 33–34
 - style concurrency, 79–80
- `AggregateException` class, TPL, 893–895
- Aggregating multiple exceptions, 724–729
- Alertable waits
 - asynchronous procedure calls and, 209
 - defined, 85
 - kernel objects and, 188
 - overview of, 193–195
- Algorithms
 - cooperative and speculative, 719
 - dataflow, 689
- natural scalability of, 760–761
- recursive, 702–703
- scalability of parallel, 666
- search, 718–719
- sorting, 681
- Alignment
 - load/store atomicity and, 487–492
 - reading from or writing to unaligned addresses, 23
- `_alloc` function, 141
- `AllocateDataSlot`, 123
- `AllocateNamedDataSlot`, 123
- AMD64 architecture, 509–511
- Amdahl's Law, 762–764
- Antidependence, 486
- Apartment threading model, COM, 197
- APC callback, 806–808
- APCs (asynchronous procedure calls)
 - kernel synchronization and, 208–210
 - lock reliability in managed code and, 878
 - overview of, 84–85
- APM (asynchronous programming model), 400–419
- ASP.NET asynchronous pages and,
 - 420–421
- callbacks, 412–413
- calling `AsyncWaitHandle` `WaitOne`, 407–410
- calling `EndFoo` directly, 405–407
- defined, 399
- designing reusable libraries with, 884–885
- implementing `IAsyncResult`, 413–418
- overview of, 400–403
- polling `IsCompleted` flag, 411

APM (asynchronous programming model), *continued*
 rendezvousing 4 ways, 403–405
 using in .NET Framework, 418–419

`AppDomain.ProcessExit` event, 116

AppDomains
 designing library locks, 870, 873–874
 fine-grained message passing support, 72
 intraprocess isolation, 32
 locking on agile objects, 278–281
 safety of thread aborts, 111
 using kernel objects for synchronization, 188

`AppDomainUnloadedException`, 104, 111

Application bugs, 140–141

`ApplicationException`, 301–302

Architecture, concurrent program, 6–8

Arrays, fine-grained locking, 616
`_asm` keyword, 148

`AsOrdered`, PLINQ, 914

ASP.NET asynchronous pages, 420–421

Assemblies, and lock orderings, 584

`AsUnordered`, PLINQ, 914–915

Async prefix, 400, 421–422

`AsyncCompletedEventArgs` class, 423

`AsyncCompletedEventHandler` event, 423

Asynchronous aborts, 109, 112–113

Asynchronous exceptions, 281–282, 298–299

Asynchronous I/O. *See also* Overlapped I/O
 .NET Framework. *See* .NET Framework
 asynchronous I/O
 benefits of, 787
 cancellation, 822–826
 Win32. *See* Win32 asynchronous I/O

Asynchronous operations
 .NET Framework, 855–856
 concurrent programs, 6

Asynchronous pages, ASP.NET, 420–421

Asynchronous procedure calls. *See* APCs
 (asynchronous procedure calls)

Asynchronous programming models
 APM. *See* APM (asynchronous programming model)
 ASP.NET asynchronous pages, 420–421
 event-based asynchronous pattern, 421–427
 overview of, 399–400

`AsynchronousOperationManager`, 830, 837

`AsyncOperationManager`, 855–856

`AsyncWaitHandle`, APM, 404, 407–410, 416

`atexit/_oneexit` function, 113

Atomic loads, 487–492, 499–500

Atomic stores, 487–492, 499–500

Atomicity, managing state with, 29–30

Auto-reset events, 226–234
 creating and opening, 228–230
 implementing queue with, 244–245
 overview of, 226–227
 priority boosts and, 232–234
 setting and resetting, 230–231
 signaled/nonsignaled state transition, 186
`WAIT_ALL` and, 231–232

`AutoBuffered` merge, PLINQ, 913–914

`AutoResetEvent`, 228–229

B

Background threads, 103

`BackGroundWorker`, 400, 426, 856–860

Bakery algorithm, 54–55

Balance set manager, 165, 609

`bAlertable` argument, 209

Barriers, phased computations, 650–654

Batcher’s bitonic sort, 681

Begin prefix, APM, 399

`BeginFoo` method, APM, 401–402, 405–407

`BeginInvoke`, 838–839

`_beginthread`, 96–98, 107, 132

`BeginThreadAffinity`, 880

`_beginthreadex`, 96–98, 103, 132

Benign race conditions, 549, 553–555, 621

Binary semaphores, 42

`BindHandle` method, I/O completion ports, 369–370

`BindIoCompletionCallback` routine, I/O completion ports, 359–360

`bInheritHandle` parameter, `CreateThread`, 95

Bit-masks, 172

Bit-test-and-reset (BTR), 502–503

Bit-test-and-set (BTS), 502–503

Bitness, load/store atomicity, 487

`Block` routine, UMS, 461–463

Blocking queues
 with condition variables, 307–309, 644–646

with events, 243–244

with monitors, 310–311, 642–644, 646–650

mutex/semaphore example, 224–226

producer/consumer data structures, 641

using `BlockingCollection<T>`, 925–928

`BlockingCollection<T>`, 925–928

Blocks, thread
 building UMS and, 461–463
 canceling calls, 730

- CLR locks avoiding, 275–277
critical sections avoiding, 263–266
data parallelism and, 665
dataflow parallelism avoiding, 695–698
designing reusable libraries, 884–885
disadvantages of fibers, 433–434
existing APIs for, 885
lock free algorithms, 519
producer/consumer data structures, 642
reasons for, 83
spin waiting and, 767
stack vs. stackless, 472–473
body delegate, 662, 757
BOOL bAlertable parameter, alertable waits, 193–195
Bounded buffer, 641
Bounded queues
 overview of, 646–650
 using `BlockingCollection<T>`, 925–928
Bounding, 642
BTR (bit-test-and-reset), 502–503
BTS (bit-test-and-set), 502–503
Buffering, in PLINQ, 912–914
Busy spin waiting, 63–64, 65
BWaitAll argument, 191
bWaitAll argument, 202
- C**
C++ Programming Language, Third Edition, 36
C programs
 coordination containers, 646–650
 creating threads in, 90, 96–98
 creating threads in .NET Framework, 100–101
 D11Main function in, 116–117
 terminating threads. *See* `Threads`, termination methods
C Runtime Library (CRT), 90, 96–98
Cache coherency, 479
Cache, using isolated state as, 32
CallbackMayRunLong, 349
Callbacks
 fiber local storage and, 446
 implementing `IAsyncResult`, 416
 rendezvous technique, 412–413
 Vista thread pool, 334–336, 347–351
Cancel function, TPL, 897
CancelAndWait function, TPL, 897
CancelAsync method, 425
Cancellation
 asynchronous I/O, 822–826
 asynchronous operations, 729–731
 event-based asynchronous pattern, 425
 task parallel library, 897
CancelWaitableTimer, 236
CAS (compare and swap) hardware
 ABA problem and, 536–537
 implementing, 496–499
 lock free FIFO queue and, 635–636
 modifying memory location atomically in, 492
Casual dependence, among threads, 62
CCR (Coordination and Concurrency Runtime)
 fine-grained message passing, 73
 message-based parallelism, 719
 stackless and nonblocking programs, 473
Change methods, CLR, 373–374
CheckForSufficientStack, 149–151
Children, task parallel library, 895–897
Circular waits, 575, 577
Cleanup groups, Vista, 343, 345–347
CloseHandle API, `CreateMutex`, 214–215
CloseThreadPool, Vista, 344
CloseThreadPoolCleanupGroup, Vista, 347
CloseThreadPoolCleanupGroupMembers, Vista, 346–347
CloseThreadpoolIo, Vista, 336
CloseThreadpoolTimer, Vista, 333
CloseThreadpoolWait, Vista, 339–340, 341–342
CloseThreadpoolWork, Vista, 327, 329
CLR. *See also* managed code
 .NET memory models, 517
 avoiding locks, 873
 fibers and, 449–453
 garbage collection, 766
 lazy initialization in .NET and, 521–526
 locks. *See* `Locks`, CLR
 process shutdown, 116, 569–571
 reaction to stack overflow, 141–142
 reader/writer locks, 254–255, 300–304
 single assignment variables, 35–36
 threads, 85–87
 unhandled exceptions in, 104
 waiting for managed code, 206–208
CLR thread pool, 364–391
 case study, 387–391
 debugging, 386–387

- CLR thread pool, *continued*
 fine-grained concurrency with, 884
 I/O completion ports, 368–371
 no ownership of threads in, 377
 overview of, 317–319, 364
 performance of, 391–397
 registered waits, 374–377
 thread management in, 377–386
 timers, 371–374
 work items, 364–368
- CMPXCHG variant, 496–499, 500–502
- Coarse-grained critical regions, 45–47, 550–553
- Coarse-grained locks, 256–257, 583, 614
- cobegin statement, structured parallelism, 70
- Code motion, 478–479
- Coffman conditions, 576–577
- COM
 APIs for waiting, 186
 how CLR waits for managed code, 207
 message pumping, 195–201, 202–204
 Single Threaded Apartments, 833–834
 using kernel objects, 188
- CoMarshalInterface API, 197
- Commit size, thread stacks
 memory layout, 138
 overflow, 140–145
 overview of, 130–133
- Communicating Sequential Processes (CSP), 71–72
- Compare and exchange, 496–499
- Compare and swap. *See* CAS (compare and swap) hardware
- Compiler
 .NET Framework memory models, 517
 creating fences in VC++ at level of, 514–515
 load/store atomicity and, 490–492
- CompilerServices, 274–275
- Completed suffix, 422
- CompletedSynchronously, APM, 417
- Composite actions, 550–553
- Concurrency, 3–12
 agents-style, 80
 hazards. *See* Correctness hazards; Liveness hazards
 importance of, 3–5
 layers of parallelism, 8–10
 limitations of, 10–11
 of parallel containers, 614
 program architecture and, 6–8
- unstructured, 896–897
 of Vista thread pool, 348
- Concurrent collections
BlockingCollection<T>, 925–928
ConcurrentQueue<T>, 928–929
ConcurrentStack<T>, 929
 defined, 924
- Concurrent exceptions, 721–729
 aggregating multiple exceptions, 724–729
 marshaling exceptions across threads, 721–724
 overview of, 721
- ConcurrentQueue<T>, 928–929
- ConcurrentStack<T>, 929
- Condition variables
 .NET Framework monitors, 68–70, 309–312
 C++ blocking queue with, 644–646
 CLR monitors, 272
 defined, 255
 overview of, 304
 Windows Vista, 304–309
- const modifier, single assignment, 35–38
- CONTEXT data structure, 151–152, 437, 440–441
- Context, defined, 82
- Context switches
 defined, 82
 expense of, 768, 884
 fibers reducing cost of, 431
 I/O operations and, 785, 787, 810, 824
 spin locks and, 769–770
- ContextSwitch, building UMS
 dispatching work, 461–463
 overview of, 464–470
 queueing work, 464–470
- ContextSwitchDeadlock, 575
- Continuation passing style (CPS), 65–66, 412–413
- Continuations, task parallel library, 900–902
- ContinueWith methods, TPL, 900–902
- Continuous iterations, 663–667
- Control flow invariants, 548
- Control synchronization, 60–73
 condition variables and. *See* Condition variables
 coordination and, 60–61
 defined, 14
 events and, 66–68
 message passing, 71–73
 monitors and, 68–70

