





1111

WINDOWS VIA C/C++

CALIBRA TO ZOODO FEET ALT

Jeffrey Richter Christophe Nasarre



PUBLISHED BY Microsoft Press A Division of Microsoft Corporation One Microsoft Way Redmond, Washington 98052-6399

Copyright © 2008 by Jeffrey Richter and Christophe Nasarre

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2007939306

ISBN: 978-0-7356-6377-0

 $1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ QGT\ 6\ 5\ 4\ 3\ 2\ 1$

Printed and bound in the United States of America.

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, ActiveX, Developer Studio, Intellisense, Internet Explorer, Microsoft Press, MSDN, MS-DOS, PowerPoint, SQL Server, SuperFetch, Visual Basic, Visual C++, Visual Studio, Win32, Win32s, Windows, Windows Media, Windows NT, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. cpossible third-party trademark info>. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Ben Ryan Developmental and Project Editor: Lynn Finnel Editorial Production: Publishing.com Technical Reviewer: Scott Seely; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Body Part No. X14-25709

Dedication

To Kristin, words cannot express how I feel about our life together. I cherish our family and all our adventures. I'm filled each day with love for you.

To Aidan, you have been an inspiration to me and have taught me to play and have fun. Watching you grow up has been so rewarding and enjoyable for me. I feel lucky to be able to partake in your life; it has made me a better person.

To My New Baby Boy (shipping Q1 2008), you have been wanted for so long it's hard to believe that you're almost here. You bring completeness and balance to our family. I look forward to playing with you, learning who you are, and enjoying our time together.

- Jeffrey Richter

To my wife Florence, au moins cette fois c'est écrit: je t'aime Flo.

To my parents who cannot believe that learning English with Dungeons & Dragons rules could have been so efficient.

- Christophe Nasarre

Contents at a Glance

Part I	Required Reading
1	Error Handling
2	Working with Characters and Strings 11
3	Kernel Objects
Part II	Getting Work Done
4	Processes
5	Jobs
6	Thread Basics
7	Thread Scheduling, Priorities, and Affinities
8	Thread Synchronization in User Mode
9	Thread Synchronization with Kernel Objects
10	Synchronous and Asynchronous Device I/O
11	The Windows Thread Pool
12	Fibers
Part III	Memory Management
13	Windows Memory Architecture
14	Exploring Virtual Memory
15	Using Virtual Memory in Your Own Applications
16	A Thread's Stack
17	Memory-Mapped Files
18	Heaps
Part IV	Dynamic-Link Libraries
19	DLL Basics
20	DLL Advanced Techniques
21	Thread-Local Storage
22	DLL Injection and API Hooking

Part V	Structured Exception Handling
23	Termination Handlers
24	Exception Handlers and Software Exceptions
25	Unhandled Exceptions, Vectored Exception Handling, and C++ Exceptions
26	Error Reporting and Application Recovery
Part VI	Appendixes
Α	The Build Environment
В	Message Crackers, Child Control Macros, and API Macros

Table of Contents

Acknowledgments	xxi
Introduction	<i>xxiii</i>
64-Bit Windows	xxiii
What's New in the Fifth Edition	xxiv
Code Samples and System Requirements	xxvi
Support for This Book	xxvi
Questions and Comments	xxvi
Required Reading	
Error Handling	3
Defining Your Own Error Codes	7
The ErrorShow Sample Application	7
Working with Characters and Strings	11
Character Encodings	12
ANSI and Unicode Character and String Data Types	
Unicode and ANSI Functions in Windows	
Unicode and ANSI Functions in the C Run-Time Library	
Secure String Functions in the C Run-Time Library	
Introducing the New Secure String Functions	19
How to Get More Control When Performing String Operations	22
Windows String Functions	24
Why You Should Use Unicode	
How We Recommend Working with Characters and Strings	26
Translating Strings Between Unicode and ANSI	27
Exporting ANSI and Unicode DLL Functions	29
	Acknowledgments Introduction 64-Bit Windows What's New in the Fifth Edition Code Samples and System Requirements Support for This Book Questions and Comments Questions and Comments Perfured Reading Error Handling Defining Your Own Error Codes The ErrorShow Sample Application Working with Characters and Strings Character Encodings ANSI and Unicode Character and String Data Types Unicode and ANSI Functions in Windows Unicode and ANSI Functions in the C Run-Time Library Secure String Functions in the C Run-Time Library Introducing the New Secure String Functions How to Get More Control When Performing String Operations Windows String Functions Why You Should Use Unicode How We Recommend Working with Characters and Strings Translating Strings Between Unicode and ANSI Exporting ANSI and Unicode DLL Functions.

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

	Determining If Text Is ANSI or Unicode
3	Kernel Objects
	What Is a Kernel Object?
	Usage Counting
	Security
	A Process' Kernel Object Handle Table
	Creating a Kernel Object 38
	Closing a Kernel Object 39
	Sharing Kernel Objects Across Process Boundaries
	Using Object Handle Inheritance
	Naming Objects
	Duplicating Object Handles 60

Part II Getting Work Done

Processes	7
Writing Your First Windows Application6	8
A Process Instance Handle7	3
The CreateProcess Function	9
pszApplicationName and pszCommandLine	9
Terminating a Process	4
The Primary Thread's Entry-Point Function Returns	4
The <i>ExitProcess</i> Function	5
The TerminateProcess Function	6
When All the Threads in the Process Die	7
When a Process Terminates10	7
Child Processes	8
Running Detached Child Processes 11	0
When Administrator Runs as a Standard User	0
Elevating a Process Automatically 11	3
Elevating a Process by Hand 11	5
What Is the Current Privileges Context?	7
Enumerating the Processes Running in the System	8
	Processes 6 Writing Your First Windows Application. 6 A Process Instance Handle. 7 The CreateProcess Function 8 pszApplicationName and pszCommandLine. 8 Terminating a Process 10 The ExitProcess Function 10 The ExitProcess Function 10 The TerminateProcess Function 10 When All the Thread's In the Process Die 10 When a Process Terminates. 10 Child Processes . 10 Running Detached Child Processes 11 When Administrator Runs as a Standard User 11 Elevating a Process by Hand. 11 What Is the Current Privileges Context? 11 Enumerating the Processes Running in the System 11

5	Jobs	125
	Placing Restrictions on a Job's Processes.	129
	Placing a Process in a Job	136
	Terminating All Processes in a Job	
	Querying Job Statistics	
	Job Notifications	140
	The Job Lab Sample Application.	143
6	Thread Basics	145
	When to Create a Thread	146
	When Not to Create a Thread	148
	Writing Your First Thread Function	149
	The CreateThread Function	150
	psa	151
	cbStackSize	151
	pfnStartAddr and pvParam	152
	dwCreateFlags	153
	pdwThreadID	153
	Terminating a Thread	153
	The Thread Function Returns	154
	The <i>ExitThread</i> Function	154
	The TerminateThread Function	154
	When a Process Terminates	155
	When a Thread Terminates	155
	Some Thread Internals	156
	C/C++ Run-Time Library Considerations	159
	Oops—I Called CreateThread Instead of _beginthreadex by Mistake	
	C/C++ Run-Time Library Functions That You Should Never Call	
	Gaining a Sense of One's Own Identity	
	Converting a Pseudohandle to a Real Handle	170

Table of Contents

7	Thread Scheduling, Priorities, and Affinities	
	Suspending and Resuming a Thread	
	Suspending and Resuming a Process	
	Sleeping	
	Switching to Another Thread	
	Switching to Another Thread on a Hyper-Threaded CPU	
	A Thread's Execution Times	
	Putting the CONTEXT in Context.	
	Thread Priorities	
	An Abstract View of Priorities	
	Programming Priorities	
	Dynamically Boosting Thread Priority Levels	
	Tweaking the Scheduler for the Foreground Process	
	Scheduling I/O Request Priorities	
	The Scheduling Lab Sample Application	
	Affinities	203
8	Thread Synchronization in User Mode	207
	Atomic Access: The Interlocked Family of Functions	
	Cache Lines	
	Advanced Thread Synchronization	
	A Technique to Avoid	
	Critical Sections	
	Critical Sections: The Fine Print.	
	Critical Sections and Spinlocks	
	Critical Sections and Error Handling	
	Slim Reader-Writer Locks	
	Condition Variables	
	The Queue Sample Application	
	Useful Tips and Techniques	

9	Thread Synchronization with Kernel Objects	241
	Wait Functions	243
	Successful Wait Side Effects	246
	Event Kernel Objects	247
	The Handshake Sample Application	252
	Waitable Timer Kernel Objects	256
	Having Waitable Timers Queue APC Entries	260
	Timer Loose Ends	261
	Semaphore Kernel Objects.	262
	Mutex Kernel Objects	265
	Abandonment Issues.	267
	Mutexes vs. Critical Sections	267
	The Queue Sample Application.	268
	A Handy Thread Synchronization Object Chart	276
	Other Thread Synchronization Functions	277
	Asynchronous Device I/O	277
	WaitForInputIdle	278
	MsgWaitForMultipleObjects(Ex)	278
	WaitForDebugEvent	279
	SignalObjectAndWait.	279
	Detecting Deadlocks with the Wait Chain Traversal API	
10	Synchronous and Asynchronous Device I/O	289
	Opening and Closing Devices	290
	A Detailed Look at <i>CreateFile</i>	292
	Working with File Devices	299
	Getting a File's Size	299
	Positioning a File Pointer	300
	Setting the End of a File	302
	Performing Synchronous Device I/O	302
	Flushing Data to the Device	303
	Synchronous I/O Cancellation	303

	Basics of Asynchronous Device I/O 305
	The OVERLAPPED Structure
	Asynchronous Device I/O Caveats
	Canceling Queued Device I/O Requests
	Receiving Completed I/O Request Notifications
	Signaling a Device Kernel Object
	Signaling an Event Kernel Object
	Alertable I/O
	I/O Completion Ports 320
11	The Windows Thread Pool
	Scenario 1: Call a Function Asynchronously
	Explicitly Controlling a Work Item
	The Batch Sample Application 342
	Scenario 2: Call a Function at a Timed Interval
	The Timed Message Box Sample Application
	Scenario 3: Call a Function When a Single Kernel Object Becomes Signaled 351
	Scenario 4: Call a Function When Asynchronous I/O Requests Complete 353
	Callback Termination Actions 355
	Customized Thread Pools 356
	Gracefully Destroying a Thread Pool: Cleanup Groups
12	Fibers
	Working with Fibers
	The Counter Sample Application

Part III Memory Management

13	Windows Memory Architecture 371
	A Process' Virtual Address Space
	How a Virtual Address Space Is Partitioned
	Null-Pointer Assignment Partition
	User-Mode Partition
	Kernel-Mode Partition

	Regions in an Address Space	
	Committing Physical Storage Within a Region	
	Physical Storage and the Paging File	
	Physical Storage Not Maintained in the Paging File	
	Protection Attributes	
	Copy-on-Write Access	
	Special Access Protection Attribute Flags	
	Bringing It All Home	
	Inside the Regions	
	The Importance of Data Alignment	
14	Exploring Virtual Memory	395
	System Information	
	The System Information Sample Application	
	Virtual Memory Status	
	Memory Management on NUMA Machines	405
	The Virtual Memory Status Sample Application.	
	Determining the State of an Address Space	408
	The <i>VMQuery</i> Function	
	The Virtual Memory Map Sample Application	
15	Using Virtual Memory in Your Own Applications	419
	Reserving a Region in an Address Space	
	Committing Storage in a Reserved Region	
	Reserving a Region and Committing Storage Simultaneously	422
	When to Commit Physical Storage	
	Decommitting Physical Storage and Releasing a Region	
	When to Decommit Physical Storage	426
	The Virtual Memory Allocation Sample Application	427
	Changing Protection Attributes	434
	Resetting the Contents of Physical Storage	435
	The MemReset Sample Application	437
	Address Windowing Extensions	439
	The AWE Sample Application	442

Table of Contents

16	A Thread's Stack 451
	The C/C++ Run-Time Library's Stack-Checking Function
	The Summation Sample Application
17	Memory-Mapped Files 463
	Memory-Mapped Executables and DLLs
	Static Data Is Not Shared by Multiple Instances of an Executable
	or a DLL
	Memory-Mapped Data Files
	Method 1: One File, One Buffer 476
	Method 2: Two Files, One Buffer 476
	Method 3: One File, Two Buffers
	Method 4: One File, Zero Buffers
	Using Memory-Mapped Files
	Step 1: Creating or Opening a File Kernel Object
	Step 2: Creating a File-Mapping Kernel Object
	Step 3: Mapping the File's Data into the Process' Address Space
	Step 4: Unmapping the File's Data from the Process' Address Space 485
	Steps 5 and 6: Closing the File-Mapping Object and the File Object 486
	The File Reverse Sample Application
	Processing a Big File Using Memory-Mapped Files
	Memory-Mapped Files and Coherence
	Specifying the Base Address of a Memory-Mapped File
	Implementation Details of Memory-Mapped Files
	Using Memory-Mapped Files to Share Data Among Processes
	Memory-Mapped Files Backed by the Paging File
	The Memory-Mapped File Sharing Sample Application
	Sparsely Committed Memory-Mapped Files
	The Sparse Memory-Mapped File Sample Application

18	Heaps
	A Process' Default Heap519
	Reasons to Create Additional Heaps520
	Component Protection
	More Efficient Memory Management
	Local Access
	Avoiding Thread Synchronization Overhead
	Quick Free
	How to Create an Additional Heap523
	Allocating a Block of Memory from a Heap
	Changing the Size of a Block526
	Obtaining the Size of a Block527
	Freeing a Block527
	Destroying a Heap528
	Using Heaps with C++
	Miscellaneous Heap Functions531

Part IV Dynamic-Link Libraries

19	DLL Basics	537
	DLLs and a Process' Address Space	
	The Overall Picture	540
	Building the DLL Module	542
	Building the Executable Module	547
	Running the Executable Module	550
20	DLL Advanced Techniques	553
20	DLL Advanced Techniques. Explicit DLL Module Loading and Symbol Linking.	553
20	DLL Advanced Techniques. Explicit DLL Module Loading and Symbol Linking. Explicitly Loading the DLL Module.	
20	DLL Advanced Techniques. Explicit DLL Module Loading and Symbol Linking. Explicitly Loading the DLL Module. Explicitly Unloading the DLL Module.	
20	DLL Advanced Techniques. Explicit DLL Module Loading and Symbol Linking. Explicitly Loading the DLL Module. Explicitly Unloading the DLL Module. Explicitly Linking to an Exported Symbol .	
20	DLL Advanced Techniques. Explicit DLL Module Loading and Symbol Linking. Explicitly Loading the DLL Module. Explicitly Unloading the DLL Module. Explicitly Linking to an Exported Symbol The DLL's Entry-Point Function.	

566
567
567
570
571
576
583
584
585
586
592
. 597
598
600
600 602
600 602
600 602 605
600 602 .605 605 608
600 602 605 605 608 609
600 602 605 608 609 610
600 602 605 605 608 609 610 621
600 602 605 605 608 609 610 621 625
600 602 605 605 608 609 610 621 625 631
600 602 605 605 608 609 610 621 625 631 633
600 602 605 608 609 610 621 625 631 633 633
600 602 605 605 608 609 610 621 625 633 633 633
600 602 605 605 608 609 610 621 625 631 633 633 633 634
600 602 605 608 609 610 621 625 631 633 633 634 635
600 602 605 605 608 609 610 621 625 633 633 633 634 635
600 602 605 605 608 609 610 621 625 631 633 633 633 634 635 636

Part V Structured Exception Handling

23	Termination Handlers	659
	Understanding Termination Handlers by Example	
	Funcenstein1	660
	Funcenstein2	661
	Funcenstein3	
	Funcfurter1	
	Pop Quiz Time: <i>FuncaDoodleDoo</i>	
	Funcenstein4	
	Funcarama1	667
	Funcarama2	
	Funcarama3	
	<i>Funcarama4</i> : The Final Frontier	
	Notes About the <i>finally</i> Block	671
	Funcfurter2	672
	The SEH Termination Sample Application	673
24	Exception Handlers and Software Exceptions	679
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers	679
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds EXCEPTION_CONTINUE_EXECUTION	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds EXCEPTION_CONTINUE_EXECUTION Use EXCEPTION_CONTINUE_EXECUTION	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds EXCEPTION_CONTINUE_EXECUTION Use EXCEPTION_CONTINUE_EXECUTION EXCEPTION_CONTINUE_SEARCH	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds Use EXCEPTION_CONTINUE_EXECUTION Use EXCEPTION_CONTINUE_EXECUTION EXCEPTION_CONTINUE_SEARCH GetExceptionCode	
24	Exception Handlers and Software Exceptions Understanding Exception Filters and Exception Handlers by Example Funcmeister1 Funcmeister2 EXCEPTION_EXECUTE_HANDLER Some Useful Examples Global Unwinds Halting Global Unwinds EXCEPTION_CONTINUE_EXECUTION Use EXCEPTION_CONTINUE_EXECUTION EXCEPTION_CONTINUE_SEARCH GetExceptionCode Memory-Related Exceptions	

	Debugging-Related Exceptions	696
	Integer-Related Exceptions	696
	Floating Point-Related Exceptions	696
	GetExceptionInformation	699
	Software Exceptions	702
25	Unhandled Exceptions, Vectored Exception Handling, and C++ Exceptions	705
	Inside the UnhandledExceptionFilter Function	707
	Action #1: Allowing Write Access to a Resource and	
	Continuing Execution	
	Action #2: Notifying a Debugger of the Unhandled	700
	Exception	
	Action #3: Notifying Your Globally Set Filter Function	
	Exception (Again)	708
	Action #5: Silently Terminating the Process	
	UnhandledExceptionFilter and WER Interactions	710
	Just-in-Time Debugging	713
	The Spreadsheet Sample Application	716
	Vectored Exception and Continue Handlers	726
	C++ Exceptions vs. Structured Exceptions	727
	Exceptions and the Debugger	729
26	Error Reporting and Application Recovery	733
	The Windows Error Reporting Console	733
	Programmatic Windows Error Reporting	736
	Disabling Report Generation and Sending	737
	Customizing All Problem Reports Within a Process	738
	Creating and Customizing a Problem Report	740
	Creating a Custom Problem Report: <i>WerReportCreate</i>	742
	Setting Report Parameters: WerReportSetParameter	743

Adding a Minidump File to the Report: <i>WerReportAddDump</i>	744
Adding Arbitrary Files to the Report: <i>WerReportAddFile</i>	745
Modifying Dialog Box Strings: WerReportSetUlOption	746
Submitting a Problem Report: <i>WerReportSubmit</i>	746
Closing a Problem Report: WerReportCloseHandle	748
The Customized WER Sample Application	748
Automatic Application Restart and Recovery	754
Automatic Application Restart	755
Support for Application Recovery	756

Part VI Appendixes

Α	The Build Environment
	The CmnHdr.h Header File761
	Microsoft Windows Version Build Option
	Unicode Build Option762
	Windows Definitions and Warning Level 4
	The <i>pragma message</i> Helper Macro
	The <i>chINRANGE</i> Macro763
	The <i>chBEGINTHREADEX</i> Macro763
	DebugBreak Improvement for x86 Platforms
	Creating Software Exception Codes
	The <i>chMB</i> Macro
	The <i>chASSERT</i> and <i>chVERIFY</i> Macros
	The <i>chHANDLE_DLGMSG</i> Macro766
	The chSETDLGICONS Macro766
	Forcing the Linker to Look for a (<i>w</i>) <i>WinMain</i> Entry-Point Function766
	Support XP-Theming of the User Interface with <i>pragma</i>

В	Message Crackers, Child Control Macros, and API Macros
	Message Crackers
	Child Control Macros
	API Macros
	Index

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Acknowledgments

We could not have written this book without the help and technical assistance of several people. In particular, we'd like to thank the following people:

Jeffrey's Family

Jeffrey would like to thank Kristin (his wife) and Aidan (his son) for their never ending love and support.

Christophe's Family

Christophe would not have been able to write the fifth edition of this book without the love and support of Florence (his wife), the never ending curiosity of Celia (his daughter), and the purring sneak attacks of Canelle and Nougat (his cats). Now, I don't have any good excuse to not take care of you!

Technical Assistance

For writing a book like this one, personal research is not enough. We were owe great thanks to various Microsoft employees who helped us. Specifically, we'd like to thank Arun Kishan, who was able to either instantly answer weird and complicated questions or find the right person on the Windows team to provide more detailed explanations. We would also like to thank Kinshuman Kinshumann, Stephan Doll, Wedson Almeida Filho, Eric Li, Jean-Yves Poublan, Sandeep Ranade, Alan Chan, Ale Contenti, Kang Su Gatlin, Kai Hsu, Mehmet Iyigun, Ken Jung, Pavel Lebedynskiy, Paul Sliwowicz, and Landy Wang. In addition, there are those who listened to questions posted on Microsoft internal forums and shared their extensive knowledge, such as Raymond Chen, Sunggook Chue, Chris Corio, Larry Osterman, Richard Russell, Mark Russinovich, Mike Sheldon, Damien Watkins, and Junfeng Zhang. Last but not least, we would like to warmly thank John "Bugslayer" Robbins and Kenny Kerr who were kind enough to provide great feedback on chapters of this book.