primitives and, 255
state dependence among threads, 61–62
structured parallelism and, 70–71
waiting for something to happen, 63–66
Convention, enforcing isolation, 32
`ConvertFiberToThread`, 442
`ConvertThreadToFiber(Ex)`, 438–439,
 442–444
Convoys, lock, 603–605
Cooperative search algorithms, 719
Coordination. *See* Control synchronization
Coordination and Concurrency Runtime. *See*
 CCR (Coordination and Concurrency
 Runtime)
Coordination containers, 640–650
C# blocking/bounded queue with multiple
 monitors, 646–650
producer/consumer data structures,
 641–642
simple C# blocking queue with critical
 sections and condition variables,
 644–646
simple C# blocking queue with monitors,
 642–644
Correctness hazards
 overview of, 546
 recursion and reentrancy, 555–561
Correctness hazards, data races, 546–555
 benign, 553–555
 composite actions, 550–553
 inconsistent synchronization, 549–550
 overview of, 546–549
Correctness hazards, locks and process
 shutdown, 561–571
 managed code and shutdown, 569–571
 overview of, 561–563
 Win32: weakening and termination,
 563–568
`CountdownEvent`, 915–917
Counting semaphores, 42
`CoWaitForMultipleHandles` API, 186,
 202–204, 207
CPS (continuation passing style), 65–66,
 412–413
CPU affinity
 assigning affinity, 173–176
 microprocessor architecture and, 178–179
 overview of, 171–173
 round robin affinitization, 176–178
`CreateEvent(Ex)`, 228–230
`CreateFiber(Ex)`, 435–436
`CreateMutex(Ex)`, 212–216
`CreateRemoteThread`, 95–96
`CreateSemaphore(ex)` APIs, 220–222
`CREATE_SUSPENDED` flag, 153, 169
`CreateThread`
 C programs, 96–98
 creating threads in .NET, 99
 creating threads in Win32, 90
 example of, 92–94
 failure of, 92
 parameters, 90–92
 specifying stack changes, 132
 thread suspension, 169
 triggering thread exit, 103
`CreateThreadPool`, Vista, 344
`CreateThreadPoolCleanupGroup`, Vista, 345–347
`CreateThreadpoolIo`, Vista, 334–335
`CreateThreadpoolTimer`, Vista, 330–331, 333
`CreateThreadpoolWait`, Vista, 336–337
`CreateThreadpoolWork`, Vista, 326–327,
 329–330
`CreateTimerQueueTimer`, legacy thread pool,
 356–358
`CreateWaitableTimer(Ex)`, 235–236
`CreateWindow(Ex)`, 195
Critical finalizers, 300
Critical paths, speedup and, 764–765
Critical regions
 avoiding deadlocks with, 576
 as binary semaphores, 42
 coarse vs. fine-grained, 45–47
 correctly built, 478
 correctness hazards, 551
 defined, 21, 40
 eliminating data races with, 40–42
 failure of in modern processors, 59
 as fences, 484–485
 implementing, 47–48
 implementing with critical sections. *See*
 Critical sections, Win32
 patterns of usage, 43–45
Critical sections, C++ blocking queue
 with, 644–646
Critical sections, CLR monitors, 272
Critical sections, Win32, 256–271
 allocating, 256–257
 debugging ownership information, 270–271
 defining, 254
 entering and leaving, 260–266

- Critical sections, Win32, *continued*
 fibers and, 448–449
 implementing critical regions, 256
 initialization and deletion, 257–259
 integration with Windows Vista condition variables, 304–309
 low resource conditions, 266–270
 overview of, 256
 process shutdown and, 563–568
 Vista thread pool completion tasks, 350
- CRITICAL_SECTION.** *See* Critical sections, Win32
- CRT (C Runtime Library),** 90, 96–98
- CSP (Communicating Sequential Processes) systems,** 71–72
- Current.ManagedThreadId,** 879
- CurrentThread,** 101
- D**
- Data access patterns, 677–678
- Data dependencies, 485–486
- Data ownership, 33–34
- Data parallelism, 659–684
 concurrent program structure, 6–7
 continuous iterations, 663–667
 defined, 657–658
 dynamic decomposition, 669–675
 loops and iteration, 660–661
 mapping over input data as parallel loops, 675–676
 nesting loops and data access patterns, 677–678
 overview of, 659–660
 prerequisites for loops, 662
 reductions and scans, 678–681
 sorting, 681–684
 static decomposition, 662–663
 striped iterations, 667–669
- Data publication, 15–16
- Data races. *See* Race conditions (data races)
- Data synchronization, 40–60
 coarse vs. fine-grained regions, 45–47
 defined, 14, 38–40
- Dekker’s and Dijkstra’s algorithm, 50–53
 general approaches to, 14
 hardware compare and swap instructions, 55–58
 implementing critical regions, 47–48
 Lamport’s bakery algorithm, 54–55
- mutual exclusion. *See* Critical sections, Win32; Locks, CLR
- overview of, 40–42
- patterns of critical region usage, 43–45
- Peterson’s algorithm, 53–54
- primitives, 254–255
- reader writer locks. *See* RWLs (reader/writer locks)
- reordering, memory models and, 58–60
- semaphores, 42
- strict alternation, 49–50
- Dataflow parallelism
 futures, 689–692
 overview of, 689
 promises of, 693–695
 resolving events to avoid blocking, 695–698
- Deadlock
 concurrency causing, 10–11
 examples of, 572–575
 fine-grained locking for FIFO queues and, 617–621
 implementing critical regions without, 47 in library code, 874–875
 livelock vs., 601–603
 from low maximum threads, 382–385
 onAppDomain agile objects, 279–281
 overview of, 572
 ReaderWriterLockSlim and, 298
- Deadlock, avoiding, 575–589
 apartment threading model, 197–198
 The Banker’s Algorithm, 577–582
 with DllMain routine, 116–117
 with lock leveling, 581–589, 875–876
 overview of, 575–577
- Deadlock, detecting, 589–597
 overview of, 589–590
 with timeouts, 594
- with Vista WCT, 594–597
 with Wait Graph Algorithm, 590–594
- Deadly embrace. *See* Deadlock
- DeallocationStack field, TEB, 149
- Debugging
 CLR monitor ownership, 285–287
 CLR thread pool, 386–387
 as concurrency problem, 11
 critical sections, 270–271
 fibers, 433–434
 kernel objects, 250–251
 legacy RWL ownership, 303–304
 SRWLS, 293

- symbols, 139
thread suspension in, 170
user-mode thread stacks, 127–130
using CLR managed assistant for, 575
Vista thread pool, 353
Declarative, LINQ as, 910
Deeply immutable objects, 34
Dekker’s algorithm
 antipattern in, 540–541
 Dijkstra’s algorithm vs., 51–53
 failure of in modern processors, 59
 overview of, 50–51
 Peterson’s algorithm vs., 53–54
Delay-abort regions, 110–111
Delays, from low maximum threads, 385–386
Delegate types, 418
Deletion
 of critical sections, 257–259
 of fibers, 441–442
 of legacy thread pool timer threads, 358–359
Dependency, among threads, 61–62
`DestroyThreadpoolEnvironment`, Vista, 343
Dictionary (hashtable), building, 626–631
Dijkstra, Edsger
 algorithm of, 51–53
 The Banker’s Algorithm, 577–581
dining philosophers problem, 573–574
Dijkstra’s algorithm, 51–53
Dining philosophers problem, 573–574
`DisassociateCurrentThreadFromCallback`, Vista, 347
`DispatcherObject`, 840–846
`Dispose` overload, CLR, 374
`DllMain` function
 creating threads, 153
 initialization/deletion of critical regions, 259
 overview of, 115–117
 performing TLS functions, 119
`DLL_PROCESS_ATTACH`, 115, 119–120, 153
`DLL_PROCESS_DETACH`, 115, 119–120
`DLL_THREAD_ATTACH`, 115–116, 119–120, 153
`DLL_THREAD_DETACH`, 116, 119–120, 154
DNS resolution, 419
Document matching, 718
Documentation
 on blocking, 884
 on library locking model, 870
`DocumentPaginator`, 427
Domain parallelism, 8–9
`DoNotLockOnObjectsWithWeakIdentity`, 281
`DoSingleWait` function, 194–195
Double-checked locking
 lazy initialization in .NET, 521–527
 lazy initialization in VC++, 528–536
 overview of, 520
DPCs (deferred procedure calls), 84–85
`DuplicateHandle`, 94
`dwDesiredAccess`, 213
`dwFlags` argument, 199–201, 437, 439
`dwStackSize` parameter, `CreateThread`
 API, 91, 132
`dwTimeout`, 190
`dwWakeMask` argument, 199
Dynamic composition, recursive locks, 559
Dynamic (on demand) decomposition,
 669–675
 defined, 663
 for known size iteration spaces, 669–672
 overview of, 669
 for unknown size iteration spaces, 669–672
Dynamic TLS, 118–120, 122–123

E

- ECMA Common Language Infrastructure, 516–518
`EDITBIN.EXE` command, 132
Efficiency
 measuring, 761–762
 natural scalability vs. speedups, 760–761
 performance improvements due to, 756
`End` method, APM, 416
`End` prefix, APM, 399
`EndFoo`, APM, 401–407
`EndInvoke`, 838–839
`_endthread`, 107
`EndThreadAffinity`, 880
`_endthreadex`, 107
`EnterCriticalSection`
 ensuring thread always leaves critical section, 262
 entering critical section, 260–261
 fibers and critical sections, 448–449
 leaving unowned critical section, 261
 low resource conditions and, 267–268
 process shutdown, 563–564
 setting spin count, 264
`Entry`, APC, 208

- `Environment.Exit`, CLR, 113–114, 569–571
`Environment.FailFast`, CLR, 114, 141–142
Environments, Vista thread pool, 342–347
Erlang language, 720
ERROR_ALREADY_EXISTS, 213, 222
ERROR_ALREADY_FIBER, 439
ERROR_FILE_NOT_FOUND, 215
ERROR_OUT_OF_MEMORY, 258, 260, 266
ERROR_STACK_OVERFLOW, 134
Escape analysis, 19
Essential COM (Box), 198
ETHREAD, 145–146, 152
Event-based asynchronous pattern, 421–427
 in .NET Framework, 426–427
 basics, 421–424
 cancellation, 425
 defined, 400
 progress reporting/incremental results, 425–426
Event handlers, asynchronous I/O
 completion, 802–805
Event signals, missed wake-ups and, 600–601
Events
 blocking queue data structure with, 243–244
 completing asynchronous operations with, 422
 control synchronization and, 66–68
EventWaitHandle, 231
Exception handling
 with contexts, 152
 parallelism and, 721–729
EXCEPTION_CONTINUE_SEARCH, 106
EXCEPTION_EXECUTE_HANDLER, 106
Exceptions, 721–729
 aggregating multiple, 724–729
 lock reliability and, 875
 marshaling across threads, 721–724
 overview of, 721
Exchange
 128-bit compare, 500–502
 compare and, 496–499
 interlocked operations, 493–496
executeOnlyOnce, CLR thread pool, 375–376
Execution order, 480–484
Execution, Windows threads, 81–82
ExecutionContext, 839
exit/_exit function, 113
ExitProcess
 hazards of using, 563
terminating threads in Win32, 113
unhandled exceptions and, 104
ExitThread
 defined, 103
 overview of, 107–109
 specifying return code at termination, 94
Explicit threading, 87–88
Exponential backoff, in spin waiting, 770
- F**
- /F switch, PE stack sizes, 132
Facial recognition, 718
FailFast, 114
Fair locks
 exacerbating convoys, 604
 FIFO data structure, 185, 605
 in newer OSs, 217, 605
Fairness, in critical regions, 47
False contention, 615
Fences. *See* Memory fences
Fiber local storage (FLS), 430, 445–447
Fiber-mode, CLR and SQL Server, 86–87
Fiber user-mode stacks, 130
FiberBlockingInfo, UMS, 455–459
FiberPool
 building UMS. *See* UMS (user-mode scheduler)
 data structures, 455–459
 dispatching work, 461
 thread and fiber routines, 459–460
~**FiberPool** destructor, 470–472
Fibers, 429–474. *See also* UMS (user-mode scheduler), building
 advantages of, 431–433
 CLR and, 449–453
 converting threads into, 438–439
 creating new, 435–438
 deleting, 441–442
 determining whether threads are, 439–440
 disadvantages of, 433–435
 fiber local storage (FLS), 445–447
 overview of, 429–431
 routines, user-mode scheduler, 460
 switching between, 440–445
 thread affinity and, 447–449
FiberState
 building user-mode scheduler, 455–459
ContextSwitch and, 464–465
 dispatching work, 461

FiberWorkRoutine method, 460, 461
FIFO queues
 alertable waits, 195
 fine-grained locking for, 617–621
 general-purpose lock free, 632–636
 managing wait lists, 185
 waiting in Win32, 192
FILETIMES, 237–241
Finalizer thread, 79
Fine-grained critical regions, 45–47,
 550–553
Fine-grained locking, 616–632
 arrays, 616
 dictionary (hashtable), 626–632
 FIFO queue, 617–621
 introducing with CLR thread pool, 884
 linked lists, 621–626
 lock leveling and, 583
 overview of, 614
FineGrainedHashTable, 628–630
Fire and forget, 893
Flags
 legacy thread pool thread management, 363
 legacy thread pool work items, 354–355
 wait registrations, legacy thread pool,
 361–362
FLS (fiber local storage), 430, 445–447
FlsAlloc function, 445
for loops, 658–660, 757
For method, Parallel class, 904–908
forall statement, 70
foreach loops, 658–660
ForEach method, Parallel class, 904–908
Fork/join parallelism, 685–688, 915–917
FreeLibraryWhenCallbackReturns, Vista
 thread pool, 350
Freeze threads, 170
FS Register, accessing TEB via, 147–148
Full fence (MFENCE), 512–515
FullyBuffered merge, PLINQ, 913–914
Functional systems, 61
Futures
 building dataflow systems, 689–692
 pipelining output of, 698–702
 promises compared with, 693
 structured parallel construct, 70
 task parallel library, 898–900
Future<T> class
 ContinueWith methods, 900–902
 overview of, 898–900

G

Game simulation, and parallelism, 718
Garbage collection (GC), 79, 766–767
General-purpose lock free FIFO queue,
 632–636
GetAvailableThreads, Vista thread
 pool, 381
GetBucketAndLockNo, dictionary, 630–631
GetCurrentFiber macro, 439–440
GetCurrentThread, 94
GetCurrentThreadId, 93, 444
GetData, TLS, 123
GetExitCodeThread, 94
GetFiberData macro, 437, 440
GetLastError
 CreateThread, 92
 mutexes, 213, 215
 semaphores, 222
GetMaxThreads, Vista thread pool, 380–381
GetMessage, 198
GetMinThreads, Vista thread pool, 380–381
GetOverlappedResult, asynchronous
 I/O, 798–800
GetProcessAffinityMask, CPU, 173–174
GetThreadContext, 151
GetThreadPriority, 160, 162
GetThreadWaitChain, WCT, 595–596
GetUserContext, threads, 153
GetWindowThreadProcessId method, 839
Global store ordering, 511
Graphical user interfaces. *See GUI (graphical
 user interfaces)*
Guard page
 creating stack overflow, 140–145
 example of, 137
 guaranteeing committed guard space,
 134–135
 overview of, 133–134
 resetting after stack overflow, 143
Guarded regions, 311–312
GUI (graphical user interfaces), 829–861
.NET Framework. *See .NET Framework*
 Asynchronous GUI
 cancellation from, 730
 message-based parallelism and, 720
 overview of, 829–830
 responsiveness, 836
Single Threaded Apartments, 833–836
threading models, 830–833

GUI message pumping
 CLR waits for managed code, 207
`CWaitForMultipleHandles`, 202–203
 deciding when, 203–204
`MsgWaitForMultipleObjects(Ex)`, 198–201
 overview of, 195–198
 using kernel objects, 188
 Gustafson's Law, 764

H

Hand over hand locking, 621–625
`handle(!)` command, 250–251
 Happens-before mechanism, 509–510
 Hardware
 architecture. *See* Parallel hardware architecture
 concurrency, 4
 for critical regions, 48
 interrupts, 84
 memory models, 509–511
 Hardware atomicity, 486–506
 interlocked operations. *See* Interlocked operations
 of ordinary of loads and stores, 487–492
 overview of, 486
 Hardware CAS (compare and swap)
 implementing critical regions with, 47
 instructions, 55–58
 reality of reordering, memory models, 58–60
 Hashtable based dictionary, 626–631
 Hashtable type, .NET, 627–631
 Hierarchy, concurrent programs, 6–7
 Holder types, C++, 262–263
 Homogeneous exceptions, collapsing, 728–729
 Hosts, CLR, 86, 298–299
 HT (HyperThreading) processor, 178, 277
`httpRuntime`, Vista thread pool, 381

I

I/O completion packets, 808
 I/O completion ports
 CLR thread pool, 368–371
 creating, 810–811
 legacy Win32 thread pool, 359–360
 overview of, 809–810
 as rendezvous method, 808–809
 thread pools and, 319–321

tricky synchronization with, 341–342
 and Vista thread pool, 334–336
 waiting for completion packets, 811–813
 I/O (Input/Output), 785–827
 .NET Framework asynchronous I/O, 817
 APC callback completion method, 806–808
 asynchronous device/file I/O, 817–819
 asynchronous I/O cancellation, 822–826
 asynchronous sockets I/O, 814–817,
 820–822
 blocking calls, 730
 completing asynchronous I/O, 796
 event handler completion method, 802–805
 I/O completion ports completion method,
 808–813
 initiating asynchronous I/O, 792–796
 overlapped I/O, 786–788
 overlapped objects, 788–792
 polling completion method, 798–800
 synchronous completion method, 797–798
 synchronous vs. asynchronous, 785–786
 wait APIs completion method, 800–802
 Win32 asynchronous I/O, 792
 I/O prioritization, 162
 IA64 architecture
 .NET Framework memory models,
 516–517
 hardware memory models, 509–511
 memory fences, 512
 IAsyncResult interface, APM
 defined, 399
 implementing APM with, 413–418
 overview of, 401–403
 rendezvousing with, 403–411
 IComponent interface, 422–423
 Ideal processor, 170, 179–180
 IdealProcessors, TaskManagerPolicy, 903
 IdealThreadsPerProcessor,
 TaskManagerPolicy, 903
 IDisposable, mutexes, 215
 ILP (instruction level parallelism), 479
 Immutability
 managing state with, 14
 overview of, 34
 protecting library using, 869
 single assignment enforcing, 34–38
 Increment statements, 23
 Incremental results, 425–426
 Infinite recursion, 140–141
 Initial count, semaphores, 42, 222

Initialization
 condition variables, 305
 critical sections, 257–258
 lazy. *See* Lazy initialization
 slim reader/writer locks, 290
 Windows Vista one-time, 529–534

InitializeCriticalSection, 258–259

InitializeCriticalSectionAndSpinCount, 258, 264–265, 267–268

InitializeCriticalSectionEx, 258–259, 264–266

Initialized thread state, 155

InitializeThreadpoolEnvironment, Vista, 343

initiallyOwned flag, mutexes, 214

Initiating asynchronous I/O, 792–796

InitOnceBeginInit, Vista, 531

InitOnceComplete, Vista, 531

InitOnceExecuteOnce, Vista, 529–534

INIT_ONCE, 529–534

INIT_ONCE-ASYNC, 532

Inline, 892

Input data, data parallelism, 657

Input/Output. *See* I/O (Input/Output)

Instant state, library, 868–869

Instruction level parallelism (ILP), 479

Instruction pointer (IP), 81–82

Instruction reordering, 479–480, 481–484

int value, WaitHandle, 206

Intel64 architecture, 509–511

Interlocked class, 494

Interlocked operations, 492–506

 128-bit compare exchanges, 500–502

 atomic loads and stores of 64-bit values, 499–500

bit-test-and-set/bit-test-and-reset, 502–503

compare and exchange, 496–499

controlling execution orders, 484

exchange, 493–496

other kinds of, 504–506

overview of, 56, 492–493

Interlocked singly-linked lists (SLists), 538–540

Interlocked.CompareExchange

 examples of low-block code, 535–536

 implementing 128-bit compare exchanges, 500–501

 implementing compare and exchange, 497–498

 lazy initialization in .NET, 526–527

 _InterlockedExchange, 493

 _InterlockedExchange64, 499

 _InterlockedExchangePointer, 495

Internal data structures, threads, 145–151

 checking available stack space, 148–151

 overview of, 145–146

 programmatically creating TEB, 146–148

Interprocess synchronization, 188

Interrupt instance method, 207

Interrupt Request Level (IRQL), DPCs, 84–85

Interrupts

 hardware, 84

 quantum accounting, 163–164

 software, 84–85

 waiting or sleeping threads, 207–208

IntPtrs, 90

Intraprocess isolation, 32

Invalid states, 20–21

InvalidWaitHandle, CLR thread pool, 374, 377

Invariants

 invalid states and broken, 20–21

 lock reliability and security, 876–877

 overview of, 547–548

 rules for recursion, 558

 static state access for libraries, 868

Invoke method, Parallel class, 904–909

IOCompletionCallback, 370

IP (instruction pointer), 81–82

IRQL (Interrupt Request Level), DPCs, 84–85

IsCompleted flag, APM, 404, 411, 416

ISO Common Language Infrastructure, 516–518

Isolation

 custom thread pools with, 387–391

 data ownership with, 33–34

 employing, 31–34

 managing state with, 14

 protecting library with, 869

ISupportsCancelation, 915

ISynchronizeInvoke, 838–839

Iterations

 continuous, 663–667

 data parallelism and, 659–661

 deciding to go parallel and, 756–757

 dynamic (on demand) decomposition, 669–675

 static decomposition and, 662–663

 striped, 667–669

itonly field modifier, 34–35

J

Java

- exiting and entering CLR locks, 274–275
- JSR133 memory model specification, 509–510