Microsoft Press Editorial Team

We would like to thank Ben Ryan (acquisitions editor) for trusting a crazy French guy like Christophe, managers Lynn Finnel and Curtis Philips for their patience, Scott Seely for his constant search for technical accuracy, Roger LeBlanc for his talent in transforming Christophe's French-like English into something understandable, and Andrea Fox for her meticulous proofreading. In addition to the Redmond team, Joyanta Sen spent a lot of his personal time supporting us.

Mutual Admiration

Christophe sincerely thanks Jeffrey Richter for trusting him not to spoil the fifth edition of Jeff's book.

Jeffrey also thanks Christophe for his tireless efforts in researching, reorganizing, rewriting, and reworking the content in an attempt to reach Jeff's idea of perfection.

Introduction

Microsoft Windows is a complex operating system. It offers so many features and does so much that it's impossible for any one person to fully understand the entire system. This complexity also makes it difficult for someone to decide where to start concentrating the learning effort. Well, I always like to start at the lowest level by gaining a solid understanding of the system's basic building blocks. Once you understand the basics, it's easy to incrementally add any higher-level aspects of the system to your knowledge. So this book focuses on Windows' basic building blocks and the fundamental concepts that you must know when architecting and implementing software targeting the Windows operating system. In short, this book teaches the reader about various Windows features and how to access them via the *C* and C++ programming languages.

Although this book does not cover some Windows concepts—such as the Component Object Model (COM)—COM is built on top of basic building blocks such as processes, threads, memory management, DLLs, thread local storage, Unicode, and so on. If you know these basic building blocks, understanding COM is just a matter of understanding how the building blocks are used. I have great sympathy for people who attempt to jump ahead in learning COM's architecture. They have a long road ahead and are bound to have gaping holes in their knowledge, which is bound to negatively affect their code and their software development schedules.

The Microsoft .NET Framework's common language runtime (CLR) is another technology not specifically addressed in this book. (However, it is addressed in my other book: *CLR via C#*, Jeffrey Richter, Microsoft Press, 2006). However, the CLR is implemented as a COM object in a dynamiclink library (DLL) that loads in a process and uses threads to execute code that manipulates Unicode strings that are managed in memory. So again, the basic building blocks presented in this book will help developers writing managed code. In addition, by way of the CLR's Platform Invocation (P/Invoke) technology, you can call into the various Windows' APIs presented throughout this book.

So that's what this book is all about: the basic Windows building blocks that every Windows developer (at least in my opinion) should be intimately aware of. As each block is discussed, I also describe how the system uses these blocks and how your own applications can best take advantage of these blocks. In many chapters, I show you how to create building blocks of your own. These building blocks, typically implemented as generic functions or C++ classes, group a set of Windows building blocks together to create a whole that is much greater than the sum of its parts.

64-Bit Windows

Microsoft has been shipping 32-bit versions of Windows that support the x86 CPU architecture for many years. Today, Microsoft also offers 64-bit versions of Windows that support the x64 and IA-64 CPU architectures. Machines based on these 64-bit CPU architectures are fast gaining acceptance. In fact, in the very near future, it is expected that all desktop and server machines will contain 64-bit CPUs. Because of this, Microsoft has stated that Windows Server 2008 will be the last 32-bit version of Windows ever! For developers, now is the time to focus on making sure your applications run correctly on 64-bit Windows. To this end, this book includes solid coverage of what you need to know to have your applications run on 64-bit Windows (as well as 32-bit Windows).

xxiv Introduction

The biggest advantage your application gets from a 64-bit address space is the ability to easily manipulate large amounts of data, because your process is no longer constrained to a 2-GB usable address space. Even if your application doesn't need all this address space, Windows itself takes advantage of the significantly larger address space (about 8 terabytes), allowing it to run faster.

Here is a quick look at what you need to know about 64-bit Windows:

- The 64-bit Windows kernel is a port of the 32-bit Windows kernel. This means that all the details and intricacies that you've learned about 32-bit Windows still apply in the 64-bit world. In fact, Microsoft has modified the 32-bit Windows source code so that it can be compiled to produce a 32-bit or a 64-bit system. They have just one source-code base, so new features and bug fixes are simultaneously applied to both systems.
- Because the kernels use the same code and underlying concepts, the Windows API is identical on both platforms. This means that you do not have to redesign or reimplement your application to work on 64-bit Windows. You can simply make slight modifications to your source code and then rebuild.
- For backward compatibility, 64-bit Windows can execute 32-bit applications. However, your application's performance will improve if the application is built as a true 64-bit application.
- Because it is so easy to port 32-bit code, there are already device drivers, tools, and applications available for 64-bit Windows. Unfortunately, Visual Studio is a native 32-bit application and Microsoft seems to be in no hurry to port it to be a native 64-bit application. However, the good news is that 32-bit Visual Studio does run quite well on 64-bit Windows; it just has a limited address space for its own data structures. And Visual Studio does allow you to debug a 64-bit application.
- There is little new for you to learn. You'll be happy to know that most data types remain 32 bits wide. These include **int**s, **DWORD**s, **LONG**s, **BOOL**s, and so on. In fact, you mostly just need to worry about pointers and handles, since they are now 64-bit values.

Because Microsoft offers so much information on how to modify your existing source code to be 64-bit ready, I will not go into those details in this book. However, I thought about 64-bit Windows as I wrote each chapter. Where appropriate, I have included information specific to 64-bit Windows. In addition, I have compiled and tested all the sample applications in this book for 64-bit Windows. So, if you follow the sample applications in this book and do as I've done, you should have no trouble creating a single source-code base that you can easily compile for 32-bit or 64-bit Windows.

What's New in the Fifth Edition

In the past, this book has been titled *Advanced Windows NT*, *Advanced Windows*, and *Programming Applications for Microsoft Windows*. In keeping with tradition, this edition of the book has gotten a new title: *Windows via C/C++*. This new title indicates that the book is for C and C++ programmers wanting to understand Windows. This new edition covers more than 170 new functions and Windows features that have been introduced in Windows XP, Windows Vista, and Windows Server 2008.

Some chapters have been completely rewritten—such as Chapter 11, which explains how the new thread pool API should be used. Existing chapters have been greatly enhanced to present new features. For example, Chapter 4 now includes coverage of User Account Control and Chapter 8 now covers new synchronization mechanisms (Interlocked Singly-Linked List, Slim Reader-Writer Locks, and condition variables).

I also give much more coverage of how the C/C++ run-time library interacts with the operating system—particularly on enhancing security as well as exception handling. Last but not least, two new chapters have been added to explain how I/O operations work and to dig into the new Windows Error Reporting system that changes the way you must think about application error reporting and application recovery.

In addition to the new organization and greater depth, I added a ton of new content. Here is a partial list of enhancements made for this edition:

- **New Windows Vista and Windows Server 2008 features** Of course, the book would not be a true revision unless it covered new features offered in Windows XP, Windows Vista, Windows Server 2008, and the C/C++ run-time library. This edition has new information on the secure string functions, the kernel object changes (such as namespaces and boundary descriptors), thread and process attribute lists, thread and I/O priority scheduling, synchronous I/O cancellation, vectored exception handling, and more.
- **64-bit Windows support** The text addresses 64-bit Windows-specific issues; all sample applications have been built and tested on 64-bit Windows.
- **Use of C++** The sample applications use C++ and require fewer lines of code, and their logic is easier to follow and understand.
- **Reusable code** Whenever possible, I created the source code to be generic and reusable. This should allow you to take individual functions or entire C++ classes and drop them into your own applications with little or no modification. The use of C++ made reusability much easier.
- **The ProcessInfo utility** This particular sample application from the earlier editions has been enhanced to show the process owner, command line, and UAC-related details.
- **The LockCop utility** This sample application is new. It shows which processes are running on the system. Once you select a process, this utility lists the threads of the process and, for each, on which kind of synchronization mechanism it is blocked—with deadlocks explicitly pointed out.
- **API hooking** I present updated C++ classes that make it trivial to hook APIs in one or all modules of a process. My code even traps run-time calls to **LoadLibrary** and **GetProcAddress** so that your API hooks are enforced.
- **Structured exception handling improvements** I have rewritten and reorganized much of the structured exception handling material. I have more information on unhandled exceptions, and I've added coverage on customizing Windows Error Reporting to fulfill your needs.

Code Samples and System Requirements

The sample applications presented throughout this book can be downloaded from the book's companion content Web page at

http://www.Wintellect.com/Books.aspx

To build the applications, you'll need Visual Studio 2005 (or later), the Microsoft Platform SDK for Windows Vista and Windows Server 2008 (which comes with some versions of Visual Studio). In addition, to run the applications, you'll need a computer (or virtual machine) with Windows Vista (or later) installed.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. As corrections or changes are collected, they will be added to an Errata document downloadable at the following Web site:

http://www.Wintellect.com/Books.aspx

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the site just mentioned, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press Attn: *Windows via C/C++* Editor One Microsoft Way Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 2 Working with Characters and Strings

In this chapter:
Character Encodings
ANSI and Unicode Character and String Data Types
Unicode and ANSI Functions in Windows15
Unicode and ANSI Functions in the C Run-Time Library
Secure String Functions in the C Run-Time Library
Why You Should Use Unicode
How We Recommend Working with Characters and Strings
Translating Strings Between Unicode and ANSI

With Microsoft Windows becoming more and more popular around the world, it is increasingly important that we, as developers, target the various international markets. It was once common for U.S. versions of software to ship as much as six months prior to the shipping of international versions. But increasing international support for the operating system is making it easier to produce applications for international markets and therefore is reducing the time lag between distribution of the U.S. and international versions of our software.

Windows has always offered support to help developers localize their applications. An application can get country-specific information from various functions and can examine Control Panel settings to determine the user's preferences. Windows even supports different fonts for our applications. Last but not least, in Windows Vista, Unicode 5.0 is now supported. (Read "Extend The Global Reach Of Your Applications With Unicode 5.0" at *http://msdn.microsoft.com/msdnmag/ issues/07/01/Unicode/default.aspx* for a high-level presentation of Unicode 5.0.)

Buffer overrun errors (which are typical when manipulating character strings) have become a vector for security attacks against applications and even against parts of the operating system. In previous years, Microsoft put forth a lot of internal and external efforts to raise the security bar in the Windows world. The second part of this chapter presents new functions provided by Microsoft in the C run-time library. You should use these new functions to protect your code against buffer overruns when manipulating strings.

I decided to present this chapter early in the book because I highly recommend that your application always use Unicode strings and that you always manipulate these strings via the new secure string functions. As you'll see, issues regarding the secure use of Unicode strings are discussed in just about every chapter and in all the sample applications presented in this book. If you have a code base that is non-Unicode, you'll be best served by moving that code base to Unicode, as this will improve your application's execution performance as well as prepare it for localization. It will also help when interoperating with COM and the .NET Framework.

Character Encodings

The real problem with localization has always been manipulating different character sets. For years, most of us have been coding text strings as a series of single-byte characters with a zero at the end. This is second nature to us. When we call **strlen**, it returns the number of characters in a zero-terminated array of ANSI single-byte characters.

The problem is that some languages and writing systems (Japanese kanji being a classic example) have so many symbols in their character sets that a single byte, which offers no more than 256 different symbols at best, is just not enough. So double-byte character sets (DBCSs) were created to support these languages and writing systems. In a double-byte character set, each character in a string consists of either 1 or 2 bytes. With kanji, for example, if the first character is between 0x81 and 0x9F or between 0xE0 and 0xFC, you must look at the next byte to determine the full character in the string. Working with double-byte character sets is a programmer's nightmare because some characters are 1 byte wide and some are 2 bytes wide. Fortunately, you can forget about DBCS and take advantage of the support of Unicode strings supported by Windows functions and the C run-time library functions.

Unicode is a standard founded by Apple and Xerox in 1988. In 1991, a consortium was created to develop and promote Unicode. The consortium consists of companies such as Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys, and Xerox. (A complete and updated list of consortium members is available at *http://www.Unicode.org.*) This group of companies is responsible for maintaining the Unicode standard. The full description of Unicode can be found in *The Unicode Standard*, published by Addison-Wesley. (This book is available through *http://www.Unicode.org.*)

In Windows Vista, each Unicode character is encoded using UTF-16 (where *UTF* is an acronym for *Unicode Transformation Format*). UTF-16 encodes each character as 2 bytes (or 16 bits). In this book, when we talk about Unicode, we are always referring to UTF-16 encoding unless we state otherwise. Windows uses UTF-16 because characters from most languages used throughout the world can easily be represented via a 16-bit value, allowing programs to easily traverse a string and calculate its length. However, 16-bits is not enough to represent all characters from certain languages. For these languages, UTF-16 supports surrogates, which are a way of using 32 bits (or 4 bytes) to represent a single character. Because few applications need to represent the characters of these languages, UTF-16 is a good compromise between saving space and providing ease of coding. Note that the .NET Framework always encodes all characters and strings using UTF-16, so using UTF-16 in your Windows application will improve performance and reduce memory consumption if you need to pass characters or strings between native and managed code.

There are other UTF standards for representing characters, including the following ones:

UTF-8 UTF-8 encodes some characters as 1 byte, some characters as 2 bytes, some characters as 3 bytes, and some characters as 4 bytes. Characters with a value below 0x0080 are compressed to 1 byte, which works very well for characters used in the United States. Characters between 0x0080 and 0x07FF are converted to 2 bytes, which works well for European and Middle Eastern languages. Characters of 0x0800 and above are converted to 3 bytes, which works well for East Asian languages. Finally, surrogate pairs are written out as 4 bytes. UTF-8 is an extremely popular encoding format, but it's less efficient than UTF-16 if you encode many characters with values of 0x0800 or above.

UTF-32 UTF-32 encodes every character as 4 bytes. This encoding is useful when you want to write a simple algorithm to traverse characters (used in any language) and you don't want to have to deal with characters taking a variable number of bytes. For example, with UTF-32, you do not need to think about surrogates because every character is 4 bytes. Obviously, UTF-32 is not an efficient encoding format in terms of memory usage. Therefore, it's rarely used for saving or transmitting strings to a file or network. This encoding format is typically used inside the program itself.

Currently, Unicode code points¹ are defined for the Arabic, Chinese bopomofo, Cyrillic (Russian), Greek, Hebrew, Japanese kana, Korean hangul, and Latin (English) alphabets—called scripts—and more. Each version of Unicode brings new characters in existing scripts and even new scripts such as Phoenician (an ancient Mediterranean alphabet). A large number of punctuation marks, mathematical symbols, technical symbols, arrows, dingbats, diacritics, and other characters are also included in the character sets.

These 65,536 characters are divided into regions. Table 2-1 shows some of the regions and the characters that are assigned to them.

16-Bit Code	Characters	16-Bit Code	Alphabet/Scripts
0000-007F	ASCII	0300–036F	Generic diacritical marks
0080-00FF	Latin1 characters	0400-04FF	Cyrillic
0100-017F	European Latin	0530–058F	Armenian
0180-01FF	Extended Latin	0590–05FF	Hebrew
0250-02AF	Standard phonetic	0600–06FF	Arabic
02B0-02FF	Modified letters	0900–097F	Devanagari

Table 2-1 Unicode Character Sets and Alphabets

ANSI and Unicode Character and String Data Types

I'm sure you're aware that the C language uses the **char** data type to represent an 8-bit ANSI character. By default, when you declare a literal string in your source code, the C compiler turns the string's characters into an array of 8-bit **char** data types:

```
// An 8-bit character
char c = 'A';
```

```
// An array of 99 8-bit characters and an 8-bit terminating zero.
char szBuffer[100] = "A String";
```

Microsoft's *C/C*++ compiler defines a built-in data type, **wchar_t**, which represents a 16-bit Unicode (UTF-16) character. Because earlier versions of Microsoft's compiler did not offer this built-in data type, the compiler defines this data type only when the **/Zc:wchar_t** compiler switch is specified. By default, when you create a C++ project in Microsoft Visual Studio, this compiler switch is specified. We recommend that you always specify this compiler switch, as it is better to work with Unicode characters by way of the built-in primitive type understood intrinsically by the compiler.

¹ A code point is the position of a symbol in a character set.



Note Prior to the built-in compiler support, a C header file defined a **wchar_t** data type as follows:

```
typedef unsigned short wchar_t;
```

Here is how you declare a Unicode character and string:

```
// A 16-bit character
wchar_t c = L'A';
```

```
// An array up to 99 16-bit characters and a 16-bit terminating zero.
wchar_t szBuffer[100] = L"A String";
```

An uppercase **L** before a literal string informs the compiler that the string should be compiled as a Unicode string. When the compiler places the string in the program's data section, it encodes each character using UTF16, interspersing zero bytes between every ASCII character in this simple case.

The Windows team at Microsoft wants to define its own data types to isolate itself a little bit from the C language. And so, the Windows header file, WinNT.h, defines the following data types:

typedef char CHAR; // An 8-bit character

```
typedef wchar_t WCHAR; // A 16-bit character
```

Furthermore, the WinNT.h header file defines a bunch of convenience data types for working with pointers to characters and pointers to strings:

```
// Pointer to 8-bit character(s)
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR
// Pointer to 16-bit character(s)
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR *PCWSTR;
```



Note If you take a look at WinNT.h, you'll find the following definition:

typedef __nullterminated WCHAR *NWPSTR, *LPWSTR, *PWSTR;

The **__nullterminated** prefix is a *header annotation* that describes how types are expected to be used as function parameters and return values. In the Enterprise version of Visual Studio, you can set the Code Analysis option in the project properties. This adds the **/analyze** switch to the command line of the compiler that detects when your code calls functions in a way that breaks the semantic defined by the annotations. Notice that only Enterprise versions of the compiler support this **/analyze** switch. To keep the code more readable in this book, the header annotations are removed. You should read the "Header Annotations" documentation on MSDN at *http://msdn2.microsoft.com/En-US/library/ aa383701.aspx* for more details about the header annotations language. In your own source code, it doesn't matter which data type you use, but I'd recommend you try to be consistent to improve maintainability in your code. Personally, as a Windows programmer, I always use the Windows data types because the data types match up with the MSDN documentation, making things easier for everyone reading the code.

It is possible to write your source code so that it can be compiled using ANSI or Unicode characters and strings. In the WinNT.h header file, the following types and macros are defined:

#ifdef UNICODE

```
typedef WCHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) quote // r_winnt
#define __TEXT(quote) L##quote
#else
typedef CHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST CHAR *PCTSTR;
#define __TEXT(quote) quote
#endif
#define TEXT(quote) __TEXT(quote)
```

These types and macros (plus a few less commonly used ones that I do not show here) are used to create source code that can be compiled using either ANSI or Unicode chacters and strings, for example:

```
// If UNICODE defined, a 16-bit character; else an 8-bit character
TCHAR c = TEXT('A');
```

```
// If UNICODE defined, an array of 16-bit characters; else 8-bit characters
TCHAR szBuffer[100] = TEXT("A String");
```

Unicode and ANSI Functions in Windows

Since Windows NT, all Windows versions are built from the ground up using Unicode. That is, all the core functions for creating windows, displaying text, performing string manipulations, and so forth require Unicode strings. If you call any Windows function passing it an ANSI string (a string of 1-byte characters), the function first converts the string to Unicode and then passes the Unicode string to the operating system. If you are expecting ANSI strings back from a function, the system converts the Unicode string to an ANSI string before returning to your application. All these conversions occur invisibly to you. Of course, there is time and memory overhead involved for the system to carry out all these string conversions.

When Windows exposes a function that takes a string as a parameter, two versions of the same function are usually provided—for example, a **CreateWindowEx** that accepts Unicode strings and a

second **CreateWindowEx** that accepts ANSI strings. This is true, but the two functions are actually prototyped as follows:

```
HWND WINAPI CreateWindowExW(
   DWORD dwExStyle,
   PCWSTR pClassName, // A Unicode string
   PCWSTR pWindowName, // A Unicode string
  DWORD dwStyle,
   int X,
   int Y.
   int nWidth,
   int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
   PVOID pParam);
HWND WINAPI CreateWindowExA(
  DWORD dwExStyle,
   PCSTR pClassName, // An ANSI string
   PCSTR pWindowName, // An ANSI string
  DWORD dwStyle,
   int X,
   int Y.
   int nWidth,
   int nHeight,
  HWND hWndParent,
  HMENU hMenu,
  HINSTANCE hInstance,
   PVOID pParam);
```

CreateWindowExW is the version that accepts Unicode strings. The uppercase *W* at the end of the function name stands for *wide*. Unicode characters are 16 bits wide, so they are frequently referred to as wide characters. The uppercase *A* at the end of **CreateWindowExA** indicates that the function accepts ANSI character strings.