K

KD.EXE (Kernel Debugger), 251

Kernel

- fibers and, 430
- overview of, 183–184
- reasons to use for synchronization, 186–189
- support for true waiting in, 64–65
- synchronization-specific, 184
- Kernel Debugger (KD.EXE), 251
- Kernel-mode APCs, 208–209
- Kernel-mode stacks, 82
- Kernel synchronization
 - asynchronous procedure calls, 208–210
 - auto-reset and manual-reset events. *See* Auto-reset events; Manual-reset events
 - debugging kernel objects, 250–251
 - in managed code, 204–208
 - mutex/semaphore example, 224–226
 - overview of, 183–184
 - using mutexes, 211–219
 - using semaphores, 219–224
 - using sparingly, 253
 - waitable timers. *See* Waitable timers

- Kernel synchronization, signals and waiting, 184–204, 241–250
 - with auto-reset events, 244–248
 - CoWaitForMultipleHandles**, 202–203
 - example of, 243–244
 - with manual-reset events, 248–250
 - message waits, 195–198
 - MsgWaitForMultipleObjects(Ex)**, 198–202
 - overview of, 184–186, 241–243
 - reasons to use kernel objects, 186–189
 - waiting in native code, 189–195
 - when to pump messages, 203–204
- Keyed events, 268–270, 289
- KTHREAD, 145–146, 152

L

- Lack of preemption, 576, 577
- Lamport’s bakery algorithm, 54–55
- Latch, 66

Latent concurrency, 5, 867

Layers, parallelism, 8–10

Layout, stack memory. *See* Stack memory layout

lazy allocation, 267–268

Lazy futures, 689

Lazy initialization

- in .NET, 520–527

- in VC++, 528–534

LazyInit<T>, 917–919**LeaveCriticalSection**

- ensuring thread always leaves, 261–263

- fibers and, 449

- leaving critical section, 260–261

- leaving unowned critical section, 261

- low resource conditions and, 267–268

- process shutdown, 563–564

LeaveCriticalSectionWhenCallbackReturns, 350–351Leveled locks. *See* Lock leveling**LFENCE** (Load fence), 512

Libraries, designing reusable, 865–886

- blocking, 884–885

- further reading, 885

- locking models, 867–870

- major themes, 866–867

- reliability, 875–879

- scalability and performance, 881–884

- scheduling and threads, 879–881

- using locks, 870–875

Linear pipelines, 711

Linear speedups, 758–760

Linearizability, managing state with, 30–31

Linearization point, 30, 520

InitialCount parameter, 222

Linked lists, 617–620, 621–626

LINQ. *See* PLINQ (Parallel LINQ)**LIST_HEADER** data structure, 538–540

Livelocks

- concurrency causing, 11

- implementing critical regions without, 47

- overview of, 601–603

Liveness hazards, 572–609

- defined, 545

- livelocks, 601–603

- lock convoys, 603–605

- missed wake-ups, 597–601

- priority inversion and starvation, 608–609

- stampedes, 605–606

- two-step dance, 606–608

- Liveness hazards, deadlock, 572–597
 avoiding, 575–577
 avoiding with lock leveling, 581–589
 avoiding with The Banker’s Algorithm, 577–582
 detecting, 589–590
 detecting with timeouts, 594
 detecting with Vista WCT, 594–597
 detecting with Wait Graph Algorithm, 590–594
examples of, 572–575
- I**MaximumCount parameter, `CreateSemaphore`, 222
- Load-after-store dependence, 485
- Load balanced pipelines, 716–717
- Load fence (LFENCE), 512
- Loader lock, 116
- Loads
- .NET memory models and, 516–518
 - atomic, 487–492, 499–500
 - hardware memory models and, 511
 - imbalances, and speed-up, 765–766
- LocalDataStoreSlot, TLS, 123
- LocalPop, work stealing queue, 637
- LocalPush, work stealing queue, 637, 640
- Lock convoys, 165, 289, 603–605
- Lock free algorithms, 28
- Lock-free data structures, 632–640
- general-purpose lock free FIFO queue, 632–636
 - parallel containers and, 615
 - work stealing queue, 636–640
- Lock free FIFO queue, 632–636
- Lock free programming
- defined, 477
 - designing reusable libraries, 882
 - overview of, 517–520
- Lock free reading, dictionary (hashtable), 627–631
- Lock freedom, 518–519. *See also* Memory models and lock freedom
- Lock hierarchies. *See* Lock leveling
- Lock leveling
- avoiding deadlock with, 875–876
 - examples of using, 582–584
 - inconvenience of, 582
 - `LOCK_TRACING` symbol in, 589
 - overview of, 581
 - sample implementation in .NET, 584–589
- Lock ordering. *See* Lock leveling
- Lock ranking. *See* Lock leveling
- lock statement, 870
- `LockFreeQueue<T>` class, 632–636
- Locking models, libraries, 867–870
- documenting, 870
 - protecting instant state, 868–869
 - protecting static state, 867–868
 - using isolation and immutability, 869–870
- `LockRecursionPolicy`,
- `ReaderWriterLockSlim`, 294
- Locks. *See also* Interlocked operations
- as concurrency problem, 10
 - deadlocks without, 574–575
 - Mellor-Crummey-Scott (MSC), 778–781
 - and process shutdown. *See* Process shutdown, locks and in reusable libraries, 870–875
 - simultaneous multilock acquisition, 578–581
 - spin only, 772–778
 - two-phase protocols for, 767–769
 - as unfair in newer OSs, 217
- Locks, CLR, 272–287
- debugging monitor ownership, 285–287
 - defining, 254
 - entering and leaving, 272–281
 - monitor implementation, 283–285
 - overview of, 272
 - reliability and monitors, 281–283
- locks command (!), 271
- `LOCK_TRACING` symbol, lock leveling, 589
- Loop blocking, 678
- Loops
- data parallelism and, 659–661
 - deciding to go parallel and, 756–757
 - loop blocking, 678
 - mapping over input data as application of parallel loops, 675–676
 - Nesting loops, 677–678
 - prerequisites for parallelizing, 662
 - reductions and scans with, 678–681
- Low-cost, implementing critical regions with, 47
- Low-lock code examples, 520–541
- Decker’s algorithm, 540–541
 - lazy initialization, 520–527, 528–534
 - nonblocking stack and ABA problem, 534–537
- Win32 singly linked lists (Slists), 538–540
- Low resource conditions, 266–270, 290–291

- lpName argument, mutex**, 213
lpParameter argument
 converting threads into fibers, 438–439
CreateFiber(Ex), 435–437
CreateThread, 91
lpPreviousCount, ReleaseSemaphore,
 223–224
lpStartAddress, CreateThread, 91
lpThreadAttributes, CreateThread, 90
lpThreadId parameter, CreateThread API,
 92–93
LPVOID parameter
 converting threads into fibers, 438
CreateFiber(Ex), 436
CreateThread API, 91
LPVOID value, TLS, 118–119
lReleaseCount, ReleaseSemaphore, 223–224
- M**
- Managed code.** *See also CLR*
 aborting threads, 109–113
 APCs and lock reliability in, 878
 fiber support not available for, 429, 433
 kernel synchronization in, 204–208
 overview of, 85–87
 process shutdown, 569–571
 thread local storage, 121–124
 triggering thread exit, 103
 using CLR thread pool in. *See CLR*
 thread pool
Managed debugging assistant (MDA), 575
ManagedThreadId property, 101
Manual-reset events, 226–234
 creating and opening events, 228–230
 events and priority boosts, 232–234
 implementing queue with, 248–250
 overview of, 226–227
 setting and resetting events, 230–231
ManualResetEventSlim, 919–920
Map/reduce paradigm, 658
Mapping over input data, 675–676
Marshal-by-bleed, 279
MarshalByRefObject, 279
Marshal.GetLastWin32Error, 881
Maximum count, semaphores, 222
Maximum threads
 CLR thread pool, 379–382
 deadlocks from low, 382–385
 Vista thread pool, 344, 348, 353
MAXIMUM_WAIT_OBJECTS
 blocking and pumping messages, 202
 registering wait callbacks in thread pools,
 322–323
 waiting in Win32, 190
MaxStackSize
 creating threads in .NET, 99
 specifying stack changes, 132
TaskManagerPolicy, 903
MDA (managed debugging assistant), 575
Measuring, speedup efficiency, 761–762
Mellor-Crummey-Scott (MSC) locks, 778–781
Memory
 slim reader/writer locks and, 289
 stack layout. *See Stack memory layout*
 stack reserve/commit sizes and, 130–133
Memory consistency models, 506–520
 .NET memory models, 516–518
 hardware memory models, 509–511
 lock free programming, 518–520
 memory fences, 511–515
 overview of, 506–508
Memory fences, 511–515
 creating in programs, 513–515
 double-checked locking in VC++ and, 528
 hardware memory models and, 510
 interlocked operations implying, 492
 overview of, 511
 release-followed-by-acquire-fence hazard,
 515
 types of, 511–513
Memory load and store reordering, 478–486
 critical regions as fences, 484–485
 impact of data dependence on, 485–486
 overview of, 478–480
 what can go wrong, 481–484
Memory models and lock freedom, 506–543
 .NET memory models, 516–518
 defining, 59–60
hardware atomicity. *See Hardware*
 atomicity
 hardware memory models, 509–511
 lock free programming, 518–520
 low-lock code examples. *See Low-lock code*
 examples
 memory fences, 511–515
 memory load and store reordering, 478–486
 overview of, 477–478
Merging, PLINQ, 912–914
Message-based parallelism, 658, 719–720

- Message loops. *See* Message pumps
Message passing, 71–73
Message Passing Interface (MPI), 720
Message pumps
 GUI and COM, 195–198
 overview of, 830–833
MFENCE (full fence), 512–515
`m_head`, 535, 537
Microprocessor architectures, 178–179
Microsoft kernel debuggers, 271
Microsoft SQL Server, 433
Microsoft Windows Internals (Russinovich and Solomon), 145, 154
`minFreeThreads` element, `httpRuntime`, 384–385
Minimum threads
 CLR thread pool, 379–382
 delays from low, 385–386
 Vista thread pool, 344, 348, 353
`MinProcessors`, `TaskManagerPolicy`, 903
Missed pulses, 597–601
Missed wake-ups, 597–601
MMCSS (multimedia class scheduler service), 167
Modal loop, GUIs, 198
Modeling, 4
Monitor, creating fences, 514
`Monitor.Enter` method
 avoiding blocking, 275–277
 CLR locks, 272–273
 ensuring thread always leaves monitor, 273–275
 locking on `AppDomain` agile objects, 279
 reliability and CLR monitors, 281–283
 using value types, 277–278
`Monitor.Exit` method
 avoiding blocking, 275–277
 CLR locks, 272–273
 ensuring thread always leaves monitor, 273–275
 using value types, 277–278
Monitors, CLR
 avoiding blocking, 275–276
 exiting and entering, 272–275
 implementing, 283–285
 overview of, 272
 reliability and, 281–283
 using value types, 277–278
Monitors, .NET Framework, 68–70, 309–312
MPI (Message Passing Interface), 720
MSC (Mellor-Crummey-Scott) locks, 778–781
MSDN Magazine, 590
`MsgWaitForMultipleObjects(Ex)` API
 kernel synchronization, 198–202
motivation for using, 833
waiting for managed code, 207
MTAs (multithreaded apartments), 575, 834–835
`MTAThreadAttribute`, 835
`MultiLockHelper.Enter`, 578
Multimedia class scheduler service (MMCSS), 167
Mutants. *See* Mutexes
Mutexes, 211–219
 abandoned, 217–219
 acquiring and releasing, 216–217
 avoiding registering waits for, 376
 care when using APCs with, 210
 creating and opening, 212–216
 defined, 42
 designing library locks, 874
 example of semaphores and, 224–226
 overview of, 211–212
 process shutdown and, 564, 568, 571
 signaled/nonsignaled state transition, 186
 Vista thread pool completion tasks, 350–351
`mutexSecurity` argument, 214
Mutual exclusion mechanisms
 avoiding deadlocks with, 576
 causing deadlocks, 575
 data synchronization. *See* Critical sections, Win32; Locks, CLR
Dekker's and Dijkstra's algorithm, 50–53
executing interlocked operations, 492–493
hardware CAS instructions, 55–58
implementing critical regions, 47–48
Lamport's bakery algorithm, 54–55
Peterson's algorithm, 53–54
strict alternation, 49–50
`m_value` class, 521–527
`MWMO-WAITALL` value, 202
“Myths about the Mutual Exclusion”, Peterson, 53

N

- NA (neutral apartments), 834–835
Natural scalability, of algorithms, 760–761
Nested parallelism, 757

- Nesting loops, data parallelism and, 677–678
.NET Framework
 avoiding building locks, 873
 creating fences, 98–101, 514
 creating threads, 152–153
 dictionary (hashtable), 626–631
 event-based asynchronous pattern in, 426–427
 legacy reader/writer lock, 300–304
 memory models, 516–518
 monitors, 309–312
 slim reader/writer lock (3.5), 293–300
 synchronization contexts, 853–854
 terminating threads. *See* Threads,
 termination methods
 timers, 373
 using APM in, 418–419
.NET Framework Asynchronous GUI
 asynchronous operations, 855–856
 `BackgroundWorker` package, 856–860
 overview of, 837
 synchronization contexts, 847–854
 Windows Forms, 837–840
 Windows Presentation Foundation, 840–846
.NET Framework asynchronous I/O
 asynchronous device/file I/O, 817–819
 asynchronous sockets I/O, 820–822
 I/O cancellation, 823
 overview of, 817
Neutral Apartments (NA), 834–835
`new Singleton()` statement, 521, 524
`NodeInfoArray`, WCT, 596
Non-const pointer, 36–38
Non-Uniform Memory Access (NUMA)
 machines, 178–179
Nonatomic software, 22
Nonblocking programming. *See also* Lock-free data structures
 ABA problem, 536–537
 defined, 477
 implementing custom nonblocking stack, 534–536
 parallel containers and, 615
 Win32 singly linked lists, 538–540
Nonlinear pipelines, 711
Nonlocal transfer of control, in Windows, 84
Nonsignaled events, 67
`NotBuffered` merge, PLINQ, 913
NP-hard problems, parallelism, 718
- `_NT_TIB`, 146–148
NULL value, `CreateThread` failure, 92
NUMA (Non-Uniform Memory Access) machines, 178–179
- ## O
- Object header inflation, 284–285
Object headers, CLR objects, 283–285
Object invariants, 548
object state argument, TPL, 890
Objects, overlapped, 788–792
Obstruction freedom, 518
128-bit interlocked operations, 500–502
Online debugging symbols, 139
`OpenEvent(Ex)` APIs, 228–230
`OpenExisting` method
 closing mutexes, 215–216
 opening events, 230
 opening existing semaphore, 221
`OpenSemaphore`, 220–222
`OpenThread`, 95
`OpenThreadWaitChainSession`, WCT, 595–596
Optimistic concurrency, 625–626
Order preservation, PLINQ, 914–915
Orderly shutdown, 569–570
Orphaned locks, 45, 561–562
Orphaning, abandoned mutexes and, 218
OS threads, 879–880
`OutOfMemoryException`, 143
Output dependence, 485–486
Overflow, stack, 140–145
Overlapped class, 369–370
 CLR thread pool I/O completion ports, 369–371
Overlapped I/O. *See also* Asynchronous I/O
 overlapped objects, 788–792
 overview of, 786–788
Overtaking race, 654
Ownership
 asserting lock, 872
 CLR thread pool and, 377
 debugging CLR monitor ownership, 285–287
 debugging legacy RWLs, 303–304
 defined, 32
 mutex, 211–212
 overview of, 33–34
 Vista thread pool, 352–353

P

- P/Invoking, 881
P (taking), semaphores, 42
Pack method, CLR thread pool, 370
PAGE_GUARD attribute, 134, 137
Parallel class, TPL, 904–908
Parallel containers, 613–655
 approaches to, 614–616
 coordination containers, 640–650
 fine-grained locking, 616–632
 lock-free data structures, 632–640
 phased computations with barriers, 650–654
 sequential containers vs., 613–614
Parallel execution
 cancellation, 729–731
 concurrent exceptions, 721–729
 data parallelism. *See* data parallelism
 message-based parallelism, 719–720
 overview of, 657–659
 task parallelism. *See* Task parallelism
Parallel extensions to .NET, 887–930
 concurrent collections, 924–929
 further reading, 930
 overview of, 887–888
 parallel LINQ, 910–915
 synchronization primitives. *See* Synchronization primitives
 TPL. *See* TPL (task parallel library)
Parallel hardware architecture, 736–756
 cache coherence, 742–750
 cache layouts, 740–742
 locality, 750–751
 memory hierarchy, 739
 overview of, 736
 profiling in Visual Studio, 754–756
 sharing access to locations, 751–754
 SMP, CMP, and HT, 736–738
 superscalar execution, 738–739
 UMA vs. NUMA, 740
Parallel LINQ. *See* PLINQ (Parallel LINQ)
Parallel merge-sort, 681–684
Parallel quick-sort, 681
Parallel traversal, 613
ParallelEnumerable class, PLINQ, 910–912
Parallelism
 deciding to go parallel, 756–758
 defined, 80
 designing reusable libraries, 866–867
 layers of, 8–10
 measuring improvement due to, 758
 overview of, 5
 structured, 70–71
ParameterizedThreadStart, 99
Parents, task parallel library, 895–897
Partitioning, 912
PE (portable executable) image, 131–132
peb (!) command, 146
PEB (process environment block), within TEB, 145
PeekMessage, 198–200
Performance
 Amdahl's Law, 762–764
 critical paths, 764–765
 deciding to go parallel, 756–758
 designing reusable libraries, 881–884
 garbage collection and scalability, 766–767
 Gustafson's Law, 764
 interlocked operations, 493, 505–506
 load imbalances and, 765–766
 measuring improvement due to parallelism, 758
 measuring speedups and efficiency, 760–762
 Mellor-Crummey-Scott (MSC) locks, 778–781
 natural scalability vs. speedups, 760–761
 overview of, 735–736
 parallel hardware architecture. *See* Parallel hardware architecture
 ReaderWriterLockSlim, 299
 recursive lock acquires, 872
 speedups and efficiencies and, 756
 spin-only locks, 772–778
 spin waiting and, 766–772
 tuning quantum settings, 163
 types of speedups, 758–760
Performance counters, querying thread state, 156–157
Periodic polling, 730
Persistent threads, Vista thread pool, 352–353
Pervasive concurrency, 865
Peterson's algorithm, 53–54
Phased computations with barriers, 650–654
Pi-calculus, 72
Pipelines
 defined, 541
 generalized data structure, 712–716

- Pipelines, *continued*
 load balanced, 716–717
 overview of, 709–712
 pipelining output of futures or promises, 698–702
- PLINQ (Parallel LINQ)
 buffering and merging, 912–914
 defined, 887
 order preservation, 914–915
 overview of, 910–912
- Pointer size values, store atomicity and, 487
- Polling
 asynchronous I/O completion, 798–800
 canceling periodic, 730
- Pollution, thread, 352, 377
- Portable executable (PE) image, 131–132
- Postconditions, as invariants, 548
- Preconditions, as invariants, 547
- Predictability, GUI, 836
- Predictability, of responsive GUIs, 836
- Preemptive scheduling, 83, 154–155
- Prerender event, ASP.NET, 421
- Priorities
 custom thread pool with, 387–391
 lock reliability and, 878
 quantum adjustments and, 164–167
 thread scheduling, 159–163
- Priority boosts, 84, 232–234
- Priority class, 159–160
- Priority inheritance, 609
- Priority inversion, 608–609, 610, 878
- Priority level, 159
- Priority, Thread class, 160
- PriorityClass, Process, 159
- PriorityLevel, ProcessThread, 160–161
- Private state, shared state vs., 15–19
- Privatization, 15–16, 33
- ProbeForStackSpace method, 145
- ProbeForSufficientStack, 144, 149
- Probes, stack, 143–145
- Process affinity masks, CPU affinity, 173–174
- Process class, 159, 175
- Process environment block (PEB), 145
- Process exit, threads, 113–115
- Process isolation, 31
- Process shutdown, locks and, 561–571
 managed code, 568
 managed code and, 569–571
 overview of, 561–563
 Win32: weakening and termination, 563–568
- Processes
 assigning CPU affinity to, 171–175
 Windows vs. UNIX, 80–81
- ProcessExit event, CLR, 569–570
- ProcessorAffinity, CPU affinity, 175
- Processors
 concurrency in modern, 5
 creating fences at level of, 512–515
 relationship between fibers, threads and, 438
- ProcessPriorityClass, 159
- ProcessThread class, 98, 160–161
- Producer/consumer containers, 614
- Producer/consumer relationship, 641–642
- Profilers, thread suspension in, 170
- Program order, 480–484
- Programming Windows (Petzold), 198
- Programs, naturally scalable, 5
- Progress reporting, 425–426
- ProgressChangedEventHandler, 426
- Promise style future, 900
- Promises
 building dataflow systems, 693–695
 pipelining output of, 698–702
- Properties, ReaderWriterLockSlim, 295
- Pseudo-handles, CreateThread, 94–95
- PTEB structure, 146
- Publication, data ownership and, 33
- Pulse
 .NET Framework monitors, 310
 missed wake-ups, 598–601
 two-step dance problems, 608
- PulseAll
 .NET Framework monitors, 310
 missed wake-ups, 598–601
 two-step dance problems, 608
- PulseEvent API, 231
- Pulsing, .NET Framework monitors, 310
- Pump messages, GUI and COM, 195–204
 CoWaitForMultipleHandles API, 202–203
 deciding when to pump messages, 203–204
- MsgWaitForMultipleObjects(Ex), 198–201
 overview of, 195–198