But usually we just include a call to **CreateWindowEx** in our code and don't directly call either **CreateWindowExW** or **CreateWindowExA**. In WinUser.h, **CreateWindowEx** is actually a macro defined as

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif
```

Whether or not **UNICODE** is defined when you compile your source code module determines which version of **CreateWindowEx** is called. When you create a new project with Visual Studio, it defines **UNICODE** by default. So, by default, any calls you make to **CreateWindowEx** expand the macro to call **CreateWindowExW**—the Unicode version of **CreateWindowEx**.

Under Windows Vista, Microsoft's source code for **CreateWindowExA** is simply a translation layer that allocates memory to convert ANSI strings to Unicode strings; the code then calls **Create WindowExW**, passing the converted strings. When **CreateWindowExW** returns, **CreateWindowExA** frees its memory buffers and returns the window handle to you. So, for functions that fill buffers with strings, the system must convert from Unicode to non-Unicode equivalents before your application can process the string. Because the system must perform all these conversions, your application requires more memory and runs slower. You can make your application perform more efficiently by developing your application using Unicode from the start. Also, Windows has been known to have some bugs in these translation functions, so avoiding them also eliminates some potential bugs.

If you're creating dynamic-link libraries (DLLs) that other software developers will use, consider using this technique: supply two exported functions in the DLL—an ANSI version and a Unicode version. In the ANSI version, simply allocate memory, perform the necessary string conversions, and call the Unicode version of the function. I'll demonstrate this process later in this chapter in "Exporting ANSI and Unicode DLL Functions" on page 29.

Certain functions in the Windows API, such as **WinExec** and **OpenFile**, exist solely for backward compatibility with 16-bit Windows programs that supported only ANSI strings. These methods should be avoided by today's programs. You should replace any calls to **WinExec** and **OpenFile** with calls to the **CreateProcess** and **CreateFile** functions. Internally, the old functions call the new functions anyway. The big problem with the old functions is that they don't accept Unicode strings and they typically offer fewer features. When you call these functions, you must pass ANSI strings. On Windows Vista, most non-obsolete functions have both Unicode and ANSI versions. However, Microsoft has started to get into the habit of producing some functions offering only Unicode versions—for example, **ReadDirectoryChangesW** and **CreateProcessWithLogonW**.

When Microsoft was porting COM from 16-bit Windows to Win32, an executive decision was made that all COM interface methods requiring a string would accept only Unicode strings. This was a great decision because COM is typically used to allow different components to talk to each other and Unicode is the richest way to pass strings around. Using Unicode throughout your application makes interacting with COM easier too.

Finally, when the resource compiler compiles all your resources, the output file is a binary representation of the resources. String values in your resources (string tables, dialog box templates, menus, and so on) are always written as Unicode strings. Under Windows Vista, the system performs internal conversions if your application doesn't define the **UNICODE** macro. For example, if **UNICODE** is not defined when you compile your source module, a call to **LoadString** will actually call the **LoadStringA** function. **LoadStringA** will then read the Unicode string from your resources and convert the string to ANSI. The ANSI representation of the string will be returned from the function to your application.

Unicode and ANSI Functions in the C Run-Time Library

Like the Windows functions, the C run-time library offers one set of functions to manipulate ANSI characters and strings and another set of functions to manipulate Unicode characters and strings. However, unlike Windows, the ANSI functions do the work; they do not translate the strings to Unicode and then call the Unicode version of the functions internally. And, of course, the Unicode versions do the work themselves too; they do not internally call the ANSI versions.

An example of a C run-time function that returns the length of an ANSI string is **strlen**, and an example of an equivalent C run-time function that returns the length of a Unicode string is **wcslen**.

Both of these functions are prototyped in String.h. To write source code that can be compiled for either ANSI or Unicode, you must also include TChar.h, which defines the following macro:

#ifdef _UNICODE
#define _tcslen wcslen
#else
#define _tcslen strlen
#endif

Now, in your code, you should call **_tcslen**. If **_UNICODE** is defined, it expands to **wcslen**; otherwise, it expands to **strlen**. By default, when you create a new C++ project in Visual Studio, **_UNICODE** is defined (just like **UNICODE** is defined). The C run-time library always prefixes identifiers that are not part of the C++ standard with underscores, while the Windows team does not do this. So, in your applications you'll want to make sure that both **UNICODE** and **_UNICODE** are defined or that neither is defined. Appendix A, "The Build Environment," will describe the details of the CmnHdr.h header file used by all the code samples of this book to avoid this kind of problem.

Secure String Functions in the C Run-Time Library

Any function that modifies a string exposes a potential danger: if the destination string buffer is not large enough to contain the resulting string, memory corruption occurs. Here is an example:

```
// The following puts 4 characters in a
// 3-character buffer, resulting in memory corruption
WCHAR szBuffer[3] = L"";
wcscpy(szBuffer, L"abc"); // The terminating 0 is a character too!
```

The problem with the **strcpy** and **wcscpy** functions (and most other string manipulation functions) is that they do not accept an argument specifying the maximum size of the buffer, and therefore, the function doesn't know that it is corrupting memory. Because the function doesn't know that it is corrupting memory, it can't report an error back to your code, and therefore, you have no way of knowing that memory was corrupted. And, of course, it would be best if the function just failed without corrupting any memory at all.

This kind of misbehavior has been heavily exploited by malware in the past. Microsoft is now providing a set of new functions that replace the unsafe string manipulation functions (such as **wcscat**, which was shown earlier) provided by the C run-time library that many of us have grown to know and love over the years. To write safe code, you should no longer use any of the familiar C run-time functions that modify a string. (Functions such as **strlen**, **wcslen**, and **_tcslen** are OK, however, because they do not attempt to modify the string passed to them even though they assume that the string is **0** terminated, which might not be the case.) Instead, you should take advantage of the new secure string functions defined by Microsoft's StrSafe.h file.



Note Internally, Microsoft has retrofitted its ATL and MFC class libraries to use the new safe string functions, and therefore, if you use these libraries, rebuilding your application to the new versions is all you have to do to make your application more secure.

Because this book is not dedicated to C/C++ programming, for a detailed usage of this library, you should take a look at the following sources of information:

- The MSDN Magazine article "Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries" by Martyn Lovell, located at http://msdn.microsoft.com/msdnmag/ issues/05/05/SafeCandC/default.aspx
- The Martyn Lovell video presentation on Channel9, located at *http://channel9.msdn.com/ Showpost.aspx?postid=186406*
- The secure strings topic on MSDN Online, located at *http://msdn2.microsoft.com/en-us/ library/ms647466.aspx*
- The list of all C run-time secured replacement functions on MSDN Online, which you can find at *http://msdn2.microsoft.com/en-us/library/wd3wzwts*(VS.80).*aspx*

However, it is worth discussing a couple of details in this chapter. I'll start by describing the patterns employed by the new functions. Next, I'll mention the pitfalls you might encounter if you are following the migration path from legacy functions to their corresponding secure versions, like using **_tcscpy_s** instead of **_tcscpy**. Then I'll show you in which case it might be more interesting to call the new **StringC*** functions instead.

Introducing the New Secure String Functions

When you include StrSafe.h, String.h is also included and the existing string manipulation functions of the C run-time library, such as those behind the **_tcscpy** macro, are flagged with obsolete warnings during compilation. Note that the inclusion of StrSafe.h must appear after all other include files in your source code. I recommend that you use the compilation warnings to explicitly replace all the occurrences of the deprecated functions by their safer substitutes—thinking each time about possible buffer overflow and, if it is not possible to recover, how to gracefully terminate the application.

Each existing function, like **_tcscpy** or **_tcscat**, has a corresponding new function that starts with the same name that ends with the **_s** (for *secure*) suffix. All these new functions share common characteristics that require explanation. Let's start by examining their prototypes in the following code snippet, which shows the side-by-side definitions of two usual string functions:

```
PTSTR _tcscpy (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscpy_s(PTSTR strDestination, size_t numberOfCharacters,
    PCTSTR strSource);
PTSTR _tcscat (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscat_s(PTSTR strDestination, size_t numberOfcharacters,
    PCTSTR strSource);
```

When a writable buffer is passed as a parameter, its size must also be provided. This value is expected in the character count, which is easily computed by using the **_countof** macro (defined in stdlib.h) on your buffer.

All of the secure (**_s**) functions validate their arguments as the first thing they do. Checks are performed to make sure that pointers are not **NULL**, that integers are within a valid range, that enumeration values are valid, and that buffers are large enough to hold the resulting data. If any of these checks fail, the functions set the thread-local *C* run-time variable **errno** and the function returns

an **errno_t** value to indicate success or failure. However, these functions don't actually return; instead, in a debug build, they display a user-unfriendly assertion dialog box similar to that shown in Figure 2-1. Then your application is terminated. The release builds directly auto-terminate.

Microsoft	Microsoft Visual C++ Debug Library	
8	Debug Assertion Failed! Program: File: Atcscpy_s.inl Line: 30 Expression: (L"Buffer is too small" && 0) For information on how your program can cause an assertion failure, see the Visual C++ documentation on asserts. (Press Retry to debug the application)	
	Abort Retry Ignore	

Figure 2-1 Assertion dialog box displayed when an error occurs

The C run time actually allows you to provide a function of your own, which it will call when it detects an invalid parameter. Then, in this function, you can log the failure, attach a debugger, or do whatever you like. To enable this, you must first define a function that matches the following prototype:

```
void InvalidParameterHandler(PCTSTR expression, PCTSTR function,
    PCTSTR file, unsigned int line, uintptr_t /*pReserved*/);
```

The **expression** parameter describes the failed expectation in the C run-time implementation code, such as **(L'Buffer is too small' && 0)**. As you can see, this is not very user friendly and should not be shown to the end user. This comment also applies to the next three parameters because **function**, **file**, and **line** describe the function name, the source code file, and the source code line number where the error occurred, respectively.



Note All these arguments will have a value of **NULL** if **DEBUG** is not defined. So this handler is valuable for logging errors only when testing debug builds. In a release build, you could replace the assertion dialog box with a more user-friendly message explaining that an unexpected error occurred that requires the application to shut down—maybe with specific logging behavior or an application restart. If its memory state is corrupted, your application execution should stop. However, it is recommended that you wait for the **errno_t** check to decide whether the error is recoverable or not.

The next step is to register this handler by calling **_set_invalid_parameter_handler**. However, this step is not enough because the assertion dialog box will still appear. You need to call **_CrtSetReportMode(_CRT_ASSERT, 0);** at the beginning of your application, disabling all assertion dialog boxes that could be triggered by the C run time.

Now, when you call one of the legacy replacement functions defined in String.h, you are able to check the returned **errno_t** value to understand what happened. Only the value **S_OK** means that
the call was successful. The other possible return values found in errno.h, such as **EINVAL**, are for invalid arguments such as **NULL** pointers.

Let's take an example of a string that is copied into a buffer that is too small for one character:

```
TCHAR szBefore[5] = {
    TEXT('B'), TEXT('B'), TEXT('B'), '\0'
};
TCHAR szBuffer[10] = {
    TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), '\0'
};
TCHAR szAfter[5] = {
    TEXT('A'), TEXT('A'), TEXT('A'), TEXT('A'), '\0'
};
errno_t result = _tcscpy_s(szBuffer, _countof(szBuffer), TEXT("0123456789"));
```

Just before the call to **_tcscpy_s**, each variable has the content shown in Figure 2-2.

Watch 1			▼ ₽	×
Name	Value		Туре	*
표 🧳 szBefore	0x0012f9f4 "BBBB"	Q +	wchar_t [5]	
🗄 🧳 szBuffer	0x0012f9d8 ""	Q, ,	wchar_t [10]	
🗄 🧳 szAfter	0x0012f9c4 "AAAA"	Q ,	wchar_t [5]	
				Ŧ

Figure 2-2 Variable state before the <u>tcscpy</u>s call

Because the "1234567890" string to be copied into **szBuffer** has exactly the same 10-character size as the buffer, there is not enough room to copy the terminating '**\0**' character. You might expect that the value of **result** is now **STRUNCATE** and the last character '**9**' has not been copied, but this is not the case. **ERANGE** is returned, and the state of each variable is shown in Figure 2-3.

1	Nat	ch 1			↓ ₽	×
	Na	me	Value		Туре	*
	+	🧳 szBefore	0x0012f9f4 "BBBB"	Q +	wchar_t [5]	
	+	🧳 szBuffer	0x0012f9d8 ""	Q, ,	wchar_t [10]	
	+	🧼 szAfter	0x0012f9c4 "AAAA"	Q, ,	wchar_t [5]	
						Ŧ

Figure 2-3 Variable state after the _tcscpy_s call

There is one side effect that you don't see unless you take a look at the memory behind **szBuffer**, as shown in Figure 2-4.

Figure 2-4 Content of szBuffer memory after a failed call

The first character of **szBuffer** has been set to '\0', and all other bytes now contain the value **0xfd**. So the resulting string has been truncated to an empty string and the remaining bytes of the buffer have been set to a filler value (**0xfd**).



Note If you wonder why the memory after all the variables have been defined is filled up with the **0xcc** value in Figure 2-4, the answer is in the result of the compiler implementation of the run-time checks (**/RTCs**, **/RTCu**, or **/RTC1**) that automatically detect buffer overrun at run time. If you compile your code without these **/RTCx** flags, the memory view will show all **sz*** variables side by side. But remember that your builds should always be compiled with these run-time checks to detect any remaining buffer overrun early in the development cycle.

How to Get More Control When Performing String Operations

In addition to the new secure string functions, the C run-time library has some new functions that provide more control when performing string manipulations. For example, you can control the filler values or how truncation is performed. Naturally, the C run time offers both ANSI (A) versions of the functions as well as Unicode (W) versions of the functions. Here are the prototypes for some of these functions (and many more exist that are not shown here):

```
HRESULT StringCchCat(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCatEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);
HRESULT StringCchCopy(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCopyEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);
HRESULT StringCchPrintf(PTSTR pszDest, size_t cchDest,
PCTSTR pszFormat, ...);
HRESULT StringCchPrintfEx(PTSTR pszDest, size_t cchDest,
PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags,
PCTSTR pszFormat, ...);
```

You'll notice that all the methods shown have "Cch" in their name. This stands for *Count of characters*, and you'll typically use the **_countof** macro to get this value. There is also a set of functions that have "Cb" in their name, such as **StringCbCat(Ex)**, **StringCbCopy(Ex)**, and **StringCb Printf(Ex)**. These functions expect that the size argument is in count of bytes instead of count of characters. You'll typically use the **sizeof** operator to get this value.

All these functions return an **HRESULT** with one of the values shown in Table 2-2.

Table 2-2 HRESULT Valu	es for Safe String Functions
------------------------	------------------------------

HRESULT Value	Description
S_0К	Success. The destination buffer contains the source string and is terminated by $' \0'$.
STRSAFE_E_INVALID_PARAMETER	Failure. The NULL value has been passed as a parameter.
STRSAFE_E_INSUFFICIENT_BUFFER	Failure. The given destination buffer was too small to contain the entire source string.

Unlike the secure (**_s** suffixed) functions, when a buffer is too small, these functions do perform truncation. You can detect such a situation when **STRSAFE_E_INSUFFICIENT_BUFFER** is returned. As you can see in StrSafe.h, the value of this code is **0x8007007a** and is treated as a failure by

SUCCEEDED/FAILED macros. However, in that case, the part of the source buffer that could fit into the destination writable buffer has been copied and the last available character is set to '**\0**'. So, in the previous example, **szBuffer** would contain the string "012345678" if **StringCchCopy** is used instead of **_tcscpy_s**. Notice that the truncation feature might or might not be what you need, depending on what you are trying to achieve, and this is why it is treated as a failure (by default). For example, in the case of a path that you are building by concatenating different pieces of information, a truncated result is unusable. If you are building a message for user feedback, this could be acceptable. It's up to you to decide how to handle a truncated result.

Last but not least, you'll notice that an extended (**Ex**) version exists for many of the functions shown earlier. These extended versions take three additional parameters, which are described in Table 2-3.

Parameters and Values	Description
size_t* pcchRemaining	Pointer to a variable that indicates the number of unused characters in the destination buffer. The copied terminating ' \0 ' character is not counted. For example, if one character is copied into a buffer that is 10 characters wide, 9 is returned even though you won't be able to use more than 8 characters without truncation. If pcchRemaining is NULL , the count is not returned.
LPTSTR* ppszDestEnd	If ppszDestEnd is non- NULL , it points to the terminating ' \0' character at the end of the string contained by the destination buffer.
DWORD dwFlags	One or more of the following values separated by ' '.
STRSAFE_FILL_BEHIND_NULL	If the function succeeds, the low byte of dwFlags is used to fill the rest of the destination buffer, just after the terminating '\0' character. (See the comment about STRSAFE_FILL_BYTE just after this table for more details.)
STRSAFE_IGNORE_NULLS	Treats NULL string pointers like empty strings (TEXT("")).
STRSAFE_FILL_ON_FAILURE	If the function fails, the low byte of dwFlags is used to fill the entire destination buffer except the first '\0' character used to set an empty string result. (See the comment about STRSAFE_FILL_BYTE just after this table for more details.) In the case of a STRSAFE_E_INSUFFICIENT_BUFFER failure, any character in the string being returned is replaced by the filler byte value.
STRSAFE_NULL_ON_FAILURE	If the function fails, the first character of the destination buffer is set to '\0' to define an empty string (TEXT("")). In the case of a STRSAFE_E_INSUFFICIENT_BUFFER failure, any truncated string is overwritten.
STRSAFE_NO_TRUNCATION	As in the case of STRSAFE_NULL_ON_FAILURE , if the function fails, the destination buffer is set to an empty string (TEXT('''')) . In the case of a STRSAFE_E_INSUFFICIENT_BUFFER failure, any truncated string is overwritten.

Table 2-3 Extended Version Parameters



Note Even if **STRSAFE_NO_TRUNCATION** is used as a flag, the characters of the source string are still copied, up to the last available character of the destination buffer. Then both the first and the last characters of the destination buffer are set to '\0'. This is not really important except if, for security purposes, you don't want to keep garbage data.

There is a last detail to mention that is related to the remark that you read at the bottom of page 21. In Figure 2-4, the **0xfd** value is used to replace all the characters after the **'\0'**, up to the end of the destination buffer. With the **Ex** version of these functions, you can choose whether you want this expensive filling operation (especially if the destination buffer is large) to occur and with which byte value. If you add **STRSAFE_FILL_BEHIND_NULL** to **dwFlag**, the remaining characters are set to **'\0'**. When you replace **STRSAFE_FILL_BEHIND_NULL** with the **STRSAFE_FILL_BYTE** macro, the given byte value is used to fill up the remaining values of the destination buffer.

Windows String Functions

Windows also offers various functions for manipulating strings. Many of these functions, such as **lstrcat** and **lstrcpy**, are now deprecated because they do not detect buffer overrun problems. Also, the ShlwApi.h file defines a number of handy string functions that format operating system—related numeric values, such as **StrFormatKBSize** and **StrFormatByteSize**. See *http://msdn2.microsoft.com/en-us/library/ms538658.aspx* for a description of shell string handling functions.

It is common to want to compare strings for equality or for sorting. The best functions to use for this are **CompareString(Ex)** and **CompareStringOrdinal**. You use **CompareString(Ex)** to compare strings that will be presented to the user in a linguistically correct manner. Here is the prototype of the **CompareString** function:

```
int CompareString(
   LCID locale,
   DWORD dwCmdFlags,
   PCTSTR pString1,
   int cch1,
   PCTSTR pString2,
   int cch2);
```

This function compares two strings. The first parameter to **CompareString** specifies a locale ID (LCID), a 32-bit value that identifies a particular language. **CompareString** uses this LCID to compare the two strings by checking the meaning of the characters as they apply to a particular language. A linguistically correct comparison produces results much more meaningful to an end user. However, this type of comparison is slower than doing an ordinal comparison. You can get the locale ID of the calling thread by calling the Windows **GetThreadLocale** function:

LCID GetThreadLocale();

The second parameter of **CompareString** identifies flags that modify the method used by the function to compare the two strings. Table 2-4 shows the possible flags.

Flag	Meaning
NORM_IGNORECASE	Ignore case difference.
LINGUISTIC_IGNORECASE	
NORM_IGNOREKANATYPE	Do not differentiate between hiragana and katakana characters.
NORM_IGNORENONSPACE	Ignore nonspacing characters.
LINGUISTIC_IGNOREDIACRITIC	
NORM_IGNORESYMBOLS	Ignore symbols.
NORM_IGNOREWIDTH	Do not differentiate between a single-byte character and the same character as a double-byte character.
SORT_STRINGSORT	Treat punctuation the same as symbols.

Table 2-4 Flags Used by the CompareString Function

The remaining four parameters of **CompareString** specify the two strings and their respective lengths in characters (not in bytes). If you pass negative values for the **cch1** parameter, the function assumes that the **pString1** string is zero-terminated and calculates the length of the string. This also is true for the **cch2** parameter with respect to the **pString2** string. If you need more advanced linguistic options, you should take a look at the **CompareStringEx** functions.

To compare strings that are used for programmatic strings (such as pathnames, registry keys/ values, XML elements/attributes, and so on), use **CompareStringOrdinal**:

```
int CompareStringOrdinal(
    PCWSTR pString1,
    int cchCount1,
    PCWSTR pString2,
    int cchCount2,
    BOOL bIgnoreCase);
```

This function performs a code-point comparison without regard to the locale, and therefore it is fast. And because programmatic strings are not typically shown to an end user, this function makes the most sense. Notice that only Unicode strings are expected by this function.

The **CompareString** and **CompareStringOrdinal** functions' return values are unlike the return values you get back from the C run-time library's ***cmp** string comparison functions. **Compare String(Ordinal**) returns **0** to indicate failure, **CSTR_LESS_THAN** (defined as **1**) to indicate that **pString1** is less than **pString2**, **CSTR_EQUAL** (defined as **2**) to indicate that **pString1** is greater than **pString2**, and **CSTR_GREATER_THAN** (defined as **3**) to indicate that **pString1** is greater than **pString2**. To make things slightly more convenient, if the functions succeed, you can subtract **2** from the return value to make the result consistent with the result of the C run-time library functions (**-1**, **0**, and **+1**).

Why You Should Use Unicode

When developing an application, we highly recommend that you use Unicode characters and strings. Here are some of the reasons why:

- Unicode makes it easy for you to localize your application to world markets.
- Unicode allows you to distribute a single binary (.exe or DLL) file that supports all languages.
- Unicode improves the efficiency of your application because the code performs faster and uses less memory. Windows internally does everything with Unicode characters and strings, so when you pass an ANSI character or string, Windows must allocate memory and convert the ANSI character or string to its Unicode equivalent.
- Using Unicode ensures that your application can easily call all nondeprecated Windows functions, as some Windows functions offer versions that operate only on Unicode characters and strings.
- Using Unicode ensures that your code easily integrates with COM (which requires the use of Unicode characters and strings).
- Using Unicode ensures that your code easily integrates with the .NET Framework (which also requires the use of Unicode characters and strings).
- Using Unicode ensures that your code easily manipulates your own resources (where strings are always persisted as Unicode).

How We Recommend Working with Characters and Strings

Based on what you've read in this chapter, the first part of this section summarizes what you should always keep in mind when developing your code. The second part of the section provides tips and tricks for better Unicode and ANSI string manipulations. It's a good idea to start converting your application to be Unicode-ready even if you don't plan to use Unicode right away. Here are the basic guidelines you should follow:

- Start thinking of text strings as arrays of characters, not as arrays of **chars** or arrays of bytes.
- Use generic data types (such as **TCHAR/PTSTR**) for text characters and strings.
- Use explicit data types (such as **BYTE** and **PBYTE**) for bytes, byte pointers, and data buffers.
- Use the **TEXT** or _**T** macro for literal characters and strings, but avoid mixing both for the sake of consistency and for better readability.
- Perform global replaces. (For example, replace **PSTR** with **PTSTR**.)
- Modify string arithmetic problems. For example, functions usually expect you to pass a buffer's size in characters, not bytes. This means you should pass _countof(szBuffer) instead of sizeof(szBuffer). Also, if you need to allocate a block of memory for a string and you have the number of characters in the string, remember that you allocate memory in bytes. This means that you must call malloc(nCharacters * sizeof(TCHAR)) and not call malloc(nCharacters). Of all the guidelines I've just listed, this is the most difficult one to remember, and the compiler offers no warnings or errors if you make a mistake. This is a good opportunity to define your own macros, such as the following:

#define chmalloc(nCharacters) (TCHAR*)malloc(nCharacters * sizeof(TCHAR)).

- Avoid **printf** family functions, especially by using %s and %S field types to convert ANSI to Unicode strings and vice versa. Use **MultiByteToWideChar** and **WideCharToMultiByte** instead, as shown in "Translating Strings Between Unicode and ANSI" below.
- Always specify both UNICODE and _UNICODE symbols or neither of them.

In terms of string manipulation functions, here are the basic guidelines that you should follow:

- Always work with safe string manipulation functions such as those suffixed with <u>s</u> or prefixed with **StringCch**. Use the latter for explicit truncation handling, but prefer the former otherwise.
- Don't use the unsafe C run-time library string manipulation functions. (See the previous recommendation.) In a more general way, don't use or implement any buffer manipulation routine that would not take the size of the destination buffer as a parameter. The C run-time library provides a replacement for buffer manipulation functions such as memcpy_s, memmove_s, wmemcpy_s, or wmemmove_s. All these methods are available when the ___STDC_WANT_SECURE_LIB__ symbol is defined, which is the case by default in CrtDefs.h. So don't undefine __STDC_WANT_SECURE_LIB__.
- Take advantage of the /GS (*http://msdn2.microsoft.com/en-us/library/aa290051*(VS.71).*aspx*) and /RTCs compiler flags to automatically detect buffer overruns.
- Don't use Kernel32 methods for string manipulation such as **lstrcat** and **lstrcpy**.
- There are two kinds of strings that we compare in our code. Programmatic strings are file names, paths, XML elements and attributes, and registry keys/values. For these, use CompareStringOrdinal, as it is very fast and does not take the user's locale into account. This is good because these strings remain the same no matter where your application is running in the world. User strings are typically strings that appear in the user interface. For these, call CompareString(Ex), as it takes the locale into account when comparing strings.

You don't have a choice: as a professional developer, you can't write code based on unsafe buffer manipulation functions. And this is the reason why all the code in this book relies on these safer functions from the C run-time library.

Translating Strings Between Unicode and ANSI

You use the Windows function **MultiByteToWideChar** to convert multibyte-character strings to wide-character strings. **MultiByteToWideChar** is shown here:

```
int MultiByteToWideChar(
    UINT uCodePage,
    DWORD dwFlags,
    PCSTR pMultiByteStr,
    int cbMultiByte,
    PWSTR pWideCharStr,
    int cchWideChar);
```

The **uCodePage** parameter identifies a code page number that is associated with the multibyte string. The **dwFlags** parameter allows you to specify an additional control that affects characters with diacritical marks such as accents. Usually the flags aren't used, and **0** is passed in the **dwFlags** parameter (For more details about the possible values for this flag, read the MSDN online help at *http://msdn2.microsoft.com/en-us/library/ms776413.aspx.*) The **pMultiByteStr** parameter

28 WIndows via C/C++

specifies the string to be converted, and the **cbMultiByte** parameter indicates the length (in bytes) of the string. The function automatically determines the length of the source string if you pass -1 for the **cbMultiByte** parameter.

The Unicode version of the string resulting from the conversion is written to the buffer located in memory at the address specified by the **pWideCharStr** parameter. You must specify the maximum size of this buffer (in characters) in the **cchWideChar** parameter. If you call **MultiByteToWide Char**, passing **0** for the **cchWideChar** parameter, the function doesn't perform the conversion and instead returns the number of wide characters (including the terminating '**0**' character) that the buffer must provide for the conversion to succeed. Typically, you convert a multibyte-character string to its Unicode equivalent by performing the following steps:

- 1. Call MultiByteToWideChar, passing NULL for the pWideCharStr parameter and 0 for the cchWideChar parameter and -1 for the cbMultiByte parameter.
- Allocate a block of memory large enough to hold the converted Unicode string. This size is computed based on the value returned by the previous call to MultiByteToWideChar multiplied by sizeof(wchar_t).
- 3. Call MultiByteToWideChar again, this time passing the address of the buffer as the pWideCharStr parameter and passing the size computed based on the value returned by the first call to MultiByteToWideChar multiplied by sizeof(wchar_t) as the cchWideChar parameter.
- 4. Use the converted string.
- 5. Free the memory block occupying the Unicode string.

The function **WideCharToMultiByte** converts a wide-character string to its multibyte-string equivalent, as shown here:

```
int WideCharToMultiByte(
    UINT uCodePage,
    DWORD dwFlags,
    PCWSTR pWideCharStr,
    int cchWideChar,
    PSTR pMultiByteStr,
    int cbMultiByte,
    PCSTR pDefaultChar,
    PBOOL pfUsedDefaultChar);
```

This function is similar to the **MultiByteToWideChar** function. Again, the **uCodePage** parameter identifies the code page to be associated with the newly converted string. The **dwFlags** parameter allows you to specify additional control over the conversion. The flags affect characters with diacritical marks and characters that the system is unable to convert. Usually, you won't need this degree of control over the conversion, and you'll pass **0** for the **dwFlags** parameter.

The **pWideCharStr** parameter specifies the address in memory of the string to be converted, and the **cchWideChar** parameter indicates the length (in characters) of this string. The function determines the length of the source string if you pass -1 for the **cchWideChar** parameter.

The multibyte version of the string resulting from the conversion is written to the buffer indicated by the **pMultiByteStr** parameter. You must specify the maximum size of this buffer (in bytes) in the **cbMultiByte** parameter. Passing **0** as the **cbMultiByte** parameter of the **WideCharToMulti Byte** function causes the function to return the size required by the destination buffer. You'll typically convert a wide-character string to a multibyte-character string using a sequence of events similar to those discussed when converting a multibyte string to a wide-character string, except that the return value is directly the number of bytes required for the conversion to succeed.

You'll notice that the **WideCharToMultiByte** function accepts two parameters more than the **MultiByteToWideChar** function: **pDefaultChar** and **pfUsedDefaultChar**. These parameters are used by the **WideCharToMultiByte** function only if it comes across a wide character that doesn't have a representation in the code page identified by the **uCodePage** parameter. If the wide character cannot be converted, the function uses the character pointed to by the **pDefaultChar** parameter. If this parameter is **NULL**, which is most common, the function uses a system default character. This default character is usually a question mark. This is dangerous for filenames because the question mark is a wildcard character.

The **pfUsedDefaul tChar** parameter points to a Boolean variable that the function sets to **TRUE** if at least one character in the wide-character string could not be converted to its multibyte equivalent. The function sets the variable to **FALSE** if all the characters convert successfully. You can test this variable after the function returns to check whether the wide-character string was converted successfully. Again, you usually pass **NULL** for this parameter.

For a more complete description of how to use these functions, please refer to the Platform SDK documentation.

Exporting ANSI and Unicode DLL Functions

You could use these two functions to easily create both Unicode and ANSI versions of functions. For example, you might have a dynamic-link library containing a function that reverses all the characters in a string. You could write the Unicode version of the function as shown here:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength) {
```

}

```
// Get a pointer to the last character in the string.
PWSTR pEndOfStr = pWideCharStr + wcsnlen_s(pWideCharStr , cchLength) - 1;
wchar_t cCharT;
// Repeat until we reach the center character in the string.
while (pWideCharStr < pEndOfStr) {</pre>
   // Save a character in a temporary variable.
   cCharT = *pWideCharStr;
   // Put the last character in the first character.
   *pWideCharStr = *pEndOfStr;
   // Put the temporary character in the last character.
   *pEndOfStr = cCharT;
   // Move in one character from the left.
   pWideCharStr++;
   // Move in one character from the right.
   pEndOfStr--;
}
// The string is reversed; return success.
return(TRUE);
```

}

And you could write the ANSI version of the function so that it doesn't perform the actual work of reversing the string at all. Instead, you could write the ANSI version so that it converts the ANSI string to Unicode, passes the Unicode string to the StringReverseW function, and then converts the reversed string back to ANSI. The function would look like this:

```
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength) {
   PWSTR pWideCharStr;
   int nLenOfWideCharStr;
   BOOL fOk = FALSE:
  // Calculate the number of characters needed to hold
   // the wide-character version of the string.
   nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
     pMultiByteStr, cchLength, NULL, 0);
  // Allocate memory from the process' default heap to
   // accommodate the size of the wide-character string.
  // Don't forget that MultiByteToWideChar returns the
  // number of characters, not the number of bytes, so
   // you must multiply by the size of a wide character.
   pWideCharStr = (PWSTR)HeapAlloc(GetProcessHeap(), 0,
      nLenOfWideCharStr * sizeof(wchar_t));
   if (pWideCharStr == NULL)
      return(f0k);
   // Convert the multibyte string to a wide-character string.
  MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, cchLength,
      pWideCharStr, nLenOfWideCharStr);
  // Call the wide-character version of this
   // function to do the actual work.
   f0k = StringReverseW(pWideCharStr, cchLength);
   if (f0k) {
     // Convert the wide-character string back
     // to a multibyte string.
     WideCharToMultiByte(CP_ACP, 0, pWideCharStr, cchLength,
         pMultiByteStr, (int)strlen(pMultiByteStr), NULL, NULL);
   }
  // Free the memory containing the wide-character string.
  HeapFree(GetProcessHeap(), 0, pWideCharStr);
   return(f0k);
```

Finally, in the header file that you distribute with the dynamic-link library, you prototype the two functions as follows:

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength);
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength);
#ifdef UNICODE
#define StringReverse StringReverseW
#else
#define StringReverse StringReverseA
#endif // !UNICODE
```

Determining If Text Is ANSI or Unicode

The Windows Notepad application allows you to open both Unicode and ANSI files as well as create them. In fact, Figure 2-5 shows Notepad's File Save As dialog box. Notice the different ways that you can save a text file.



Figure 2-5 The Windows Vista Notepad File Save As dialog box

For many applications that open text files and process them, such as compilers, it would be convenient if, after opening a file, the application could determine whether the text file contained ANSI characters or Unicode characters. The **IsTextUnicode** function exported by AdvApi32.dll and declared in WinBase.h can help make this distinction:

BOOL IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);

The problem with text files is that there are no hard and fast rules as to their content. This makes it extremely difficult to determine whether the file contains ANSI or Unicode characters. **IsText Unicode** uses a series of statistical and deterministic methods to guess at the content of the buffer. Because this is not an exact science, it is possible that **IsTextUnicode** will return an incorrect result.

32 WIndows via C/C++

The first parameter, **pvBuffer**, identifies the address of a buffer that you want to test. The data is a void pointer because you don't know whether you have an array of ANSI characters or an array of Unicode characters.

The second parameter, **cb**, specifies the number of bytes that **pvBuffer** points to. Again, because you don't know what's in the buffer, **cb** is a count of bytes rather than a count of characters. Note that you do not have to specify the entire length of the buffer. Of course, the more bytes **IsText Unicode** can test, the more accurate a response you're likely to get.

The third parameter, **pResult**, is the address of an integer that you must initialize before calling **IsTextUnicode**. You initialize this integer to indicate which tests you want **IsTextUnicode** to perform. You can also pass **NULL** for this parameter, in which case **IsTextUnicode** will perform every test it can. (See the Platform SDK documentation for more details.)

If **IsTextUnicode** thinks that the buffer contains Unicode text, **TRUE** is returned; otherwise, **FALSE** is returned. If specific tests were requested in the integer pointed to by the **pResult** parameter, the function sets the bits in the integer before returning to reflect the results of each test.

The FileRev sample application presented in Chapter 17, "Memory-Mapped Files," demonstrates the use of the **IsTextUnicode** function.

Chapter 10 Synchronous and Asynchronous Device I/O

In this chapter:	
Opening and Closing Devices	290
Working with File Devices	299
Performing Synchronous Device I/O	302
Basics of Asynchronous Device I/O	305
Receiving Completed I/O Request Notifications	310

I can't stress enough the importance of this chapter, which covers the Microsoft Windows technologies that enable you to design high-performance, scalable, responsive, and robust applications. A scalable application handles a large number of concurrent operations as efficiently as it handles a small number of concurrent operations. For a service application, typically these operations are processing client requests that arrive at unpredictable times and require an unpredictable amount of processing power. These requests usually arrive from I/O devices such as network adapters; processing the requests frequently requires additional I/O devices such as disk files.

In Microsoft Windows applications, threads are the best facility available to help you partition work. Each thread is assigned to a processor, which allows a multiprocessor machine to execute multiple operations simultaneously, increasing throughput. When a thread issues a synchronous device I/O request, the thread is temporarily suspended until the device completes the I/O request. This suspension hurts performance because the thread is unable to do useful work, such as initiate another client's request for processing. So, in short, you want to keep your threads doing useful work all the time and avoid having them block.

To help keep threads busy, you need to make your threads communicate with one another about the operations they will perform. Microsoft has spent years researching and testing in this area and has developed a finely tuned mechanism to create this communication. This mechanism, called the I/O completion port, can help you create high-performance, scalable applications. By using the I/O completion port, you can make your application's threads achieve phenomenal throughput by reading and writing to devices without waiting for the devices to respond.

The I/O completion port was originally designed to handle device I/O, but over the years, Microsoft has architected more and more operating system facilities that fit seamlessly into the I/O completion port model. One example is the job kernel object, which monitors its processes and sends event notifications to an I/O completion port. The Job Lab sample application detailed in Chapter 5, "Jobs," demonstrates how I/O completion ports and job objects work together.

Throughout my many years as a Windows developer, I have found more and more uses for the I/O completion port, and I feel that every Windows developer must fully understand how the I/O completion port works. Even though I present the I/O completion port in this chapter about device I/O, be aware that the I/O completion port doesn't have to be used with device I/O at

all-simply put, it is an awesome interthread communication mechanism with an infinite number of uses.

From this fanfare, you can probably tell that I'm a huge fan of the I/O completion port. My hope is that by the end of this chapter, you will be too. But instead of jumping right into the details of the I/O completion port, I'm going to explain what Windows originally offered developers for device I/O. This will give you a much greater appreciation for the I/O completion port. In "I/O Completion Ports" on page 320 I'll discuss the I/O completion port.

Opening and Closing Devices

One of the strengths of Windows is the sheer number of devices that it supports. In the context of this discussion, I define a device to be anything that allows communication. Table 10-1 lists some devices and their most common uses.

Device	Most Common Use
File	Persistent storage of arbitrary data
Directory	Attribute and file compression settings
Logical disk drive	Drive formatting
Physical disk drive	Partition table access
Serial port	Data transmission over a phone line
Parallel port	Data transmission to a printer
Mailslot	One-to-many transmission of data, usually over a network to a machine running Windows
Named pipe	One-to-one transmission of data, usually over a network to a machine running Windows
Anonymous pipe	One-to-one transmission of data on a single machine (never over the network)
Socket	Datagram or stream transmission of data, usually over a network to any machine supporting sockets (The machine need not be running Windows.)
Console	A text window screen buffer

Table 10-1 Various Devices and Their Common Uses

This chapter discusses how an application's threads communicate with these devices without waiting for the devices to respond. Windows tries to hide device differences from the software developer as much as possible. That is, once you open a device, the Windows functions that allow you to read and write data to the device are the same no matter what device you are communicating with. Although only a few functions are available for reading and writing data regardless of the device, devices are certainly different from one another. For example, it makes sense to set a baud rate for a serial port, but a baud rate has no meaning when using a named pipe to communicate over a network (or over the local machine). Devices are subtly different from one another, and I will not attempt to address all their nuances. However, I will spend some time addressing files because files are so common. To perform any type of I/O, you must first open the desired device and get a handle to it. The way you get the handle to a device depends on the particular device. Table 10-2 lists various devices and the functions you should call to open them.

Device	Function Used to Open the Device
File	CreateFile (pszName is pathname or UNC pathname).
Directory	CreateFile (pszName is directory name or UNC directory name). Windows allows you to open a directory if you specify the FILE_ FLAG_BACKUP_SEMANTICS flag in the call to CreateFile . Open- ing the directory allows you to change the directory's attributes (to normal, hidden, and so on) and its time stamp.
Logical disk drive	CreateFile (pszName is "\\.\x:"). Windows allows you to open a logical drive if you specify a string in the form of "\\.\x:" where x is a drive letter. For example, to open drive A, you specify "\\.\A:". Opening a drive allows you to format the drive or determine the media size of the drive.
Physical disk drive	CreateFile (pszName is "\\\PHYSICALDRIVEx"). Windows allows you to open a physical drive if you specify a string in the form of "\\.PHYSICALDRIVEx" where <i>x</i> is a physical drive number. For example, to read or write to physical sectors on the user's first physical hard disk, you specify "\\.PHYSICALDRIVE0". Opening a physical drive allows you to access the hard drive's partition tables directly. Opening the physical drive is potentially dangerous; an incorrect write to the drive could make the disk's contents inacces- sible by the operating system's file system.
Serial port	CreateFile (pszName is "COMx").
Parallel port	CreateFile (pszName is "LPTx").
Mailslot server	CreateMailslot (pszName is "\\.\mailslot\ <i>mailslotname</i> ").
Mailslot client	CreateFile (pszName is "\\servername\mailslot\mailslotname").
Named pipe server	CreateNamedPipe (pszName is "\\.\pipe\ <i>pipename</i> ").
Named pipe client	CreateFile (pszName is "\\servername\pipe\pipename").
Anonymous pipe	CreatePipe client and server.
Socket	socket, accept, or AcceptEx.
Console	CreateConsoleScreenBuffer or GetStdHandle.

Table 10-2 Functions for Opening Various Devices

Each function in Table 10-2 returns a handle that identifies the device. You can pass the handle to various functions to communicate with the device. For example, you call **SetCommConfig** to set the baud rate of a serial port:

```
BOOL SetCommConfig(
HANDLE hCommDev,
LPCOMMCONFIG pCC,
DWORD dwSize);
```

And you use **SetMailslotInfo** to set the time-out value when waiting to read data:

```
BOOL SetMailslotInfo(
HANDLE hMailslot,
DWORD dwReadTimeout);
```

As you can see, these functions require a handle to a device for their first argument.

When you are finished manipulating a device, you must close it. For most devices, you do this by calling the very popular **CloseHandle** function:

```
BOOL CloseHandle(HANDLE hObject);
```

However, if the device is a socket, you must call **closesocket** instead:

```
int closesocket(SOCKET s);
```

Also, if you have a handle to a device, you can find out what type of device it is by calling **GetFileType**:

DWORD GetFileType(HANDLE hDevice);

All you do is pass to the **GetFileType** function the handle to a device, and the function returns one of the values listed in Table 10-3.

Table 10-3 Values Returned by the GetFileType Function

Value	Description
FILE_TYPE_UNKNOWN	The type of the specified file is unknown.
FILE_TYPE_DISK	The specified file is a disk file.
FILE_TYPE_CHAR	The specified file is a character file, typically an LPT device or a console.
FILE_TYPE_PIPE	The specified file is either a named pipe or an anonymous pipe.

A Detailed Look at CreateFile

The **CreateFile** function, of course, creates and opens disk files, but don't let the name fool you it opens lots of other devices as well:

```
HANDLE CreateFile(
    PCTSTR pszName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hFileTemplate);
```

As you can see, **CreateFile** requires quite a few parameters, allowing for a great deal of flexibility when opening a device. At this point, I'll discuss all these parameters in detail.

When you call **CreateFile**, the **pszName** parameter identifies the device type as well as a specific instance of the device.

The **dwDesiredAccess** parameter specifies how you want to transmit data to and from the device. You can pass these four generic values, which are described in Table 10-4. Certain devices allow for additional access control flags. For example, when opening a file, you can specify access flags such as **FILE_READ_ATTRIBUTES**. See the Platform SDK documentation for more information about these flags.

Value	Meaning
0	You do not intend to read or write data to the device. Pass 0 when you just want to change the device's configuration settings—for example, if you want to change only a file's time stamp.
GENERIC_READ	Allows read-only access from the device.
GENERIC_WRITE	Allows write-only access to the device. For example, this value can be used to send data to a printer and by backup software. Note that GENERIC_WRITE does not imply GENERIC_READ .
GENERIC_READ GENERIC_WRITE	Allows both read and write access to the device. This value is the most common because it allows the free exchange of data.

Table 10-4 Generic Values That Can Be Passed for CreateFile's dwDesiredAccess Parameter

The **dwShareMode** parameter specifies device-sharing privileges. It controls how the device can be opened by additional calls to **CreateFile** while you still have the device opened yourself (that is, you haven't closed the device yet by calling **CloseHandle**). Table 10-5 describes the possible values that can be passed for the **dwShareMode** parameter.

Value	Meaning
0	You require exclusive access to the device. If the device is already opened, your call to CreateFile fails. If you successfully open the device, future calls to CreateFile fail.
FILE_SHARE_READ	You require that the data maintained by the device can't be changed by any other kernel object referring to this device. If the device is already opened for write or exclusive access, your call to CreateFile fails. If you successfully open the device, future calls to CreateFile fail if GENERIC_WRITE access is requested.
FILE_SHARE_WRITE	You require that the data maintained by the device can't be read by any other kernel object referring to this device. If the device is already opened for read or exclusive access, your call to CreateFile fails. If you successfully open the device, future calls to CreateFile fail if GENERIC_READ access is requested.
FILE_SHARE_READ FILE_SHARE_WRITE	You don't care if the data maintained by the device is read or written to by any other kernel object referring to this device. If the device is already opened for exclusive access, your call to CreateFile fails. If you successfully open the device, future calls to CreateFile fail when exclusive read, exclusive write, or exclusive read/write access is requested.
FILE_SHARE_DELETE	You don't care if the file is logically deleted or moved while you are working with the file. Internally, Windows marks a file for deletion and deletes it when all open handles to the file are closed.

 Table 10-5
 Values Related to I/O That Can Be Passed for CreateFile's dwShareMode Parameter



Note If you are opening a file, you can pass a pathname that is up to **MAX_PATH** (defined as 260 in WinDef.h) characters long. However, you can transcend this limit by calling **CreateFileW** (the Unicode version of **CreateFile**) and precede the pathname with "\\?\". Calling **CreateFileW** removes the prefix and allows you to pass a path that is almost 32,000 Unicode characters long. Remember, however, that you must use fully qualified paths when using this prefix; the system does not process relative directories such as "." and "...". Also, each individual component of the path is still limited to **MAX_PATH** characters. Don't be surprised to also see the **_MAX_PATH** constant in various source code because this is what C/C++ standard libraries define in stdlib.h as 260.

The **psa** parameter points to a **SECURITY_ATTRIBUTES** structure that allows you to specify security information and whether or not you'd like **CreateFile**'s returned handle to be inheritable. The security descriptor inside this structure is used only if you are creating a file on a secure file system such as NTFS; the security descriptor is ignored in all other cases. Usually, you just pass **NULL** for the **psa** parameter, indicating that the file is created with default security and that the returned handle is noninheritable.

The **dwCreationDisposition** parameter is most meaningful when **CreateFile** is being called to open a file as opposed to another type of device. Table 10-6 lists the possible values that you can pass for this parameter.

Value	Meaning
CREATE_NEW	Tells CreateFile to create a new file and to fail if a file with the same name already exists.
CREATE_ALWAYS	Tells CreateFile to create a new file regardless of whether a file with the same name already exists. If a file with the same name already exists, CreateFile overwrites the existing file.
OPEN_EXISTING	Tells CreateFile to open an existing file or device and to fail if the file or device doesn't exist.
OPEN_ALWAYS	Tells CreateFile to open the file if it exists and to create a new file if it doesn't exist.
TRUNCATE_EXISTING	Tells CreateFile to open an existing file, truncate its size to 0 bytes, and fail if the file doesn't already exist.

Table 10-6 Values That Can Be Passed for CreateFile's dwCreationDisposition Parameter



Note When you are calling **CreateFile** to open a device other than a file, you must pass **OPEN_EXISTING** for the **dwCreationDisposition** parameter.

CreateFile's **dwFlagsAndAttributes** parameter has two purposes: it allows you to set flags that fine-tune the communication with the device, and if the device is a file, you also get to set the file's attributes. Most of these communication flags are signals that tell the system how you intend to access the device. The system can then optimize its caching algorithms to help your application work more efficiently. I'll describe the communication flags first and then discuss the file attributes.

CreateFile Cache Flags

This section describes the various **CreateFile** cache flags, focusing on file system objects. For other kernel objects such as mailslots, you should refer to the MSDN documentation to get more specific details.

FILE_FLAG_NO_BUFFERING This flag indicates not to use any data buffering when accessing a file. To improve performance, the system caches data to and from disk drives. Normally, you do not specify this flag, and the cache manager keeps recently accessed portions of the file system in memory. This way, if you read a couple of bytes from a file and then read a few more bytes, the file's data is most likely loaded in memory and the disk has to be accessed only once instead of twice, greatly improving performance. However, this process does mean that portions of the file's data are in memory twice: the cache manager has a buffer, and you called some function (such as **ReadFile**) that copied some of the data from the cache manager's buffer into your own buffer.

When the cache manager is buffering data, it might also read ahead so that the next bytes you're likely to read are already in memory. Again, speed is improved by reading more bytes than necessary from the file. Memory is potentially wasted if you never attempt to read further in the file. (See the **FILE_FLAG_SEQUENTIAL_SCAN** and **FILE_FLAG_RANDOM_ACCESS** flags, discussed next, for more about reading ahead.)

By specifying the **FILE_FLAG_NO_BUFFERING** flag, you tell the cache manager that you do not want it to buffer any data—you take on this responsibility yourself! Depending on what you're doing, this flag can improve your application's speed and memory usage. Because the file system's device driver is writing the file's data directly into the buffers that you supply, you must follow certain rules:

- You must always access the file by using offsets that are exact multiples of the disk volume's sector size. (Use the GetDiskFreeSpace function to determine the disk volume's sector size.)
- You must always read/write a number of bytes that is an exact multiple of the sector size.
- You must make sure that the buffer in your process' address space begins on an address that is integrally divisible by the sector size.

FILE_FLAG_SEQUENTIAL_SCAN and FILE_FLAG_RANDOM_ACCESS These flags are useful only if you allow the system to buffer the file data for you. If you specify the **FILE_FLAG_NO_BUFFERING** flag, both of these flags are ignored.

If you specify the **FILE_FLAG_SEQUENTIAL_SCAN** flag, the system thinks you are accessing the file sequentially. When you read some data from the file, the system actually reads more of the file's data than the amount you requested. This process reduces the number of hits to the hard disk and improves the speed of your application. If you perform any direct seeks on the file, the system has spent a little extra time and memory caching data that you are not accessing. This is perfectly OK, but if you do it often, you'd be better off specifying the **FILE_FLAG_RANDOM_ACCESS** flag. This flag tells the system not to pre-read file data.

To manage a file, the cache manager must maintain some internal data structures for the file—the larger the file, the more data structures required. When working with extremely large files, the cache manager might not be able to allocate the internal data structures it requires and will fail to

296 Windows via C/C++

open the file. To access extremely large files, you must open the file using the **FILE_FLAG_NO_BUFFERING** flag.

FILE_FLAG_WRITE_THROUGH This is the last cache-related flag. It disables intermediate caching of file-write operations to reduce the potential for data loss. When you specify this flag, the system writes all file modifications directly to the disk. However, the system still maintains an internal cache of the file's data, and file-read operations use the cached data (if available) instead of reading data directly from the disk. When this flag is used to open a file on a network server, the Windows file-write functions do not return to the calling thread until the data is written to the server's disk drive.

That's it for the buffer-related communication flags. Now let's discuss the remaining communication flags.

Miscellaneous CreateFile Flags

This section describes the other flags that exist to customize **CreateFile** behaviors outside of caching.

FILE_FLAG_DELETE_ON_CLOSE Use this flag to have the file system delete the file after all handles to it are closed. This flag is most frequently used with the **FILE_ATTRIBUTE_TEMPORARY** attribute. When these two flags are used together, your application can create a temporary file, write to it, read from it, and close it. When the file is closed, the system automatically deletes the file—what a convenience!

FILE_FLAG_BACKUP_SEMANTICS Use this flag in backup and restore software. Before opening or creating any files, the system normally performs security checks to be sure that the process trying to open or create a file has the requisite access privileges. However, backup and restore software is special in that it can override certain file security checks. When you specify the **FILE_FLAG_BACKUP_SEMANTICS** flag, the system checks the caller's access token to see whether the Backup/ Restore File and Directories privileges are enabled. If the appropriate privileges are enabled, the system allows the file to be opened. You can also use the **FILE_FLAG_BACKUP_SEMANTICS** flag to open a handle to a directory.

FILE_FLAG_POSIX_SEMANTICS In Windows, filenames are case-preserved, whereas filename searches are case-insensitive. However, the POSIX subsystem requires that filename searches be case-sensitive. The **FILE_FLAG_POSIX_SEMANTICS** flag causes **CreateFile** to use a case-sensitive filename search when creating or opening a file. Use the **FILE_FLAG_POSIX_SEMANTICS** flag with extreme caution—if you use it when you create a file, that file might not be accessible to Windows applications.

FILE_FLAG_OPEN_REPARSE_POINT In my opinion, this flag should have been called **FILE_FLAG_ IGNORE_REPARSE_POINT** because it tells the system to ignore the file's reparse attribute (if it exists). Reparse attributes allow a file system filter to modify the behavior of opening, reading, writing, and closing a file. Usually, the modified behavior is desired, so using the **FILE_FLAG_OPEN_ REPARSE_POINT** flag is not recommended.

FILE_FLAG_OPEN_NO_RECALL This flag tells the system not to restore a file's contents from offline storage (such as tape) back to online storage (such as a hard disk). When files are not accessed for long periods of time, the system can transfer the file's contents to offline storage, freeing up hard disk space. When the system does this, the file on the hard disk is not destroyed; only the data in the file is destroyed. When the file is opened, the system automatically restores the data

from offline storage. The **FILE_FLAG_OPEN_NO_RECALL** flag instructs the system not to restore the data and causes I/O operations to be performed against the offline storage medium.

FILE_FLAG_OVERLAPPED This flag tells the system that you want to access a device asynchronously. You'll notice that the default way of opening a device is synchronous I/O (not specifying **FILE_FLAG_OVERLAPPED**). Synchronous I/O is what most developers are used to. When you read data from a file, your thread is suspended, waiting for the information to be read. Once the information has been read, the thread regains control and continues executing.

Because device I/O is slow when compared with most other operations, you might want to consider communicating with some devices asynchronously. Here's how it works: Basically, you call a function to tell the operating system to read or write data, but instead of waiting for the I/O to complete, your call returns immediately, and the operating system completes the I/O on your behalf using its own threads. When the operating system has finished performing your requested I/O, you can be notified. Asynchronous I/O is the key to creating high-performance, scalable, responsive, and robust applications. Windows offers several methods of asynchronous I/O, all of which are discussed in this chapter.

File Attribute Flags

Now it's time to examine the attribute flags for **CreateFile**'s **dwFlagsAndAttributes** parameter, described in Table 10-7. These flags are completely ignored by the system unless you are creating a brand new file and you pass **NULL** for **CreateFile**'s **hFileTemplate** parameter. Most of the attributes should already be familiar to you.

Flag	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this flag to mark files for backup or removal. When Create File creates a new file, this flag is automatically set.
FILE_ATTRIBUTE_ENCRYPTED	The file is encrypted.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It won't be included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only when it's used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The file exists, but its data has been moved to offline storage. This flag is useful for hierarchical storage systems.
FILE_ATTRIBUTE_READONLY	The file is read-only. Applications can read the file but can't write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of the operating system or is used exclusively by the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file's data will be used only for a short time. The file system tries to keep the file's data in RAM rather than on disk to keep the access time to a minimum.

Table 10-7File Attribute Flags That Can Be Passed for CreateFile's dwFlagsAndAttributesParameter

298 Windows via C/C++

Use **FILE_ATTRIBUTE_TEMPORARY** if you are creating a temporary file. When **CreateFile** creates a file with the temporary attribute, **CreateFile** tries to keep the file's data in memory instead of on the disk. This makes accessing the file's contents much faster. If you keep writing to the file and the system can no longer keep the data in RAM, the operating system will be forced to start writing the data to the hard disk. You can improve the system's performance by combining the **FILE_ATTRIBUTE_TEMPORARY** flag with the **FILE_FLAG_DELETE_ON_CLOSE** flag (discussed earlier). Normally, the system flushes a file's cached data when the file is closed. However, if the system sees that the file is to be deleted when it is closed, the system doesn't need to flush the file's cached data.

In addition to all these communication and attribute flags, a number of flags allow you to control the security quality of service when opening a named-pipe device. Because these flags are specific to named pipes only, I will not discuss them here. To learn about them, please read about the **CreateFile** function in the Platform SDK documentation.

CreateFile's last parameter, **hFileTemplate**, identifies the handle of an open file or is **NULL**. If **hFileTemplate** identifies a file handle, **CreateFile** ignores the attribute flags in the **dwFlagsAndAttributes** parameter completely and uses the attributes associated with the file identified by **hFileTemplate**. The file identified by **hFileTemplate** must have been opened with the **GENERIC_READ** flag for this to work. If **CreateFile** is opening an existing file (as opposed to creating a new file), the **hFileTemplate** parameter is ignored.

If **CreateFile** succeeds in creating or opening a file or device, the handle of the file or device is returned. If **CreateFile** fails, **INVALID_HANDLE_VALUE** is returned.

Note Most Windows functions that return a handle return **NULL** when the function fails. However, **CreateFile** returns **INVALID_HANDLE_VALUE** (defined as -1) instead. I have often seen code like this, which is incorrect:

```
HANDLE hFile = CreateFile(...);
if (hFile == NULL) {
    // We'll never get in here
} else {
    // File might or might not be created OK
}
Here's the correct way to check for an invalid file handle:
HANDLE hFile = CreateFile(...);
if (hFile == INVALID_HANDLE_VALUE) {
    // File not created
} else {
    // File created OK
}
```

Working with File Devices

Because working with files is so common, I want to spend some time addressing issues that apply specifically to file devices. This section shows how to position a file's pointer and change a file's size.

The first issue you must be aware of is that Windows was designed to work with extremely large files. Instead of representing a file's size using 32-bit values, the original Microsoft designers chose to use 64-bit values. This means that theoretically a file can reach a size of 16 EB (exabytes).

Dealing with 64-bit values in a 32-bit operating system makes working with files a little unpleasant because a lot of Windows functions require you to pass a 64-bit value as two separate 32-bit values. But as you'll see, working with the values is not too difficult and, in normal day-to-day operations, you probably won't need to work with a file greater than 4 GB. This means that the high 32 bits of the file's 64-bit size will frequently be 0 anyway.

Getting a File's Size

When working with files, quite often you will need to acquire the file's size. The easiest way to do this is by calling **GetFileSizeEx**:

```
BOOL GetFileSizeEx(
HANDLE hFile,
PLARGE_INTEGER pliFileSize);
```

The first parameter, **hFile**, is the handle of an opened file, and the **pliFileSize** parameter is the address of a **LARGE_INTEGER** union. This union allows a 64-bit signed value to be referenced as two 32-bit values or as a single 64-bit value, and it can be quite convenient when working with file sizes and offsets. Here is (basically) what the union looks like:

```
typedef union _LARGE_INTEGER {
   struct {
     DWORD LowPart; // Low 32-bit unsigned value
     LONG HighPart; // High 32-bit signed value
   };
   LONGLONG QuadPart; // Full 64-bit signed value
} LARGE_INTEGER, *PLARGE_INTEGER;
```

In addition to **LARGE_INTEGER**, there is a **ULARGE_INTEGER** structure representing an unsigned 64-bit value:

```
typedef union _ULARGE_INTEGER {
   struct {
     DWORD LowPart; // Low 32-bit unsigned value
     DWORD HighPart; // High 32-bit unsigned value
   };
   ULONGLONG QuadPart; // Full 64-bit unsigned value
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Another very useful function for getting a file's size is GetCompressedFileSize:

```
DWORD GetCompressedFileSize(
PCTSTR pszFileName,
PDWORD pdwFileSizeHigh);
```

This function returns the file's physical size, whereas **GetFileSizeEx** returns the file's logical size. For example, consider a 100-KB file that has been compressed to occupy 85 KB. Calling **GetFile SizeEx** returns the logical size of the file–100 KB–whereas **GetCompressedFileSize** returns the actual number of bytes on disk occupied by the file–85 KB.

Unlike **GetFileSizeEx**, **GetCompressedFileSize** takes a filename passed as a string instead of taking a handle for the first parameter. The **GetCompressedFileSize** function returns the 64-bit size of the file in an unusual way: the low 32 bits of the file's size are the function's return value. The high 32 bits of the file's size are placed in the **DWORD** pointed to by the **pdwFileSizeHigh** parameter. Here the use of the **ULARGE_INTEGER** structure comes in handy:

```
ULARGE_INTEGER ulFileSize;
ulFileSize.LowPart = GetCompressedFileSize(TEXT("SomeFile.dat"),
    &ulFileSize.HighPart);
```

// 64-bit file size is now in ulFileSize.QuadPart

Positioning a File Pointer

Calling **CreateFile** causes the system to create a file kernel object that manages operations on the file. Inside this kernel object is a file pointer. This file pointer indicates the 64-bit offset within the file where the next synchronous read or write should be performed. Initially, this file pointer is set to 0, so if you call **ReadFile** immediately after a call to **CreateFile**, you will start reading the file from offset 0. If you read 10 bytes of the file into memory, the system updates the pointer associated with the file handle so that the next call to **ReadFile** starts reading at the eleventh byte in the file at offset 10. For example, look at this code, in which the first 10 bytes from the file are read into the buffer, and then the next 10 bytes are read into the buffer:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
ReadFile(hFile, pb, 10, &dwNumBytes, NULL); // Reads bytes 0 - 9
ReadFile(hFile, pb, 10, &dwNumBytes, NULL); // Reads bytes 10 - 19
```

Because each file kernel object has its own file pointer, opening the same file twice gives slightly different results:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
HANDLE hFile2 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL); // Reads bytes 0 - 9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL); // Reads bytes 0 - 9
```

In this example, two different kernel objects manage the same file. Because each kernel object has its own file pointer, manipulating the file with one file object has no effect on the file pointer maintained by the other object, and the first 10 bytes of the file are read twice.