Q

- Quantums, 83, 163–167
- QueueUserWorkItem
 APM, 402–403
 CLR thread pool, 371

- legacy thread pool, 354–356, 363
ThreadPool class, 364–366
QueueWork functions, user-mode scheduler, 463–464
- ## R
- Race conditions (data races), 546–555
benign, 553–555
composite actions and, 550–553
concurrency causing, 10
eliminating with critical regions, 40
famous bugs due to, 610
inconsistent synchronization and, 26, 549–550
invariants and, 548
in library code, 874–875
overview of, 546–549
patterns of critical region usage, 43–45
reasons for, 26–27
two-step dance problems due to, 607–608
Radix sort, algorithms, 681
Random access, linked lists, 621
Randomized backoff, 602–603
RCWs (runtime callable wrappers), 575
Reactive systems, 61
Read-only synchronization, 881–882
Read/read hazards, 28, 34
Read/write hazards, 28
 ReadBarrier, 529
Reader/writer locks. *See* RWLs
 (reader/writer locks)
ReaderWriterLock
 as legacy version, 300–304
 motivating development of new lock, 299–300
 overview of, 293–294
 for read-only synchronization, 881–882
 reliability limitation, 298
ReaderWriterLockSlim
 creating fences using, 514
 motivation for, 299–300
 overview of, 293–294
 process shutdown, 565
 recursive acquires, 297–298
 reliability limitation, 298–299
 three modes of, 294–295
 upgrading, 296–297
ReadFile, 792
readonly fields, single assignment, 35–36
- readonly keyword, single assignment, 35
Ready thread state, 155
Recursion
 avoiding lock, 872
 detecting in spin waiting, 773–775, 777
 reentrancy and, 555–558
 rules controlling, 558
 task parallelism and, 702–709
Recursive acquires
 avoiding lock, 872
 example of, 557–558
 mutex support for, 217
 overview of, 556–557
 ReaderWriterLockSlim, 297–298
 SRWLocks non-support for, 292–293
 using, 558–561
Recursive algorithms, 558–559
Recursive locks, 556
RecursiveReadCount, ReaderWriterLockSlim, 295
RecursiveUpgradeCount,
 ReaderWriterLockSlim, 295
RecursiveWriteCount,
 ReaderWriterLockSlim, 295
Reduction, in data parallelism, 678–681
Reentrancy
 caused by pumping, 203
 concurrency causing, 11
 lock reliability and, 877–878
 overview of, 555–556
 system introduced, 559–561
Registered waits
 CLR thread pool, 374–377
 legacy Win32 thread pool, 360–363
 thread pools and, 322–323
 Vista thread pool, 336–341
RegisteredWaitHandle, CLR, 376
RegisterWaitForSingleObject
 building user-mode scheduler, 466–467
 CLR thread pool, 375
 legacy thread pool, 360–361
Relative priority, individual threads, 159
Release fence, 512
Release-followed-by-acquire-fence
 hazard, 515
releaseCount argument, 224
ReleaseLock, legacy RWLs, 301
ReleaseMutex, 215–216
ReleaseMutexWhenCallbackReturns, 350
ReleaseSemaphore, 223–224

`ReleaseSemaphoreWhenCallbackReturns`, 351
`ReleaseSRWLockExclusive`, 290, 293
`ReleaseSRWLockShared`, 290, 293
Reliability
 designing library locks, 875–879
 designing reusable libraries, 875–879
 lock freedom and, 519–520
Remove, dictionary, 631
Rendezvous methods, asynchronous I/O
 APC callback, 806–808
 event handler, 802–805
 I/O completion ports, 808–813
 overview of, 792, 796
 polling, 798–800
 synchronous, 797–798
 wait APIs, 800–802
Rendezvous patterns, ATM, 403–405
Reserve size, threads
 creating stack overflow, 140–145
 overview of, 130–133
 stack memory layout, 138
ResetEvent, 230
`_resetstkoflw`, 143
Responsiveness, GUI, 834–836
RestoreLock, legacy RWLs, 301
Resume, `Thread` class, 140
ResumeThread, 91
ResumeThread, 169
 retirement algorithm, 378–379
Rude shutdowns, 563
Rude thread aborts, 112
Run method, 831
RunClass Constructor, 877–878
Running state, threads, 155, 158–159
Runtime callable wrappers (RCWs), 575
Runtime, fibers and CLR, 450–453
RuntimeHelpers.ProbeForSufficientStack, 144, 149
RuntimeHelpers.RunClass Constructor, 877–878
RWLs (reader/writer locks), 287–304
 .NET Framework legacy, 300–304
 .NET Framework slim, 293–300
 defined, 28
 defining, 254–255
 overview of, 287–289
 read-only synchronization using, 881–882
Windows Vista SRWL, 288, 289–293

S

SafeHandles, 90
Scalability
 asynchronous I/O and, 787–788
 designing reusable libraries for, 881–884
 garbage collection and, 766–767
 of parallel algorithms, 666
 speedups vs. natural, 760–761
Scalable access, of parallel containers, 613
Scans, and data parallelism, 681
Schedules, `thread`, 878–879
Scheduling, 879–881. *See also Thread scheduler*, Windows; Thread scheduling
Search algorithms, 718–719, 730
Security
 creating threads in .NET, 99
 creating threads in Win32, 90
 using kernel objects, 188
SEH (structured exception handling), 104–106, 721
Self-replication, TPL, 909–910
Semaphores, 219–226
 creating and opening, 220–222
 designing library locks, 874
 mutex/semafore example, 224–226
 overview of, 42, 219–220
 signaled/nonsignaled state transition, 186
 taking and releasing, 223–224
 Vista thread pool completion tasks, 351
 waiting and, 185
SemaphoreSlim, 920–921
Sense-reversing barriers, 650
Sentinel nodes, FIFO queues, 617–618
Sequential programming, 727–728
Serializable, 30
Serializable history, 25
Serialized threads, 25
Servers, garbage collection, 766–767
SetCriticalSectionSpinCount, 264–265
SetData, TLS, 123
SetErrorMode, 105
SetEvent, 230
SetMaxThreads, Vista, 381
SetPriorityClass, 159
SetProcessAffinityMask, CPU affinity, 173–175
SetThreadAffinityMask, CPU affinity, 174
SetThreadContext, 151

`SetThreadpoolCallbackRunLong`, Vista, 349–350
`SetThreadPoolMaximum`, legacy, 363
`SetThreadPoolMaximum`, Vista, 344, 348, 353
`SetThreadPoolMinimum`, Vista, 344–345, 348, 353
`SetThreadpoolTimer`, Vista, 330–333
`SetThreadpoolWait`, Vista, 337–338, 340
`SetThreadPriority`, 160, 162, 352
`SetThreadPriorityBoost`, 165
`SetThreadStackGuarantee`, 134–135, 136–137, 142
`SetWaitableTimer`, 236–237
SFENCE (store fence), 512
Shallow immutable objects, 34
Shared mode, `ReaderWriterLockSlim`, 294–295
Shared resources, among threads, 80–81
Shared state, 14–19
`SharedReaderLock` method, 300
`SharedWriterLock` method, 300
Shutdown, building UMS, 470–472
Shutdown method, 470–471
Signaled events, 67
Signaled, vs. nonsignaled kernel objects, 184–185
`SignalObjectAndWait`
 blocking queue data structure with auto-reset, 244–248
 blocking queue data structure with events, 243–244
 overview of, 241–243
`SimpleAsyncResult` class, APM, 413–418
Simultaneous multilock acquisition, 578–581
Single assignment, 34–38
Single threaded apartments. *See* STAs (single threaded apartments)
`Singleton` class, 521–523
64-bit Values, 499–500
`Sleep` API, 168
`SleepConditionVariableCS`, 305–306
`SleepConditionVariableSRW`, 305–306
`SleepEx` API, 168
Sleeping
 condition variables and, 305–307
 thread scheduling and, 167–168
Slim reader/writer locks. *See* SRWLs (slim reader/writer locks)
`SLIST_ENTRY` data structure, 538–540
SLists (singly linked lists), 538–540
Socket class, APM, 419
Sockets
 asynchronous sockets I/O in .NET, 820–822
 asynchronous sockets I/O in Win32, 814–817
Software interrupts, 84–85
`someLock`, 598–601
Sort key, simultaneous multilock acquisition, 579–581
Sorting, 681–684
SOS debugging extensions, 285–287, 386–387
`SoundPlayer`, `System.dll` assembly, 427
Speculative search algorithms, 719
Speedup
 Amdahlís Law, 762–764
 critical paths, 764–765
 deciding to igo parallelí, 756–758
 garbage collection and scalability, 766–767
 Gustafsonís Law, 764
 load imbalances and, 765–766
 measuring, 758, 761–762
 natural scalability vs., 760–761
 overview of, 756
 types of, 758–760
Spin locks
 building, 921–923
 difficulty of implementing, 769
 Mellor-Crummey-Scott, 778–781
 for performance scalability, 873, 883
 on Windows, 769–772
Spin-only locks, 772–778
Spin waiting
 avoiding blocking in CLR locks, 276–277
 avoiding blocking in critical sections, 264–266
 avoiding hand coding, 882
 defining, 63–64
 Mellor-Crummey-Scott (MSC) locks, 778–781
 overview of, 767–769
 spin-only locks and, 772–778
 SRWLs, 290
 Windows OSs and, 769–772
`SpinLock`, 921–923
`SpinWait`, 923–924
Spurious wake-ups, 311–312, 598
SQL Server, fiber-based UMS, 86–87
`SqlCommand` type, APM, 419
`SRWLOCK`, 290–292
`SRWLock`, 565–567

- SRWLocks (slim reader/writer locks)
 .NET Framework, 293–300
 integration with Windows Vista condition variables, 304–309
 Windows Vista, 288, 289–293
- SSA (static single assignment), 34–38
- Stack limit, 133, 135–138
- Stack memory layout, 133–140
 example of, 135–138
 guaranteeing committed guard space, 134–135
 overview of, 133–134
 stack traces, 138–140
- Stack space, 133, 135–138
`/STACK` switch, 132
- Stack traces, 138–140
`stackalloc` keyword, 141
- StackBase field, TEB, 147, 149
- StackLimit field, TEB, 147, 149
- `StackOverflowException`, 142
- Stacks
 ABA problem and, 536–537
 creating new fibers, 436
 implementing custom nonblocking, 534–536
 overflow, 140–145
 overview of, 82–83
 reservation and commit sizes, 130–133
 user-mode, 127–130
- `StackTrace` class, 140
- Stale read, 28
- Stampedes, 605–606
- Standby thread state, 155–156
- START command, CPU affinity, 175
- Start method, `Thread` class, 99
- StartNew methods, TPL, 890
- `StartThreadpoolIo` function, Vista, 335–336
- Starvation, 608–609, 878
- STAs (single threaded apartments)
 deadlocks and, 574–575
 overview of, 833–836
 system introduced reentrancy and, 560–561
- State, 14–38
 atomicity, 29–30
 broken invariants and invalid states, 20–21
 in concurrent programs, 6–8
 dependency, 61–62
 fiber execution and, 430–431
 general approaches to, 14
 identifying shared vs. private, 15–19
- immutability, 34–38
 isolation, 31–34
 linearizability, 30–31
 overview of, 14–15
 serializability, 30
 simple data race, 22–29
 state machines and time, 19–20
 thread. *See Thread state*
- `STAThreadAttribute`, 835
- Static decomposition
 continuous iterations and, 663
 data parallelism and, 662–663
 flaws in, 666
- `static` methods, `BlockingCollection<T>`, 927–928
- Static single assignment (SSA), 34–38
- Static TLS, 118, 120–122
- static variables, 867–868
- `STATUS_GUARD_PAGE_VIOLATION` exception, 134
- `std::iterator` objects (C++), 672
- stopped state, threads, 158
- Store-after-load dependence, 486
- Store-after-store dependence, 485–486
- Store atomicity, 487–492
- Store fence (`SFENCE`), 512
- Stores
 .NET Framework memory models, 516–518
 of 64-bit values, 499–500
 atomic, 487–492, 499–500
 hardware memory models and, 510
- `Stream` class, APM, 419
- Strict alternation
 Dekker's algorithm vs., 50–51
 failure of in modern processors, 58–59
 overview of, 49–50
- Striped iterations, 667–669
- Striping, 614–615
- `strtok` function, 96
- Structured exception handling (SEH), 104–106, 721
- Structured fork/join, 687
- Structured parallelism, 70–71
- Structured tasks, 896
- Sublinear speedups, 758–760
- `SubmitThreadpoolWork` API, Vista, 326–330
- Superlinear speedups, 719, 758–760
- `Suspend`, `Thread` class, 140
- Suspended state, threads, 158–159
- `SuspendThread`, 169