I think one more example will help make all this clear:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), ...); // Pointer set to 0
HANDLE hFile2;
DuplicateHandle(
   GetCurrentProcess(), hFile1,
   GetCurrentProcess(), &hFile2,
   0, FALSE, DUPLICATE_SAME_ACCESS);
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL); // Reads bytes 0 - 9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL); // Reads bytes 10 - 19
```

In this example, one file kernel object is referenced by two file handles. Regardless of which handle is used to manipulate the file, the one file pointer is updated. As in the first example, different bytes are read each time.

If you need to access a file randomly, you will need to alter the file pointer associated with the file's kernel object. You do this by calling **SetFilePointerEx**:

```
BOOL SetFilePointerEx(
HANDLE hFile,
LARGE_INTEGER liDistanceToMove,
PLARGE_INTEGER pliNewFilePointer,
DWORD dwMoveMethod);
```

The **hFile** parameter identifies the file kernel object whose file pointer you want to change. The **liDistanceToMove** parameter tells the system by how many bytes you want to move the pointer. The number you specify is added to the current value of the file's pointer, so a negative number has the effect of stepping backward in the file. The last parameter of **SetFilePointerEx**, **dwMoveMethod**, tells **SetFilePointerEx** how to interpret the **liDistanceToMove** parameter. Table 10-8 describes the three possible values you can pass via **dwMoveMethod** to specify the starting point for the move.

Value	Meaning
FILE_BEGIN	The file object's file pointer is set to the value specified by the liDistanceToMove parameter. Note that liDistanceToMove is interpreted as an unsigned 64-bit value.
FILE_CURRENT	The file object's file pointer has the value of liDistanceToMove added to it. Note that liDistanceToMove is interpreted as a signed 64-bit value, allowing you to seek backward in the file.
FILE_END	The file object's file pointer is set to the logical file size plus the liDistanceToMove parameter. Note that liDistanceToMove is inter- preted as a signed 64-bit value, allowing you to seek backward in the file.

Table 10-8 Values That Can Be Passed for SetFilePointerEx's dwMoveMethod Parameter

After **SetFilePointerEx** has updated the file object's file pointer, the new value of the file pointer is returned in the **LARGE_INTEGER** pointed to by the **pliNewFilePointer** parameter. You can pass **NULL** for **pliNewFilePointer** if you're not interested in the new pointer value.

Here are a few facts to note about **SetFilePointerEx**:

- Setting a file's pointer beyond the end of the file's current size is legal. Doing so does not actually increase the size of the file on disk unless you write to the file at this position or call SetEndOfFile.
- When using **SetFilePointerEx** with a file opened with **FILE_FLAG_NO_BUFFERING**, the file pointer can be positioned only on sector-aligned boundaries. The FileCopy sample application later in this chapter demonstrates how to do this properly.
- Windows does not offer a GetFilePointerEx function, but you can use SetFile
 PointerEx to move the pointer by 0 bytes to get the desired effect, as shown in the following code snippet:

```
LARGE_INTEGER liCurrentPosition = { 0 };
SetFilePointerEx(hFile, liCurrentPosition, &liCurrentPosition, FILE_CURRENT);
```

Setting the End of a File

Usually, the system takes care of setting the end of a file when the file is closed. However, you might sometimes want to force a file to be smaller or larger. On those occasions, call

```
BOOL SetEndOfFile(HANDLE hFile);
```

This **SetEndOfFile** function truncates or extends a file's size to the size indicated by the file object's file pointer. For example, if you wanted to force a file to be 1024 bytes long, you'd use **SetEndOfFile** this way:

```
HANDLE hFile = CreateFile(...);
LARGE_INTEGER liDistanceToMove;
liDistanceToMove.QuadPart = 1024;
SetFilePointerEx(hFile, liDistanceToMove, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);
```

Using Windows Explorer to examine the properties of this file reveals that the file is exactly 1024 bytes long.

Performing Synchronous Device I/O

This section discusses the Windows functions that allow you to perform synchronous device I/O. Keep in mind that a device can be a file, mailslot, pipe, socket, and so on. No matter which device is used, the I/O is performed using the same functions.

Without a doubt, the easiest and most commonly used functions for reading from and writing to devices are **ReadFile** and **WriteFile**:

```
BOOL ReadFile(
  HANDLE
              hFile.
  PVOID
               pvBuffer.
  DWORD
              nNumBytesToRead,
   PDWORD
              pdwNumBytes,
  OVERLAPPED* pOverlapped);
BOOL WriteFile(
  HANDLE
              hFile,
  CONST VOID *pvBuffer,
  DWORD
              nNumBytesToWrite,
   PDWORD
              pdwNumBytes,
  OVERLAPPED* pOverlapped);
```

The **hFile** parameter identifies the handle of the device you want to access. When the device is opened, you must not specify the **FILE_FLAG_OVERLAPPED** flag, or the system will think that you want to perform asynchronous I/O with the device. The **pvBuffer** parameter points to the buffer to which the device's data should be read or to the buffer containing the data that should be written to the device. The **nNumBytesToRead** and **nNumBytesToWrite** parameters tell **ReadFile** and **WriteFile** how many bytes to read from the device and how many bytes to write to the device, respectively.

The **pdwNumBytes** parameters indicate the address of a **DWORD** that the functions fill with the number of bytes successfully transmitted to and from the device. The last parameter, **pOverlapped**, should be **NULL** when performing synchronous I/O. You'll examine this parameter in more detail shortly when asynchronous I/O is discussed.

Both **ReadFile** and **WriteFile** return **TRUE** if successful. By the way, **ReadFile** can be called only for devices that were opened with the **GENERIC_READ** flag. Likewise, **WriteFile** can be called only when the device is opened with the **GENERIC_WRITE** flag.

Flushing Data to the Device

Remember from our look at the **CreateFile** function that you can pass quite a few flags to alter the way in which the system caches file data. Some other devices, such as serial ports, mailslots, and pipes, also cache data. If you want to force the system to write cached data to the device, you can call **FlushFileBuffers**:

```
BOOL FlushFileBuffers(HANDLE hFile);
```

The **FlushFileBuffers** function forces all the buffered data associated with a device that is identified by the **hFile** parameter to be written. For this to work, the device has to be opened with the **GENERIC_WRITE** flag. If the function is successful, **TRUE** is returned.

Synchronous I/O Cancellation

Functions that do synchronous I/O are easy to use, but they block any other operations from occurring on the thread that issued the I/O until the request is completed. A great example of this is a **CreateFile** operation. When a user performs mouse and keyboard input, window messages are inserted into a queue that is associated with the thread that created the window that the input is destined for. If that thread is stuck inside a call to **CreateFile**, waiting for **CreateFile** to

304 Windows via C/C++

return, the window messages are not getting processed and all the windows created by the thread are frozen. The most common reason why applications hang is because their threads are stuck waiting for synchronous I/O operations to complete!

With Windows Vista, Microsoft has added some big features in an effort to alleviate this problem. For example, if a console (CUI) application hangs because of synchronous I/O, the user is now able to hit Ctrl+C to gain control back and continue using the console; the user no longer has to kill the console process. Also, the new Vista file open/save dialog box allows the user to press the Cancel button when opening a file is taking an excessively long time (typically, as a result of attempting to access a file on a network server).

To build a responsive application, you should try to perform asynchronous I/O operations as much as possible. This typically also allows you to use very few threads in your application, thereby saving resources (such as thread kernel objects and stacks). Also, it is usually easy to offer your users the ability to cancel an operation when you initiate it asynchronously. For example, Internet Explorer allows the user to cancel (via a red X button or the Esc key) a Web request if it is taking too long and the user is impatient.

Unfortunately, certain Windows APIs, such as **CreateFile**, offer no way to call the methods asynchronously. Although some of these methods do ultimately time out if they wait too long (such as when attempting to access a network server), it would be best if there was an application programming interface (API) that you could call to force the thread to abort waiting and to just cancel the synchronous I/O operation. In Windows Vista, the following function allows you to cancel a pending synchronous I/O request for a given thread:

BOOL CancelSynchronousIo(HANDLE hThread);

The **hThread** parameter is a handle of the thread that is suspended waiting for the synchronous I/O request to complete. This handle must have been created with the **THREAD_TERMINATE** access. If this is not the case, **CancelSynchronousIo** fails and **GetLastError** returns **ERROR_ACCESS_DENIED**. When you create the thread yourself by using **CreateThread** or **_beginthreadex**, the returned handle has **THREAD_ALL_ACCESS**, which includes **THREAD_TERMINATE** access. However, if you are taking advantage of the thread pool or your cancellation code is called by a timer callback, you usually have to call **OpenThread** to get a thread handle corresponding to the current thread ID; don't forget to pass **THREAD_TERMINATE** as the first parameter.

If the specified thread was suspended waiting for a synchronous I/O operation to complete, CancelSynchronousIo wakes the suspended thread and the operation it was trying to perform returns failure; calling GetLastError returns ERROR_OPERATION_ABORTED. Also, Cancel SynchronousIo returns TRUE to its caller.

Note that the thread calling **CancelSynchronousIo** doesn't really know where the thread that called the synchronous operation is. The thread could have been pre-empted and it has yet to actually communicate with the device; it could be suspended, waiting for the device to respond; or the device could have just responded, and the thread is in the process of returning from its call. If **CancelSynchronousIo** is called when the specified thread is not actually suspended waiting for the device to respond, **CancelSynchronousIo** returns **FALSE** and **GetLastError** returns **ERROR_NOT_FOUND**.

For this reason, you might want to use some additional thread synchronization (as discussed in Chapter 8, "Thread Synchronization in User Mode," and Chapter 9, "Thread Synchronization with

Kernel Objects") to know for sure whether you are cancelling a synchronous operation or not. However, in practice, this is usually not necessary, as it is typical for a user to initiate the cancellation and this usually happens because the user sees the application is suspended. Also, a user could try to initiate cancellation twice (or more) if the first attempt to cancel doesn't seem to work. By the way, Windows calls **CancelSynchronousIo** internally to allow the user to regain control of a command console and the file open/save dialog box.



Caution Cancellation of I/O requests depends on the driver implementing the corresponding system layer. It might happen that such a driver does not support cancellation. In that case, **CancelSynchronousIo** would have returned **TRUE** anyway because this function has found a request to be marked as being cancelled. The real cancellation of the request is left as the responsibility of the driver. An example of a driver that was updated to support synchronous cancellation for Windows Vista is the network redirector.

Basics of Asynchronous Device I/O

Compared to most other operations carried out by a computer, device I/O is one of the slowest and most unpredictable. The CPU performs arithmetic operations and even paints the screen much faster than it reads data from or writes data to a file or across a network. However, using asynchronous device I/O enables you to better use resources and thus create more efficient applications.

Consider a thread that issues an asynchronous I/O request to a device. This I/O request is passed to a device driver, which assumes the responsibility of actually performing the I/O. While the device driver waits for the device to respond, the application's thread is not suspended as it waits for the I/O request to complete. Instead, this thread continues executing and performs other useful tasks.

At some point, the device driver finishes processing the queued I/O request and must notify the application that data has been sent, data has been received, or an error has occurred. You'll learn how the device driver notifies you of I/O completions in "Receiving Completed I/O Request Notifications" on page 310. For now, let's concentrate on how to queue asynchronous I/O requests. Queuing asynchronous I/O requests is the essence of designing a high-performance, scalable application, and it is what the remainder of this chapter is all about.

To access a device asynchronously, you must first open the device by calling **CreateFile**, specifying the **FILE_FLAG_OVERLAPPED** flag in the **dwFlagsAndAttributes** parameter. This flag notifies the system that you intend to access the device asynchronously.

To queue an I/O request for a device driver, you use the **ReadFile** and **WriteFile** functions that you already learned about in "Performing Synchronous Device I/O" on page 302. For convenience, I'll list the function prototypes again:

BOOL ReadFile(HANDLE hFile,

PVOID	p∨Buffer,
DWORD	nNumBytesToRead
PDWORD	pdwNumBytes,
OVERLAPPED*	pOverlapped);

```
BOOL WriteFile(
HANDLE hFile,
CONST VOID *pvBuffer,
DWORD nNumBytesToWrite,
PDWORD pdwNumBytes,
OVERLAPPED* pOverlapped);
```

When either of these functions is called, the function checks to see if the device, identified by the **hFile** parameter, was opened with the **FILE_FLAG_OVERLAPPED** flag. If this flag is specified, the function performs asynchronous device I/O. By the way, when calling either function for asynchronous I/O, you can (and usually do) pass **NULL** for the **pdwNumBytes** parameter. After all, you expect these functions to return before the I/O request has completed, so examining the number of bytes transferred is meaningless at this time.

The OVERLAPPED Structure

When performing asynchronous device I/O, you must pass the address to an initialized **OVER LAPPED** structure via the **pOver1apped** parameter. The word "overlapped" in this context means that the time spent performing the I/O request overlaps the time your thread spends performing other tasks. Here's what an **OVERLAPPED** structure looks like:

```
typedef struct _OVERLAPPED {
   DWORD Internal; // [out] Error code
   DWORD InternalHigh; // [out] Number of bytes transferred
   DWORD Offset; // [in] Low 32-bit file offset
   DWORD OffsetHigh; // [in] High 32-bit file offset
   HANDLE hEvent; // [in] Event handle or data
} OVERLAPPED, *LPOVERLAPPED;
```

This structure contains five members. Three of these members–**Offset**, **OffsetHigh**, and **hEvent**—must be initialized prior to calling **ReadFile** or **WriteFile**. The other two members, **Internal** and **InternalHigh**, are set by the device driver and can be examined when the I/O operation completes. Here is a more detailed explanation of these member variables:

Offset and OffsetHigh When a file is being accessed, these members indicate the 64-bit offset in the file where you want the I/O operation to begin. Recall that each file kernel object has a file pointer associated with it. When issuing a synchronous I/O request, the system knows to start accessing the file at the location identified by the file pointer. After the operation is complete, the system updates the file pointer automatically so that the next operation can pick up where the last operation left off.

When performing asynchronous I/O, this file pointer is ignored by the system. Imagine what would happen if your code placed two asynchronous calls to **ReadFile** (for the same file kernel object). In this scenario, the system wouldn't know where to start reading for the second call to **ReadFile**. You probably wouldn't want to start reading the file at the same location used by the first call to **ReadFile**. You might want to start the second read at the byte in the file that followed the last byte that was read by the first call to **ReadFile**. To avoid the confusion of multiple asynchronous calls to the same object, all asynchronous I/O requests must specify the starting file offset in the **OVERLAPPED** structure.

Note that the **Offset** and **OffsetHigh** members are not ignored for nonfile devices—you must initialize both members to **0** or the I/O request will fail and **GetLastError** will return **ERROR_INVALID_PARAMETER**.

- hEvent This member is used by one of the four methods available for receiving I/O completion notifications. When using the alertable I/O notification method, this member can be used for your own purposes. I know many developers who store the address of a C++ object in hEvent. (This member will be discussed more in "Signaling an Event Kernel Object" on page 312.)
- Internal This member holds the processed I/O's error code. As soon as you issue an asynchronous I/O request, the device driver sets Internal to STATUS_PENDING, indicating that no error has occurred because the operation has not started. In fact, the macro HasOverlappedIoCompleted, which is defined in WinBase.h, allows you to check whether an asynchronous I/O operation has completed. If the request is still pending, FALSE is returned; if the I/O request is completed, TRUE is returned. Here is the macro's definition:

```
#define HasOverlappedIoCompleted(pOverlapped) \
    ((pOverlapped)->Internal != STATUS_PENDING)
```

InternalHigh When an asynchronous I/O request completes, this member holds the number of bytes transferred.

When first designing the **OVERLAPPED** structure, Microsoft decided not to document the **Internal** and **InternalHigh** members (which explains their names). As time went on, Microsoft realized that the information contained in these members would be useful to developers, so it documented them. However, Microsoft didn't change the names of the members because the operating system source code referenced them frequently, and Microsoft didn't want to modify the code.



Note When an asynchronous I/O request completes, you receive the address of the **OVERLAPPED** structure that was used when the request was initiated. Having more contextual information passed around with an **OVERLAPPED** structure is frequently useful—for example, if you wanted to store the handle of the device used to initiate the I/O request inside the **OVERLAPPED** structure. The **OVERLAPPED** structure doesn't offer a device handle member or other potentially useful members for storing context, but you can solve this problem quite easily.

I frequently create a C++ class that is derived from an **OVERLAPPED** structure. This C++ class can have any additional information in it that I want. When my application receives the address of an **OVERLAPPED** structure, I simply cast the address to a pointer of my C++ class. Now I have access to the **OVERLAPPED** members and any additional context information my application needs. The FileCopy sample application at the end of this chapter demonstrates this technique. See my **CIOReq** C++ class in the FileCopy sample application for the details.

Asynchronous Device I/O Caveats

You should be aware of a couple of issues when performing asynchronous I/O. First, the device driver doesn't have to process queued I/O requests in a first-in first-out (FIFO) fashion. For

example, if a thread executes the following code, the device driver will quite possibly write to the file and then read from the file:

```
OVERLAPPED o1 = { 0 };
OVERLAPPED o2 = { 0 };
BYTE bBuffer[100];
ReadFile (hFile, bBuffer, 100, NULL, &o1);
WriteFile(hFile, bBuffer, 100, NULL, &o2);
```

A device driver typically executes I/O requests out of order if doing so helps performance. For example, to reduce head movement and seek times, a file system driver might scan the queued I/O request list looking for requests that are near the same physical location on the hard drive.

The second issue you should be aware of is the proper way to perform error checking. Most Windows functions return **FALSE** to indicate failure or nonzero to indicate success. However, the **ReadFile** and **WriteFile** functions behave a little differently. An example might help to explain.

When attempting to queue an asynchronous I/O request, the device driver might choose to process the request synchronously. This can occur if you're reading from a file and the system checks whether the data you want is already in the system's cache. If the data is available, your I/O request is not queued to the device driver; instead, the system copies the data from the cache to your buffer, and the I/O operation is complete. The driver always performs certain operations synchronously, such as NTFS file compression, extending the length of a file or appending information to a file. For more information about operations that are always performed synchronously, please see *http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932*.

ReadFile and **WriteFile** return a nonzero value if the requested I/O was performed synchronously. If the requested I/O is executing asynchronously, or if an error occurred while calling **ReadFile** or **WriteFile**, **FALSE** is returned. When **FALSE** is returned, you must call **GetLast Error** to determine specifically what happened. If **GetLastError** returns **ERROR_IO_PENDING**, the I/O request was successfully queued and will complete later.

If **GetLastError** returns a value other than **ERROR_IO_PENDING**, the I/O request could not be queued to the device driver. Here are the most common error codes returned from **GetLastError** when an I/O request can't be queued to the device driver:

- **ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY** Each device driver maintains a fixed-size list (in a nonpaged pool) of outstanding I/O requests. If this list is full, the system can't queue your request, **ReadFile** and **WriteFile** return **FALSE**, and **GetLast Error** reports one of these two error codes (depending on the driver).
- ERROR_NOT_ENOUGH_QUOTA Certain devices require that your data buffer's storage be page locked so that the data cannot be swapped out of RAM while the I/O is pending. This page-locked storage requirement is certainly true of file I/O when using the FILE_FLAG_NO_
 BUFFERING flag. However, the system restricts the amount of storage that a single process can page lock. If ReadFile and WriteFile cannot page lock your buffer's storage, the functions return FALSE and GetLastError reports ERROR_NOT_ENOUGH_QUOTA. You can increase a process' quota by calling SetProcessWorkingSetSize.

How should you handle these errors? Basically, these errors occur because a number of outstanding I/O requests have not yet completed, so you need to allow some pending I/O requests to complete and then reissue the calls to **ReadFile** and **WriteFile**.

The third issue you should be aware of is that the data buffer and **OVERLAPPED** structure used to issue the asynchronous I/O request must not be moved or destroyed until the I/O request has completed. When queuing an I/O request to a device driver, the driver is passed the *address* of the data buffer and the *address* of the **OVERLAPPED** structure. Notice that just the address is passed, not the actual block. The reason for this should be quite obvious: memory copies are very expensive and waste a lot of CPU time.

When the device driver is ready to process your queued request, it transfers the data referenced by the **pvBuffer** address, and it accesses the file's offset member and other members contained within the **OVERLAPPED** structure pointed to by the **pOverlapped** parameter. Specifically, the device driver updates the **Internal** member with the I/O's error code and the **InternalHigh** member with the number of bytes transferred.



Note It is absolutely essential that these buffers not be moved or destroyed until the I/O request has completed; otherwise, memory will be corrupted. Also, you must allocate and initialize a unique **OVERLAPPED** structure for each I/O request.

The preceding note is very important and is one of the most common bugs developers introduce when implementing an asynchronous device I/O architecture. Here's an example of what *not* to do:

```
VOID ReadData(HANDLE hFile) {
   OVERLAPPED o = { 0 };
   BYTE b[100];
   ReadFile(hFile, b, 100, NULL, &o);
}
```

This code looks fairly harmless, and the call to **ReadFile** is perfect. The only problem is that the function returns after queuing the asynchronous I/O request. Returning from the function essentially frees the buffer and the **OVERLAPPED** structure from the thread's stack, but the device driver is not aware that **ReadData** returned. The device driver still has two memory addresses that point to the thread's stack. When the I/O completes, the device driver is going to modify memory on the thread's stack, corrupting whatever happens to be occupying that spot in memory at the time. This bug is particularly difficult to find because the memory modification occurs asynchronously. Sometimes the device driver might perform I/O synchronously, in which case you won't see the bug. Sometimes the I/O might complete right after the function returns, or it might complete over an hour later, and who knows what the stack is being used for then.

Canceling Queued Device I/O Requests

Sometimes you might want to cancel a queued device I/O request before the device driver has processed it. Windows offers a few ways to do this:

■ You can call **Cancel Io** to cancel all I/O requests queued by the calling thread for the specified handle (unless the handle has been associated with an I/O completion port):