- Suspension, thread
overview of, 91
stack trace and, 140
using in scheduling, 168–170
Swallowing exceptions, CLR, 105
`SwitchToFiber`, 440–441, 443–445, 466
`SwitchToThread` API, 168
Synchronizes-with mechanism, 509–510
Synchronization
inconsistent, 549–550
lock free vs. lock-based algorithm and, 519
never using thread suspension for, 170
synchronization contexts in .NET, 853–854
synchronization contexts in Windows, 847–853
torn reads from flawed, 490
two-phase locking protocols, 767–769
Vista thread pool, 341–342
Windows kernel. *See* Kernel
 synchronization
Synchronization and time, 13–75
control. *See* Control synchronization
data. *See* Data synchronization
managing program state. *See* State
overview of, 13–14, 38–40
Synchronization burden, 7–8
Synchronization primitives, 915–924
 `CountdownEvent`, 915–917
 `ISupportsCancelation`, 915
 `LazyInit<T>`, 917–919
 `ManualResetEventSlim`, 919–920
 `SemaphoreSlim`, 920–921
 `SpinLock`, 921–923
 `SpinWait`, 923–924
 `SynchronizationContext`, 830, 837, 847–854
Synchronous aborts, 109, 111
Synchronous completion method, 797–798
Synchronous I/O, asynchronous I/O vs., 795
Synchronous I/O cancellation, 823, 824–825
`SyncLock`, 607
Synclock keyword, 274, 277–278
System affinity mask, 172–173
System introduced reentrancy, 559
System registry key, 163
- T**
- `targetLock`, 592
- Task parallel library. *See* TPL (task parallel library)
- Task parallelism, 684–719
dataflow parallelism, 689
defined, 658
fork/join parallelism, 685–688
futures used to build dataflow systems, 689–692
generalized pipeline data structure, 712–716
load balanced pipelines, 716–717
overview of, 684–685
pipelines, 709–712
pipelining output of futures or promises, 698–702
promises, 693–695
recursion, 702–709
resolving events to avoid blocking, 695–698
search algorithms and, 718–719
`TaskCreationOptions` enum, 891
`TaskManagerPolicy`, TPL, 902–904
`TaskManagers`, TPL
defined, 890
overview of, 902–904
TATAS locks, 778
Taxonomy
concurrent program structure, 6–8
parallelism, 9
`TEB` address, 121
`TEB` (thread environment block)
checking available stack space, 148–150
as internal data structure, 145–146
printing out information, 146
programmatically accessing, 146–148
stack memory layout, 135–138
thread creation details, 152
thread scheduling and, 881
thread state and, 127
Temporary boosting, 164–167
Terminated thread state, 156
`TerminateProcess` API
shutting down thread with brute force, 103
terminating process with, 563
terminating threads in Win32, 113
Windows Vista shutdown, 564
`TerminateThread` API
abrupt termination with, 113–114, 153–154
overview of, 107–109
specifying return code at termination, 94
Termination, thread. *See* Threads,
 termination methods
Testing, wait condition inside locks, 878–879

- The Banker's Algorithm, 577–581
- Thin lock, 284
- Third party in-process add-ins, 563
- Third party locks, 873
- Thread affinity
 - defined, 87
 - designing reusable libraries, 866
 - fibers and, 433, 447–449
 - fibers and CLR, 452–453
- Thread blocks. *See* *Blocks, thread*
- Thread class, 98–101, 132, 160
- Thread coordination, 60–73
 - control synchronization and, 60–62
 - events, 66–68
 - message passing, 71–73
 - monitors and condition variables, 68–70
 - state dependence among threads, 61–62
 - structured parallelism, 70–71
 - waiting for something to happen, 63–66
- Thread environment block. *See* *TEB (thread environment block)*
- Thread information block (TIB), 145
- Thread injection, 378–379
- Thread local storage. *See* *TLS (thread local storage)*
- Thread management
 - legacy Win32 thread pool, 363–364
 - Vista thread pool, 347–350
- Thread management, CLR thread pool, 377–386
 - deadlocks from low maximum, 382–385
 - delays from low minimum, 385–386
 - minimum and maximum threads, 379–382
 - thread injection and retirement algorithm, 378–379
- Thread pools, 315–398
 - CLR. *See* *CLR thread pool*
 - I/O callbacks, 319–321
 - introduction to, 316–317
 - legacy Win32. *See* *Win32 legacy thread pool*
 - overview of, 315–316
 - performance improvements of, 391–397
 - registered waits, 322–323
 - timers, 321–322
 - UMS scheduler vs., 454
 - using explicit threading vs., 88
- Windows Vista. *See* *Windows Vista thread pool*
- work callbacks, 319
- writing own, 318–319
- Thread safety, 662
- Thread scheduler, Windows
 - advantages of fibers, 432
 - blocks and, 83–84
 - CPU affinity, 170–179
 - defined, 81–82
 - disadvantages of fibers, 433–434
 - functions of, 83
 - ideal processor, 179–180
 - priority and quantum adjustments, 164–167
 - priority based, 155
 - programmatically creating threads, 89
- Thread scheduling, 154–180
 - advantages of fibers, 432
 - CPU affinity, 170–179
 - disadvantages of fibers, 433–434
 - ideal processor, 179–180
 - multimedia scheduler, 167
 - overview of, 154–155
 - priorities, 159–163
 - priority and quantum adjustments, 164–167
 - quanta, 163–164
 - sleeping and yielding, 167–168
 - suspension, 168–170
 - thread states, 155–159
- Thread start routine, 89–90, 103
- Thread state, 127–145
 - defined, 158
 - stack memory layout, 133–140
 - stack overflow, 140–145
 - stack reservation and commit sizes, 130–133
- thread scheduling and, 155–159
- user-mode thread stacks, 127–130
- ThreadAbortException*, 104
- Threading models, GUI
 - overview of, 830–833
 - single threaded apartments (STAs), 833–836
- ThreadInterruptedException*, 208
- Thread.Join*, 100–101, 885
- Thread.MemoryBarrier*, 514
- !threadpool SOS extension command*, 386–387
- ThreadPriority_TaskManagerPolicy*, 903
- Threads, 79–125
 - asynchronous I/O cancellation for any, 825–826
 - asynchronous I/O cancellation for current, 823–824
- CLR, 85–87

- contexts, 151–152
- converting into fibers, 438–439
- creating, 152–153
- creating and deleting in Vista thread pool, 347–350
- designing reusable libraries, 879–881
- determining whether fibers are, 439–440
- DLLMain function, 115–117
- explicit threading and alternatives, 87–88
- fibers vs., 430–431
- internal data structures, 145–151
- local storage, 117–124
- marshaling exceptions across, 721–724
- overview of, 79–81
- programmatically creating, 89–90
- programmatically creating in C programs, 96–98
- programmatically creating in .NET Framework, 98–101
- programmatically creating in Win32, 90–96
- routines, user-mode scheduler, 459–460
- scheduling, 154–180
- state. *See* Thread state
- synchronous I/O cancellation for, 824–825
- terminating, 153–154
- Windows, 81–85
- Threads, termination methods, 101–114
 - defined, 83
 - details of, 153–154
 - `ExitThread` and `TerminateThread`, 107–109
 - overview of, 101–103
 - process exit, 113–115
 - returning from thread start routine, 103
 - thread aborts for managed code, 109–113
 - unhandled exceptions, 103–106
- `Thread.Sleep` API, 167–168, 882–883, 885
- `ThreadState` property, 157
- `ThreadStaticAttribute` type, TLS, 121–122
- `Thread.VolatileRead` method, 514
- `ThreadWorkRoutine` method, building UMS, 459–460
- Thresholds, stopping parallel recursion, 706
- TIB (thread information block), 145
- `TimeBeginPeriod` API, 168
- `TimeEndPeriod` API, 168
- Timeouts
 - .NET Framework monitors, 309–310
 - calling `AsyncWaitHandles.WaitOne`, 407–410
- condition variables, 306
- detecting deadlocks, 594
- Timer class, 371–374
- Timer class, CLR thread pool, 372–374
- Timer queue, 356–359
- TimerCallback, CLR thread pool, 372
- Timers. *See also* Waitable timers
 - CLR thread pool, 371–374
 - legacy Win32 thread pool, 356–359
 - overview of, 321–322
 - Vista thread pool, 330–334
- Timeslice, 83. *See also* Preemptive scheduling
- TimeSpan value, `WaitHandle` class, 206
- Timing, and concurrent programs, 24–29
- TLS (thread local storage), 117–124
 - accessing through .NET Framework, 880
 - creating threads in C programs, 96
 - fiber local storage vs., 445–447
 - managed code, 121–124
 - overview of, 117
 - Win32, 118–121
- `TlsAlloc` API, 118–119
- `TlsFree` function, 119
- `TlsGetValue` API, 118–119
- `TLS_OUT_OF_INDEXES` errors, 118–119
- `TlsSetValue` API, 118–119
- Torn reads, 487–490, 491–492
- TPL (task parallel library), 888–910
 - cancellation, 897
 - continuations, 900–902
 - defined, 887
 - futures, 898–900
 - overview of, 888–893
 - parents and children, 895–897
 - putting it all together, 904–909
 - self-replication, 909–910
 - task managers, 902–904
 - unhandled exceptions, 893–895
- `TP_TIMER` objects, 330–331
- `TP_WORK` objects, 326–328, 330–334
- Traces, stack, 138–140
- Transfer, of data ownership, 33–34
- Transition thread state, 156
- Transitive causality, 483, 511
- `TreadAbortExceptions`, 110
- `Tread.ResetAbort` API, 110
- True dependence, 485
- True waiting, 64–65
- Try/finally block, 273–275
- `TryAndPerform` method, linked lists, 621, 624