```
BOOL CancelIo(HANDLE hFile);
```

- You can cancel all queued I/O requests, regardless of which thread queued the request, by closing the handle to a device itself.
- When a thread dies, the system automatically cancels all I/O requests issued by the thread, except for requests made to handles that have been associated with an I/O completion port.
- If you need to cancel a single, specific I/O request submitted on a given file handle, you can call **CancelIoEx**:

BOOL CancelIoEx(HANDLE hFile, LPOVERLAPPED pOverlapped);.

With **CancelIoEx**, you are able to cancel pending I/O requests emitted by a thread different from the calling thread. This function marks as canceled all I/O requests that are pending on **hFile** and associated with the given **pOverlapped** parameter. Because each outstanding I/O request should have its own **OVERLAPPED** structure, each call to **CancelIoEx** should cancel just one outstanding request. However, if the **pOverlapped** parameter is **NULL**, **CancelIoEx** cancels all outstanding I/O requests for the specified **hFile**.



Note Canceled I/O requests complete with an error code of **ERROR_OPERATION_ ABORTED**.

Receiving Completed I/O Request Notifications

At this point, you know how to queue an asynchronous device I/O request, but I haven't discussed how the device driver notifies you after the I/O request has completed.

Windows offers four different methods (briefly described in Table 10-9) for receiving I/O completion notifications, and this chapter covers all of them. The methods are shown in order of complexity, from the easiest to understand and implement (signaling a device kernel object) to the hardest to understand and implement (I/O completion ports).

Technique	Summary
Signaling a device kernel object	Not useful for performing multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Signaling an event kernel object	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it.
Using alertable I/O	Allows multiple simultaneous I/O requests against a single device. The thread that issued an I/O request must also process it.
Using I/O completion ports	Allows multiple simultaneous I/O requests against a single device. Allows one thread to issue an I/O request and another thread to process it. This technique is highly scalable and has the most flexibility.

Table 10-9 Methods for Receiving I/O Completion Notifications

As stated at the beginning of this chapter, the I/O completion port is the hands-down best method of the four for receiving I/O completion notifications. By studying all four, you'll learn why Microsoft added the I/O completion port to Windows and how the I/O completion port solves all the problems that exist for the other methods.

Signaling a Device Kernel Object

Once a thread issues an asynchronous I/O request, the thread continues executing, doing useful work. Eventually, the thread needs to synchronize with the completion of the I/O operation. In other words, you'll hit a point in your thread's code at which the thread can't continue to execute unless the data from the device is fully loaded into the buffer.

In Windows, a device kernel object can be used for thread synchronization, so the object can either be in a signaled or nonsignaled state. The **ReadFile** and **WriteFile** functions set the device kernel object to the nonsignaled state just before queuing the I/O request. When the device driver completes the request, the driver sets the device kernel object to the signaled state.

A thread can determine whether an asynchronous I/O request has completed by calling either **WaitForSingleObject** or **WaitForMultipleObjects**. Here is a simple example:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
BYTE bBuffer[100];
OVERLAPPED o = \{ 0 \};
o.Offset = 345:
BOOL bReadDone = ReadFile(hFile, bBuffer, 100, NULL, &o);
DWORD dwError = GetLastError();
if (!bReadDone && (dwError == ERROR_IO_PENDING)) {
   // The I/O is being performed asynchronously; wait for it to complete
  WaitForSingleObject(hFile, INFINITE);
  bReadDone = TRUE;
}
if (bReadDone) {
  // o.Internal contains the I/O error
  // o.InternalHigh contains the number of bytes transferred
  // bBuffer contains the read data
} else {
  // An error occurred; see dwError
}
```

This code issues an asynchronous I/O request and then immediately waits for the request to finish, defeating the purpose of asynchronous I/O! Obviously, you would never actually write code similar to this, but the code does demonstrate important concepts, which I'll summarize here:

- The device must be opened for asynchronous I/O by using the **FILE_FLAG_OVERLAPPED** flag.
- The **OVERLAPPED** structure must have its **Offset**, **OffsetHigh**, and **hEvent** members initialized. In the code example, I set them all to **0** except for **Offset**, which I set to **345** so that **ReadFile** reads data from the file starting at byte 346.
- ReadFile's return value is saved in bReadDone, which indicates whether the I/O request was performed synchronously.

- If the I/O request was not performed synchronously, I check to see whether an error occurred or whether the I/O is being performed asynchronously. Comparing the result of **GetLastError** with **ERROR_IO_PENDING** gives me this information.
- To wait for the data, I call **WaitForSingleObject**, passing the handle of the device kernel object. As you saw in Chapter 9, calling this function suspends the thread until the kernel object becomes signaled. The device driver signals the object when it completes the I/O. After **WaitForSingleObject** returns, the I/O is complete and I set **bReadDone** to **TRUE**.
- After the read completes, you can examine the data in **bBuffer**, the error code in the **OVERLAPPED** structure's **Internal** member, and the number of bytes transferred in the **OVERLAPPED** structure's **InternalHigh** member.
- If a true error occurred, **dwError** contains the error code giving more information.

Signaling an Event Kernel Object

The method for receiving I/O completion notifications just described is very simple and straightforward, but it turns out not to be all that useful because it does not handle multiple I/O requests well. For example, suppose you were trying to carry out multiple asynchronous operations on a single file at the same time. Say that you wanted to read 10 bytes from the file and write 10 bytes to the file simultaneously. The code might look like this:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
```

```
BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);
BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
...
WaitForSingleObject(hFile, INFINITE);
```

// We don't know what completed: Read? Write? Both?

You can't synchronize your thread by waiting for the device to become signaled because the object becomes signaled as soon as either of the operations completes. If you call **WaitForSingle Object**, passing it the device handle, you will be unsure whether the function returned because the read operation completed, the write operation completed, or both operations completed. Clearly, there needs to be a better way to perform multiple, simultaneous asynchronous I/O requests so that you don't run into this predicament—fortunately, there is.

The last member of the **OVERLAPPED** structure, **hEvent**, identifies an event kernel object. You must create this event object by calling **CreateEvent**. When an asynchronous I/O request completes, the device driver checks to see whether the **hEvent** member of the **OVERLAPPED** structure is **NULL**. If **hEvent** is not **NULL**, the driver signals the event by calling **SetEvent**. The driver also sets the device object to the signaled state just as it did before. However, if you are using events to determine when a device operation has completed, you shouldn't wait for the device object to become signaled—wait for the event instead.


Note To improve performance slightly, you can tell Windows not to signal the file object when the operation completes. You do so by calling the **SetFileCompletionNotifica tionModes** function:

BOOL SetFileCompletionNotificationModes(HANDLE hFile, UCHAR uFlags);

The **hFile** parameter identifies a file handle, and the **uFlags** parameter indicates how Windows should modify its normal behavior with respect to completing an I/O operation. If you pass the **FILE_SKIP_SET_EVENT_ON_HANDLE** flag, Windows will not signal the file handle when operations on the file complete. Note that the **FILE_SKIP_SET_EVENT_ ON_HANDLE** flag is very poorly named; a better name would have been something like **FILE_SKIP_SIGNAL**.

If you want to perform multiple asynchronous device I/O requests simultaneously, you must create a separate event object for each request, initialize the **hEvent** member in each request's **OVERLAPPED** structure, and then call **ReadFile** or **WriteFile**. When you reach the point in your code at which you need to synchronize with the completion of the I/O request, simply call **WaitForMultipleObjects**, passing in the event handles associated with each outstanding I/O request's **OVERLAPPED** structures. With this scheme, you can easily and reliably perform multiple asynchronous device I/O operations simultaneously and use the same device object. The following code demonstrates this approach:

```
HANDLE hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
BYTE bReadBuffer[10];
OVERLAPPED oRead = \{0\};
oRead.Offset = 0;
oRead.hEvent = CreateEvent(...);
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);
BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
oWrite.hEvent = CreateEvent(...);
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
. . .
HANDLE h[2];
h[0] = oRead.hEvent;
h[1] = oWrite.hEvent;
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw Đ WAIT_OBJECT_0) {
   case 0: // Read completed
     break:
   case 1: // Write completed
      break;
}
```

This code is somewhat contrived and is not *exactly* what you'd do in a real-life application, but it does illustrate my point. Typically, a real-life application has a loop that waits for I/O requests to complete. As each request completes, the thread performs the desired task, queues another asynchronous I/O request, and loops back around, waiting for more I/O requests to complete.

GetOverlappedResult

Recall that originally Microsoft was not going to document the **OVERLAPPED** structure's **Internal** and **InternalHigh** members, which meant it needed to provide another way for you to know how many bytes were transferred during the I/O processing and get the I/O's error code. To make this information available to you, Microsoft created the **GetOverlappedResult** function:

```
BOOL GetOverlappedResult(
HANDLE hFile,
OVERLAPPED* pOverlapped,
PDWORD pdwNumBytes,
BOOL bWait);
```

Microsoft now documents the **Internal** and **InternalHigh** members, so the **GetOverlapped Result** function is not very useful. However, when I was first learning asynchronous I/O, I decided to reverse engineer the function to help solidify concepts in my head. The following code shows how **GetOverlappedResult** is implemented internally:

```
BOOL GetOverlappedResult(
  HANDLE hFile,
  OVERLAPPED* po,
   PDWORD pdwNumBytes,
   BOOL bWait) {
   if (po->Internal == STATUS_PENDING) {
     DWORD dwWaitRet = WAIT_TIMEOUT;
     if (bWait) {
        // Wait for the I/O to complete
         dwWaitRet = WaitForSingleObject(
            (po->hEvent != NULL) ? po->hEvent : hFile, INFINITE);
     }
     if (dwWaitRet == WAIT_TIMEOUT) {
         // I/O not complete and we're not supposed to wait
         SetLastError(ERROR_IO_INCOMPLETE);
         return(FALSE);
     }
      if (dwWaitRet != WAIT_OBJECT_0) {
         // Error calling WaitForSingleObject
         return(FALSE);
     }
   }
   // I/O is complete; return number of bytes transferred
   *pdwNumBytes = po->InternalHigh;
   if (SUCCEEDED(po->Internal)) {
      return(TRUE); // No I/O error
   }
   // Set last error to I/O error
   SetLastError(po->Internal);
   return(FALSE);
}
```

Alertable I/O

The third method available to you for receiving I/O completion notifications is called *alertable I/O*. At first, Microsoft touted alertable I/O as the absolute best mechanism for developers who wanted to create high-performance, scalable applications. But as developers started using alertable I/O, they soon realized that it was not going to live up to the promise.

I have worked with alertable I/O quite a bit, and I'll be the first to tell you that alertable I/O is horrible and should be avoided. However, to make alertable I/O work, Microsoft added some infrastructure into the operating system that I have found to be extremely useful and valuable. As you read this section, concentrate on the infrastructure that is in place and don't get bogged down in the I/O aspects.

Whenever a thread is created, the system also creates a queue that is associated with the thread. This queue is called the asynchronous procedure call (APC) queue. When issuing an I/O request, you can tell the device driver to append an entry to the calling thread's APC queue. To have completed I/O notifications queued to your thread's APC queue, you call the **ReadFileEx** and **WriteFileEx** functions:

```
BOOL ReadFileEx(
HANDLE hFile,
PVOID pvBuffer,
DWORD nNumBytesToRead,
OVERLAPPED* pOverlapped,
LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
BOOL WriteFileEx(
HANDLE hFile,
CONST VOID *pvBuffer,
DWORD nNumBytesToWrite,
OVERLAPPED* pOverlapped,
LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
```

Like **ReadFile** and **WriteFile**, **ReadFileEx** and **WriteFileEx** issue I/O requests to a device driver, and the functions return immediately. The **ReadFileEx** and **WriteFileEx** functions have the same parameters as the **ReadFile** and **WriteFile** functions, with two exceptions. First, the ***Ex** functions are not passed a pointer to a **DWORD** that gets filled with the number of bytes transferred; this information can be retrieved only by the callback function. Second, the ***Ex** functions require that you pass the address of a callback function, called a *completion routine*. This routine must have the following prototype:

```
VOID WINAPI CompletionRoutine(
  DWORD dwError,
  DWORD dwNumBytes,
  OVERLAPPED* po);
```

When you issue an asynchronous I/O request with **ReadFileEx** and **WriteFileEx**, the functions pass the address of this function to the device driver. When the device driver has completed the I/O request, it appends an entry in the issuing thread's APC queue. This entry contains the address of the completion routine function and the address of the **OVERLAPPED** structure used to initiate the I/O request.



Note By the way, when an alertable I/O completes, the device driver does not attempt to signal an event object. In fact, the device does not reference the **OVERLAPPED** structure's **hEvent** member at all. Therefore, you can use the **hEvent** member for your own purposes if you like.

When the thread is in an alertable state (discussed shortly), the system examines its APC queue and, for every entry in the queue, the system calls the completion function, passing it the I/O error code, the number of bytes transferred, and the address of the **OVERLAPPED** structure. Note that the error code and number of bytes transferred can also be found in the **OVERLAPPED** structure's **Internal** and **InternalHigh** members. (As I mentioned earlier, Microsoft originally didn't want to document these, so it passed them as parameters to the function.)

I'll get back to this completion routine function shortly. First let's look at how the system handles the asynchronous I/O requests. The following code queues three different asynchronous operations:

```
hFile = CreateFile(..., FILE_FLAG_OVERLAPPED, ...);
```

```
ReadFileEx(hFile, ...); // Perform first ReadFileEx
WriteFileEx(hFile, ...); // Perform first WriteFileEx
ReadFileEx(hFile, ...); // Perform second ReadFileEx
```

SomeFunc();

If the call to **SomeFunc** takes some time to execute, the system completes the three operations before **SomeFunc** returns. While the thread is executing the **SomeFunc** function, the device driver is appending completed I/O entries to the thread's APC queue. The APC queue might look something like this:

```
first WriteFileEx completed
second ReadFileEx completed
first ReadFileEx completed
```

The APC queue is maintained internally by the system. You'll also notice from the list that the system can execute your queued I/O requests in any order, and that the I/O requests that you issue last might be completed first and vice versa. Each entry in your thread's APC queue contains the address of a callback function and a value that is passed to the function.

As I/O requests complete, they are simply queued to your thread's APC queue—the callback routine is not immediately called because your thread might be busy doing something else and cannot be interrupted. To process entries in your thread's APC queue, the thread must put itself in an alertable state. This simply means that your thread has reached a position in its execution where it can handle being interrupted. Windows offers six functions that can place a thread in an alertable state:

```
DWORD SleepEx(
   DWORD dwMilliseconds,
   BOOL bAlertable);
DWORD WaitForSingleObjectEx(
   HANDLE hObject,
   DWORD dwMilliseconds,
   BOOL bAlertable);
DWORD WaitForMultipleObjectsEx(
   DWORD cObjects,
   CONST HANDLE* phObjects,
   BOOL bWaitAll,
   DWORD dwMilliseconds,
   BOOL bAlertable);
BOOL SignalObjectAndWait(
   HANDLE hObjectToSignal,
   HANDLE hObjectToWaitOn,
   DWORD dwMilliseconds,
   BOOL bAlertable);
BOOL GetQueuedCompletionStatusEx(
   HANDLE hCompPort.
   LPOVERLAPPED_ENTRY pCompPortEntries,
   ULONG ulCount,
   PULONG pulNumEntriesRemoved,
   DWORD dwMilliseconds,
   BOOL bAlertable);
DWORD MsgWaitForMultipleObjectsEx(
   DWORD nCount,
   CONST HANDLE* pHandles,
   DWORD dwMilliseconds,
   DWORD dwWakeMask,
   DWORD dwFlags);
```

The last argument to the first five functions is a Boolean value indicating whether the calling thread should place itself in an alertable state. For **MsgWaitForMultipleObjectsEx**, you must use the **MWMO_ALERTABLE** flag to have the thread enter an alertable state. If you're familiar with the **Sleep**, **WaitForSingleObject**, and **WaitForMultipleObjects** functions, you shouldn't be surprised to learn that, internally, these non-**Ex** functions call their **Ex** counterparts, always passing **FALSE** for the **bAlertable** parameter.

When you call one of the six functions just mentioned and place your thread in an alertable state, the system first checks your thread's APC queue. If at least one entry is in the queue, the system does not put your thread to sleep. Instead, the system pulls the entry from the APC queue and your thread calls the callback routine, passing the routine the completed I/O request's error code, number of bytes transferred, and address of the **OVERLAPPED** structure. When the callback routine returns to the system checks for more entries in the APC queue. If more entries exist,

318 Windows via C/C++

they are processed. However, if no more entries exist, your call to the alertable function returns. Something to keep in mind is that if any entries are in your thread's APC queue when you call any of these functions, your thread never sleeps!

The only time these functions suspend your thread is when no entries are in your thread's APC queue at the time you call the function. While your thread is suspended, the thread will wake up if the kernel object (or objects) that you're waiting on becomes signaled or if an APC entry appears in your thread's queue. Because your thread is in an alertable state, as soon as an APC entry appears, the system wakes your thread and empties the queue (by calling the callback routines). Then the functions immediately return to the caller—your thread does not go back to sleep waiting for kernel objects to become signaled.

The return value from these six functions indicates why they have returned. If they return **WAIT_**IO_COMPLETION (or if **GetLastError** returns **WAIT_IO_COMPLETION**), you know that the thread is continuing to execute because at least one entry was processed from the thread's APC queue. If the methods return for any other reason, the thread woke up because the sleep period expired, the specified kernel object or objects became signaled, or a mutex was abandoned.

The Bad and the Good of Alertable I/O

At this point, we've discussed the mechanics of performing alertable I/O. Now you need to know about the two issues that make alertable I/O a horrible method for doing device I/O:

- **Callback functions** Alertable I/O requires that you create callback functions, which makes implementing your code much more difficult. These callback functions typically don't have enough contextual information about a particular problem to guide you, so you end up placing a lot of information in global variables. Fortunately, these global variables don't need to be synchronized because the thread calling one of the six alterable functions is the same thread executing the callback functions. A single thread can't be in two places at one time, so the variables are safe.
- **Threading issues** The real big problem with alertable I/O is this: The thread issuing the I/O request must also handle the completion notification. If a thread issues several requests, that thread must respond to each request's completion notification, even if other threads are sitting completely idle. Because there is no load balancing, the application doesn't scale well.

Both of these problems are pretty severe, so I strongly discourage the use of alertable I/O for device I/O. I'm sure you guessed by now that the I/O completion port mechanism, discussed in the next section, solves both of the problems I just described. I promised to tell you some good stuff about the alertable I/O infrastructure, so before I move on to the I/O completion port, I'll do that.

Windows offers a function that allows you to manually queue an entry to a thread's APC queue:

```
DWORD QueueUserAPC(
PAPCFUNC pfnAPC,
HANDLE hThread,
ULONG_PTR dwData);
```

The first parameter is a pointer to an APC function that must have the following prototype:

```
VOID WINAPI APCFunc(ULONG_PTR dwParam);
```

The second parameter is the handle of the thread for which you want to queue the entry. Note that this thread can be any thread in the system. If **hThread** identifies a thread in a different process' address space, **pfnAPC** must specify the memory address of a function that is in the address space of the target thread's process. The last parameter to **QueueUserAPC**, **dwData**, is a value that simply gets passed to the callback function.

Even though **QueueUserAPC** is prototyped as returning a **DWORD**, the function actually returns a **BOOL** indicating success or failure. You can use **QueueUserAPC** to perform extremely efficient interthread communication, even across process boundaries. Unfortunately, however, you can pass only a single value.

QueueUserAPC can also be used to force a thread out of a wait state. Suppose you have a thread calling **WaitForSingleObject**, waiting for a kernel object to become signaled. While the thread is waiting, the user wants to terminate the application. You know that threads should cleanly destroy themselves, but how do you force the thread waiting on the kernel object to wake up and kill itself? **QueueUserAPC** is the answer.

The following code demonstrates how to force a thread out of a wait state so that the thread can exit cleanly. The main function spawns a new thread, passing it the handle of some kernel object. While the secondary thread is running, the primary thread is also running. The secondary thread (executing the **ThreadFunc** function) calls **WaitForSingleObjectEx**, which suspends the thread, placing it in an alertable state. Now, say that the user tells the primary thread to terminate the application. Sure, the primary thread could just exit, and the system would kill the whole process. However, this approach is not very clean, and in many scenarios, you'll just want to kill an operation without terminating the whole process.

So the primary thread calls **QueueUserAPC**, which places an APC entry in the secondary thread's APC queue. Because the secondary thread is in an alertable state, it now wakes and empties its APC queue by calling the **APCFunc** function. This function does absolutely nothing and just returns. Because the APC queue is now empty, the thread returns from its call to **WaitForSingleObjectEx** with a return value of **WAIT_IO_COMPLETION**. The **ThreadFunc** function checks specifically for this return value, knowing that it received an APC entry indicating that the thread should exit.