T
TryEnter method, CLR locks, 275–276
TryEnterCriticalSection, 263–266
TrySignalAndWait, 653–654
TrySteal, work stealing queue, 637, 639–640
TrySubmitThreadpoolCallback API, Vista thread pool, 324–328
Two-phase locking protocols, 767
Two-step dance, 606–608
Type objects, 278–281, 873–874
TypeLoadException, 492

U

ULONG, 134
UMS (user-mode scheduler)
advantages of fibers, 431–432
defined, 430
UMS (user-mode scheduler), building, 453–473
context switches, 464–470
cooperative blocking, 461–463
dispatching work, 461
fiber pool data structures, 455–459
overview of, 453–455
queueing work, 463–464
shutdown, 470–472
stack vs. stackless blocking, 472–473
thread and fiber routines, 459–460
Unhandled exceptions
overriding default behavior, 105–106
task parallel library, 893–895
terminating threads, 103–105
UnhandledExceptionsAreFatal flag, TPL, 893
UNIX, 80
UnregisterWait(Ex), 362–363
Unrepeatable reads, 28
UnsafePack, CLR thread pool, 370
UnsafeQueueUserWorkItem, CLR thread pool, 364–366, 371
UnsafeRegisterWaitForSingleObject, CLR thread pool, 375
Unstarted thread state, 157
Unstructured concurrency, 896–897
Upgrading
legacy RWLs, 302–303
ReaderWriterLockSlim, 294–297
User experience, and concurrency, 4
User-mode APCs, 208, 209–210
User-mode scheduler. *See* UMS (user-mode scheduler)

User-mode scheduling, 87
User-mode stacks, 82
allocated when creating new fibers, 436
overview of, 127–130
reservation and commit sizes of, 130–133
thread creation and, 153

V

V (releasing), semaphores, 42
!vadump command, 135–138
VADUMP.EXE, 135
VB SyncLock statement, 870
VC++
creating fences in, 514–515
process shutdown, 565–567
Virtual memory, 130–133
VirtualAlloc function, 138, 143
VirtualQuery Win32 function, 149–151
volatile variable
creating fences, 513–514
interlocked operations, 494
lazy initialization in .NET, 524–525

W

Wait APIs, 800–802
Wait Chain Traversal (WCT), Windows Vista, 590, 594–597
Wait conditions, 878–879
Wait freedom, 518
Wait graphs, 589–594
Wait method, Task class, 892–893
WAIT_ABANDONED value
abandoned mutexes, 218–219
blocking and pumping messages, 199
process shutdown, 564, 568
waiting in Win32, 190–191
Waitable timers, 234–241
creating and opening, 235–236
overview of, 234–235
setting and waiting, 236–237
using FILETIMEs, 237–241
WAIT_ALL flags, 231–232
WaitAll, **WaitHandle** class, 205–206
WaitAny, **WaitHandle** class, 205–206
WAIT_FAILED, 190–191, 199
WaitForMultipleObjects(Ex) APIs
acquiring and releasing mutexes, 216
alertable waits, 193–195

building user-mode scheduler, 466–467
taking and releasing semaphores, 223–224
waiting in Win32, 190–193

WaitForSingleObject(Ex) APIs
abandoned mutexes and, 218
acquiring and releasing mutexes, 216
alertable waits, 193–195
taking and releasing semaphores, 223–224
waiting in Win32, 189–190

WaitForThreadpoolCallbacks, Vista, 328–330

WaitForThreadpoolTimer, Vista, 334

WaitForThreadpoolTimerCallbacks, Vista, 334

WaitForThreadpoolWaitCallbacks, Vista, 339, 341–342, 347

WaitHandle class, 204–206, 374

WaitHandle.WaitAll, 202, 231–232, 885

WaitHandle.WaitAny, 885

WaitHandle.WaitOne, 186

WaitHandle.WaitTimeout, 206

Waiting
.NET Framework monitors, 309–310
avoiding deadlocks with, 576
calling `AsyncWaitHandles`' `WaitOne` method, 407–410
causing deadlocks, 575
message waits, 195–198
in native code, 189–195
synchronization via kernel objects with, 184–186
using kernel objects, 188

Waiting, in control synchronization
busy spin waiting, 63–64
continuation passing style vs., 65–66
monitors and condition variables, 68–70
real waiting in OS kernel, 64–65
using events, 66–68

Waiting state, threads, 156

WaitingReadCount, `ReaderWriterLockSlim`, 295

WaitingUpgradeCount, `ReaderWriterLockSlim`, 295

WaitingWriteCount, `ReaderWriterLockSlim`, 295

WAIT_IO_COMPLETION
alertable waits, 193
asynchronous procedure calls and, 209
blocking and pumping messages, 199–201

WAIT_OBJECT_0, 190–191, 199–202

WaitOne method, APM, 407–410, 416

WaitOne method, `WaitHandle` class, 205–206

WaitOrTimerCallback, CLR thread pool, 375

WaitSleepJoin thread state, 158–159, 207–208

WAIT_TIMEOUT, 190–191, 199–201

Wake-all, stampedes, 605–606

Wake-one, stampedes, 605–606

Waking, condition variables and, 306–307, 309

WCF (Windows Communication Foundation), 72–73, 719

WCT (Wait Chain Traversal), Windows Vista, 590, 594–597

Weakening the lock, process shutdown, 563–564

WebClient, 427

WebRequest, APM, 419

WF (Workflow Foundation), 719–720

while loops
data parallelism and, 658–659, 661
iteration and, 672

Win32
bit operations in, 502–503
creating threads in, 90–96
critical sections. *See* Critical sections, Win32
`DllMain` function in, 115–117
interlocked singly-linked lists, 538–540
process shutdown in, 562, 563–568
slim reader/writer locks. *See* SRWLs (slim reader/writer locks)
stack overflow disasters in, 141
terminating threads. *See* Threads, termination methods

thread local storage, 118–121

waiting in, 189–195

Win32 asynchronous I/O, 792
APC callback completion method, 806–808
asynchronous sockets I/O, 814–817
completing, 796
event handler completion method, 802–805
I/O completion ports completion method, 808–813
initiating, 792–796
overview of, 792
polling completion method, 798–800
synchronous completion method, 797–798
wait APIs completion method, 800–802

Win32 legacy thread pool, 353–364

I/O completion ports, 359–360
overview of, 317–319

- Win32 legacy thread pool, *continued*
 performance of, 391–397
 registered waits, 360–363
 thread management, 363–364
 timers, 356–359
 understanding, 353–354
 work items, 354–356
- WinDbg command, 146
- Window procedures, 831
- Windows
 CLR threads vs., 85–87
 GUIs on, 831
 kernel synchronization. *See* Kernel
 synchronization processes, 80–81
 spin waiting, 769–772
 stack overflow disasters in, 141
 threads, 81–85, 152–153
- Windows Communication Foundation (WCF), 72–73, 719
- Windows Forms, 837–840
 identifying calls that need marshalling, 839
 `ISynchronizeInvoke` for marshalling calls, 838–839
 overview of, 837–838
 running message loop mid-stack, 839–840
- Windows Performance Monitor (perfmon.exe), 156–157
- Windows Presentation Foundation (WPF), 840–846
- Windows Task Manager, 175
- Windows Vista
 condition variables, 304–309
 one-time initialization, 529–534
 performance of, 391–397
 process shutdown in, 563–568
 slim reader/writer lock, 288, 289–293
 synchronous I/O cancellation, 823
 Wait Chain Traversal, 590
- Windows Vista thread pool, 323–353
 callback completion tasks, 350–351
 creating timers, 330–334
 debugging, 353
 environments, 342–347
 I/O completion ports, 334–336
 introduction to, 323–324
 no thread ownership and, 352–353
 overview of, 317–319
 registered waits, 336–341
 synchronization with callback completion, 341–342
 thread management, 347–350
 work items, 324–330
- Work callbacks, thread pools and, 319
- Work items
 CLR thread pool, 364–368
 legacy Win32 thread pool, 354–356
 thread pool performance and, 391–397
 Vista thread pool, 324–330
- Work stealing queue, 636–640
- WorkCallback, 456–459, 461
- Workflow Foundation (WF), 719–720
- Workstations (concurrent), garbage collection, 766
- WPF (Windows Presentation Foundation), 840–846
- Write/read hazards, 28
- Write/write hazards, 28
 `_WriteBarrier`, 529
 `WriteFile`, 792

X

- X86 architecture, 509–511, 512
- XADD instruction, 504
- XCHG primitive, 493–499

'When you begin using multi-threading throughout an application, the importance of clean architecture and design is critical.... This places an emphasis on understanding not only the platform's capabilities but also emerging best practices. Joe does a great job interspersing best practices alongside theory throughout his book.'

—From the Foreword by **Craig Mundie**,
Chief Research and Strategy Officer, Microsoft Corporation

Author Joe Duffy has risen to the challenge of explaining how to write software that takes full advantage of concurrency and hardware parallelism. In **Concurrent Programming on Windows**, he explains how to design, implement, and maintain large-scale concurrent programs, primarily using C# and C++ for Windows.

Duffy aims to give application, system, and library developers the tools and techniques needed to write efficient, safe code for multicore processors. This is important not only for the kinds of problems where concurrency is inherent and easily exploitable—such as server applications, compute-intensive image manipulation, financial analysis, simulations, and AI algorithms—but also for problems that can be speeded up using parallelism but require more effort—such as math libraries, sort routines, report generation, XML manipulation, and stream processing algorithms.

Concurrent Programming on Windows has four major sections: The first introduces concurrency at a high level, followed by a section that focuses on the fundamental platform features, inner workings, and API details. Next, there is a section that describes common patterns, best practices, algorithms, and data structures that emerge while writing concurrent software. The final section covers many of the common system-wide architectural and process concerns of concurrent programming.

This is the *only* book you'll need in order to learn the best practices and common patterns for programming with concurrency on Windows and .NET.

Joe Duffy is the development lead, architect, and founder of the Parallel Extensions to the .NET Framework team at Microsoft. In addition to hacking code and managing a team of developers, he works on long-term vision and incubation efforts, such as language and type system support for concurrency safety. He previously worked on the Common Language Runtime team. Joe blogs regularly at www.bluebytesoftware.com/blog.

informati.com/msdotnetseries

Cover photograph by Jorg Greuel/ GettyImages Inc.

 Text printed on recycled paper

 Addison-Wesley
Pearson Education

Microsoft® **.NET** Development Series

"Supported by the leaders and principal authorities of core Microsoft technologies, this series has an author pool that combines some of the most insightful authors in the industry with the lead software architects and developers at Microsoft and the developer community at large

—**Don Box**
Architect, Microsoft

"This is a great resource for professional .NET developers. It covers all bases, from expert perspective to reference and how-to. Books in this series are essential reading for those who want to judiciously expand their knowledge base and expertise."

—**John Montgomery**
Principal Group Program Manager,
Developer Division, Microsoft

"This foremost series on .NET contains vital information for developers who need to get the most out of the .NET Framework. Our authors are selected from the key innovators who create the technology and are the most respected practitioners of it."

—**Brad Abrams**
Group Program Manager, Microsoft

ISBN-13: 978-0-321-43482-1
ISBN-10: 0-321-43482-X



9 780321 434821

FREE Online Edition
with purchase of this book.

 Details on Last Page

\$49.99 U.S. | \$54.99 CANADA