```
// The APC callback function has nothing to do
VOID WINAPI APCFunc(ULONG_PTR dwParam) {
   // Nothing to do in here
}
UINT WINAPI ThreadFunc(PVOID pvParam) {
   HANDLE hEvent = (HANDLE) pvParam; // Handle is passed to this thread
   // Wait in an alertable state so that we can be forced to exit cleanly
   DWORD dw = WaitForSingleObjectEx(hEvent, INFINITE, TRUE);
   if (dw == WAIT_OBJECT_0) {
      // Object became signaled
   }
   if (dw == WAIT_IO_COMPLETION) {
      // QueueUserAPC forced us out of a wait state
      return(0); // Thread dies cleanly
   }
   return(0);
}
```

```
void main() {
  HANDLE hEvent = CreateEvent(...);
  HANDLE hThread = (HANDLE) _beginthreadex(NULL, 0,
    ThreadFunc, (PVOID) hEvent, 0, NULL);
   ...
  // Force the secondary thread to exit cleanly
  QueueUserAPC(APCFunc, hThread, NULL);
  WaitForSingleObject(hThread, INFINITE);
  CloseHandle(hThread);
  CloseHandle(hEvent);
}
```

I know that some of you are thinking that this problem could have been solved by replacing the call to **WaitForSingleObjectEx** with a call to **WaitForMultipleObjects** and by creating another event kernel object to signal the secondary thread to terminate. For my simple example, your solution would work. However, if my secondary thread called **WaitForMultipleObjects** to wait until all objects became signaled, **QueueUserAPC** would be the only way to force the thread out of a wait state.

I/O Completion Ports

Windows is designed to be a secure, robust operating system running applications that service literally thousands of users. Historically, you've been able to architect a service application by following one of two models:

- **Serial model** A single thread waits for a client to make a request (usually over the network). When the request comes in, the thread wakes and handles the client's request.
- **Concurrent model** A single thread waits for a client request and then creates a new thread to handle the request. While the new thread is handling the client's request, the original thread loops back around and waits for another client request. When the thread that is handling the client's request is completely processed, the thread dies.

The problem with the serial model is that it does not handle multiple, simultaneous requests well. If two clients make requests at the same time, only one can be processed at a time; the second request must wait for the first request to finish processing. A service that is designed using the serial approach cannot take advantage of multiprocessor machines. Obviously, the serial model is good only for the simplest of server applications, in which few client requests are made and requests can be handled very quickly. A Ping server is a good example of a serial server.

Because of the limitations in the serial model, the concurrent model is extremely popular. In the concurrent model, a thread is created to handle each client request. The advantage is that the thread waiting for incoming requests has very little work to do. Most of the time, this thread is sleeping. When a client request comes in, the thread wakes, creates a new thread to handle the request, and then waits for another client request. This means that incoming client requests are handled expediently. Also, because each client request gets its own thread, the server application scales well and can easily take advantage of multiprocessor machines. So if you are using the concurrent model and upgrade the hardware (add another CPU), the performance of the server application improves.

Service applications using the concurrent model were implemented using Windows. The Windows team noticed that application performance was not as high as desired. In particular, the

team noticed that handling many simultaneous client requests meant that many threads were running in the system concurrently. Because all these threads were *runnable* (not suspended and waiting for something to happen), Microsoft realized that the Windows kernel spent too much time context switching between the running threads, and the threads were not getting as much CPU time to do their work. To make Windows an awesome server environment, Microsoft needed to address this problem. The result is the I/O completion port kernel object.

Creating an I/O Completion Port

The theory behind the I/O completion port is that the number of threads running concurrently must have an upper bound—that is, 500 simultaneous client requests cannot allow 500 runnable threads to exist. What, then, is the proper number of concurrent, runnable threads? Well, if you think about this question for a moment, you'll come to the realization that if a machine has two CPUs, having more than two runnable threads—one for each processor—really doesn't make sense. As soon as you have more runnable threads than CPUs available, the system has to spend time performing thread context switches, which wastes precious CPU cycles—a potential deficiency of the concurrent model.

Another deficiency of the concurrent model is that a new thread is created for each client request. Creating a thread is cheap when compared to creating a new process with its own virtual address space, but creating threads is far from free. The service application's performance can be improved if a pool of threads is created when the application initializes, and these threads hang around for the duration of the application. I/O completion ports were designed to work with a pool of threads.

An I/O completion port is probably the most complex kernel object. To create an I/O completion port, you call **CreateIoCompletionPort**:

```
HANDLE CreateIoCompletionPort(
HANDLE hFile,
HANDLE hExistingCompletionPort,
ULONG_PTR CompletionKey,
DWORD dwNumberOfConcurrentThreads);
```

This function performs two different tasks: it creates an I/O completion port, and it associates a device with an I/O completion port. This function is overly complex, and in my opinion, Microsoft should have split it into two separate functions. When I work with I/O completion ports, I separate these two capabilities by creating two tiny functions that abstract the call to **CreateIo CompletionPort**. The first function I write is called **CreateNewCompletionPort**, and I implement it as follows:

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads) {
```

}

This function takes a single argument, **dwNumberOfConcurrentThreads**, and then calls the Windows **CreateIoCompletionPort** function, passing in hard-coded values for the first three parameters and **dwNumberOfConcurrentThreads** for the last parameter. You see, the first three parameters to **CreateIoCompletionPort** are used only when you are associating a device with a completion port. (I'll talk about this shortly.) To create just a completion port, I pass **INVALID**_ **HANDLE_VALUE**, **NULL**, and **0**, respectively, to **CreateIoCompletionPort**'s first three parameters.

322 Windows via C/C++

The **dwNumberOfConcurrentThreads** parameter tells the I/O completion port the maximum number of threads that should be runnable at the same time. If you pass **0** for the **dwNumberOf ConcurrentThreads** parameter, the completion port defaults to allowing as many concurrent threads as there are CPUs on the host machine. This is usually exactly what you want so that extra context switching is avoided. You might want to increase this value if the processing of a client request requires a lengthy computation that rarely blocks, but increasing this value is strongly discouraged. You might experiment with the **dwNumberOfConcurrentThreads** parameter by trying different values and comparing your application's performance on your target hardware.

You'll notice that **CreateIoCompletionPort** is about the only Windows function that creates a kernel object but does not have a parameter that allows you to pass the address of a **SECURITY_ATTRIBUTES** structure. This is because completion ports are intended for use within a single process only. The reason will be clear to you when I explain how to use completion ports.

Associating a Device with an I/O Completion Port

When you create an I/O completion port, the kernel actually creates five different data structures, as shown in Figure 10-1. You should refer to this figure as you continue reading.

The first data structure is a device list indicating the device or devices associated with the port. You associate a device with the port by calling **CreateIoCompletionPort**. Again, I created my own function, **AssociateDeviceWithCompletionPort**, which abstracts the call to **CreateIoCom pletionPort**:

```
BOOL AssociateDeviceWithCompletionPort(
   HANDLE hCompletionPort, HANDLE hDevice, DWORD dwCompletionKey) {
   HANDLE h = CreateIoCompletionPort(hDevice, hCompletionPort, dwCompletionKey, 0);
   return(h == hCompletionPort);
```

}

AssociateDeviceWithCompletionPort appends an entry to an existing completion port's device list. You pass to the function the handle of an existing completion port (returned by a previous call to **CreateNewCompletionPort**), the handle of the device (this can be a file, a socket, a mailslot, a pipe, and so on), and a completion key (a value that has meaning to you; the operating system doesn't care what you pass here). Each time you associate a device with the port, the system appends this information to the completion port's device list.



Note The **CreateIoCompletionPort** function is complex, and I recommend that you mentally separate the two reasons for calling it. There is one advantage to having the function be so complex: you can create an I/O completion port and associate a device with it at the same time. For example, the following code opens a file and creates a new completion port, associating the file with it. All I/O requests to the file complete with a completion key of **CK_FILE**, and the port allows as many as two threads to execute concurrently.

```
#define CK_FILE 1
HANDLE hFile = CreateFile(...);
HANDLE hCompletionPort = CreateIoCompletionPort(hFile,NULL,CK_FILE,2);
```

Device List



I/O Completion Queue (FIFO)

Each record contains						
dwBytesTransferred	dwCompletionKey	pOverlapped	dwError			
 Entry is added when I/O request completes. PostQueuedCompletionStatus is called. 						
Entry is removed whenCompletion port removes an entry from the Waiting Thread Queue.						

Waiting Thread Queue (LIFO)



Figure 10-1 The internal workings of an I/O completion port

324 Windows via C/C++

The second data structure is an I/O completion queue. When an asynchronous I/O request for a device completes, the system checks to see whether the device is associated with a completion port and, if it is, the system appends the completed I/O request entry to the end of the completion port's I/O completion queue. Each entry in this queue indicates the number of bytes transferred, the completion key value that was set when the device was associated with the port, the pointer to the I/O request's **OVERLAPPED** structure, and an error code. I'll discuss how entries are removed from this queue shortly.



Note Issuing an I/O request to a device and not having an I/O completion entry queued to the I/O completion port is possible. This is not usually necessary, but it can come in handy occasionally—for example, when you send data over a socket and you don't care whether the data actually makes it or not.

To issue an I/O request without having a completion entry queued, you must load the **OVER LAPPED** structure's **hEvent** member with a valid event handle and bitwise-**OR** this value with 1, like this:

```
Overlapped.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWORD_PTR) Overlapped.hEvent | 1);
ReadFile(..., &Overlapped);
```

Now you can issue your I/O request, passing the address of this **OVERLAPPED** structure to the desired function (such as **ReadFile** above).

It would be nice if you didn't have to create an event just to stop the queuing of the I/O completion. I would like to be able to do the following, but it doesn't work:

Overlapped.hEvent = 1;

```
ReadFile(..., &Overlapped);
```

Also, don't forget to reset the low-order bit before closing this event handle:

CloseHandle((HANDLE) ((DWORD_PTR) Overlapped.hEvent & ~1));

Architecting Around an I/O Completion Port

When your service application initializes, it should create the I/O completion port by calling a function such as **CreateNewCompletionPort**. The application should then create a pool of threads to handle client requests. The question you ask now is, "How many threads should be in the pool?" This is a tough question to answer, and I will address it in more detail later in "How Many Threads in the Pool?" on page 328. For now, a standard rule of thumb is to take the number of CPUs on the host machine and multiply it by 2. So on a dual-processor machine, you should create a pool of four threads.

All the threads in the pool should execute the same function. Typically, this thread function performs some sort of initialization and then enters a loop that should terminate when the service process is instructed to stop. Inside the loop, the thread puts itself to sleep waiting for device I/O requests to complete to the completion port. Calling **GetQueuedCompletionStatus** does this:

BOOL GetQueuedCompletionStatus(

```
HANDLEhCompletionPort,PDWORDpdwNumberOfBytesTransferred,PULONG_PTRpCompletionKey,OVERLAPPED**ppOverlapped,DWORDdwMilliseconds);
```

The first parameter, **hCompletionPort**, indicates which completion port the thread is interested in monitoring. Many service applications use a single I/O completion port and have all I/O request notifications complete to this one port. Basically, the job of **GetQueuedCompletionStatus** is to put the calling thread to sleep until an entry appears in the specified completion port's I/O completion queue or until the specified time-out occurs (as specified in the **dwMilliseconds** parameter).

The third data structure associated with an I/O completion port is the waiting thread queue. As each thread in the thread pool calls **GetQueuedCompletionStatus**, the ID of the calling thread is placed in this waiting thread queue, enabling the I/O completion port kernel object to always know which threads are currently waiting to handle completed I/O requests. When an entry appears in the port's I/O completion queue, the completion port wakes one of the threads in the waiting thread queue. This thread gets the pieces of information that make up a completed I/O entry: the number of bytes transferred, the completion key, and the address of the **OVERLAPPED** structure. This information is returned to the thread via the **pdwNumberOfBytesTransferred**, **pCompletionKey**, and **ppOverlapped** parameters passed to **GetQueuedCompletionStatus**.

Determining the reason that **GetQueuedCompletionStatus** returned is somewhat difficult. The following code demonstrates the proper way to do it:

```
DWORD dwNumBytes;
ULONG_PTR CompletionKey;
OVERLAPPED* pOverlapped;
// hIOCP is initialized somewhere else in the program
BOOL bOk = GetQueuedCompletionStatus(hIOCP,
   &dwNumBytes, &CompletionKey, &pOverlapped, 1000);
DWORD dwError = GetLastError();
if (b0k) {
   // Process a successfully completed I/O request
} else {
   if (pOverlapped != NULL) {
      // Process a failed completed I/O request
      // dwError contains the reason for failure
    } else {
      if (dwError == WAIT_TIMEOUT) {
         // Time-out while waiting for completed I/O entry
      } else {
         // Bad call to GetQueuedCompletionStatus
         // dwError contains the reason for the bad call
      }
   }
}
```

As you would expect, entries are removed from the I/O completion queue in a first-in first-out fashion. However, as you might not expect, threads that call **GetQueuedCompletionStatus** are awakened in a last-in first-out (LIFO) fashion. The reason for this is again to improve performance. For example, say that four threads are waiting in the waiting thread queue. If a single completed I/O entry appears, the last thread to call **GetQueuedCompletionStatus** wakes up to process the entry. When this last thread is finished processing the entry, the thread again calls **GetQueued CompletionStatus** to enter the waiting thread queue. Now if another I/O completion entry appears, the same thread that processed the first entry is awakened to process the new entry.

326 Windows via C/C++

As long as I/O requests complete so slowly that a single thread can handle them, the system just keeps waking the one thread, and the other three threads continue to sleep. By using this LIFO algorithm, threads that don't get scheduled can have their memory resources (such as stack space) swapped out to the disk and flushed from a processor's cache. This means having many threads waiting on a completion port isn't bad. If you do have several threads waiting but few I/O requests completing, the extra threads have most of their resources swapped out of the system anyway.

In Windows Vista, if you expect a large number of I/O requests to be constantly submitted, instead of multiplying the number of threads to wait on the completion port and incurring the increasing cost of the corresponding context switches, you can retrieve the result of several I/O requests at the same time by calling the following function:

```
BOOL GetQueuedCompletionStatusEx(
HANDLE hCompletionPort,
LPOVERLAPPED_ENTRY pCompletionPortEntries,
ULONG ulCount,
PULONG pulNumEntriesRemoved,
DWORD dwMilliseconds,
BOOL bAlertable);
```

The first parameter, **hCompletionPort**, indicates which completion port the thread is interested in monitoring. The entries present in the specified completion port's I/O completion queue when this function is called are retrieved, and their description is copied into the **pCompletionPort Entries** array parameter. The **ulCount** parameter indicates how many entries can be copied in this array, and the long value pointed to by **pulNumEntriesRemoved** receives the exact number of I/O requests that were extracted from the completion queue.

Each element of the **pCompletionPortEntries** array is an **OVERLAPPED_ENTRY** that stores the pieces of information that make up a completed I/O entry: the completion key, the address of the **OVERLAPPED** structure, the result code (error) of the I/O request, and the number of bytes transferred.

```
typedef struct _OVERLAPPED_ENTRY {
    ULONG_PTR lpCompletionKey;
    LPOVERLAPPED lpOverlapped;
    ULONG_PTR Internal;
    DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

The **Internal** field is opaque and should not be used.

If the last **bAlertable** parameter is set to **FALSE**, the function waits for a completed I/O request to be queued on the completion port until the specified time-out occurs (as specified in the **dwMilliseconds** parameter). If the **bAlertable** parameter is set to **TRUE** and there is no completed I/O request in the queue, the thread enters an alertable state as explained earlier in this chapter.



Note When you issue an asynchronous I/O request to a device that is associated with a completion port, Windows queues the result to the completion port. Windows does this even if the asynchronous request is performed synchronously in order to give the programmer a consistent programming model. However, maintaining this consistent programming model hurts performance slightly because the completed request information must be placed in the port and a thread must extract it from the port.

To improve performance slightly, you can tell Windows not to queue a synchronously performed asynchronous request to the completion port associated with the device by calling the **SetFileCompletionNotificationModes** function (described in "Signaling an Event Kernel Object" on page 312) passing it the **FILE_SKIP_COMPLETION_PORT_ON_ SUCCESS** flag.

The extremely performance-conscious programmer might also want to consider use of the **SetFileIoOverlappedRange** function. (See the Platform SDK documentation for more information.)

How the I/O Completion Port Manages the Thread Pool

Now it's time to discuss why I/O completion ports are so useful. First, when you create the I/O completion port, you specify the number of threads that can run concurrently. As I said, you usually set this value to the number of CPUs on the host machine. As completed I/O entries are queued, the I/O completion port wants to wake up waiting threads. However, the completion port wakes up only as many threads as you have specified. So if four I/O requests complete and four threads are waiting in a call to **GetQueuedCompletionStatus**, the I/O completion port will allow only two threads to wake up; the other two threads continue to sleep. As each thread processes a completed I/O entry, the thread again calls **GetQueuedCompletionStatus**. The system sees that more entries are queued and wakes the same threads to process the remaining entries.

If you're thinking about this carefully, you should notice that something just doesn't make a lot of sense: if the completion port only ever allows the specified number of threads to wake up concurrently, why have more threads waiting in the thread pool? For example, suppose I'm running on a machine with two CPUs and I create the I/O completion port, telling it to allow no more than two threads to process entries concurrently. But I create four threads (twice the number of CPUs) in the thread pool. It seems as though I am creating two additional threads that will never be awakened to process anything.

But I/O completion ports are very smart. When a completion port wakes a thread, the completion port places the thread's ID in the fourth data structure associated with the completion port, a released thread list. (See Figure 10-1.) This allows the completion port to remember which threads it awakened and to monitor the execution of these threads. If a released thread calls any function that places the thread in a wait state, the completion port detects this and updates its internal data structures by moving the thread's ID from the released thread list to the paused thread list (the fifth and final data structure that is part of an I/O completion port).

The goal of the completion port is to keep as many entries in the released thread list as are specified by the concurrent number of threads value used when creating the completion port. If a released thread enters a wait state for any reason, the released thread list shrinks and the completion port releases another waiting thread. If a paused thread wakes, it leaves the paused thread list and reenters the released thread list. This means that the released thread list can now have more entries in it than are allowed by the maximum concurrency value.



Note Once a thread calls **GetQueuedCompletionStatus**, the thread is "assigned" to the specified completion port. The system assumes that all assigned threads are doing work on behalf of the completion port. The completion port wakes threads from the pool only if the number of running assigned threads is less than the completion port's maximum concurrency value.

You can break the thread/completion port assignment in one of three ways:

- Have the thread exit.
- Have the thread call **GetQueuedCompletionStatus**, passing the handle of a different I/O completion port.
- Destroy the I/O completion port that the thread is currently assigned to.

Let's tie all of this together now. Say that we are again running on a machine with two CPUs. We create a completion port that allows no more than two threads to wake concurrently, and we create four threads that are waiting for completed I/O requests. If three completed I/O requests get queued to the port, only two threads are awakened to process the requests, reducing the number of runnable threads and saving context-switching time. Now if one of the running threads calls **Sleep**, **WaitForSingleObject**, **WaitForMultipleObjects**, **SignalObjectAndWait**, a synchronous I/O call, or any function that would cause the thread not to be runnable, the I/O completion port would detect this and wake a third thread immediately. The goal of the completion port is to keep the CPUs saturated with work.

Eventually, the first thread will become runnable again. When this happens, the number of runnable threads will be higher than the number of CPUs in the system. However, the completion port again is aware of this and will not allow any additional threads to wake up until the number of threads drops below the number of CPUs. The I/O completion port architecture presumes that the number of runnable threads will stay above the maximum for only a short time and will die down quickly as the threads loop around and again call **GetQueuedCompletionStatus**. This explains why the thread pool should contain more threads than the concurrent thread count set in the completion port.

How Many Threads in the Pool?

Now is a good time to discuss how many threads should be in the thread pool. Consider two issues. First, when the service application initializes, you want to create a minimum set of threads so that you don't have to create and destroy threads on a regular basis. Remember that creating and destroying threads wastes CPU time, so you're better off minimizing this process as much as possible. Second, you want to set a maximum number of threads because creating too many threads wastes system resources. Even if most of these resources can be swapped out of RAM, minimizing the use of system resources and not wasting even paging file space is to your advantage, if you can manage it.

You will probably want to experiment with different numbers of threads. Most services (including Microsoft Internet Information Services) use heuristic algorithms to manage their thread pools. I

recommend that you do the same. For example, you can create the following variables to manage the thread pool:

```
LONG g_nThreadsMin; // Minimum number of threads in pool
LONG g_nThreadsMax; // Maximum number of threads in pool
LONG g_nThreadsCrnt; // Current number of threads in pool
LONG g_nThreadsBusy; // Number of busy threads in pool
```

When your application initializes, you can create the **g_nThreadsMin** number of threads, all executing the same thread pool function. The following pseudocode shows how this thread function might look:

```
DWORD WINAPI ThreadPoolFunc(PVOID pv) {
   // Thread is entering pool
   InterlockedIncrement(&g_nThreadsCrnt);
   InterlockedIncrement(&g_nThreadsBusy);
   for (BOOL bStayInPool = TRUE; bStayInPool;) {
     // Thread stops executing and waits for something to do
     InterlockedDecrement(&m_nThreadsBusy);
     BOOL bOk = GetQueuedCompletionStatus(...);
     DWORD dwIOError = GetLastError();
     // Thread has something to do, so it's busy
     int nThreadsBusy = InterlockedIncrement(&m_nThreadsBusy);
     // Should we add another thread to the pool?
     if (nThreadsBusy == m_nThreadsCrnt) {
                                               // All threads are busy
                                             // The pool isn't full
         if (nThreadsBusy < m_nThreadsMax) {</pre>
            if (GetCPUUsage() < 75) { // CPU usage is below 75%
               // Add thread to pool
               CloseHandle(chBEGINTHREADEX(...));
           }
         }
     }
     if (!bOk && (dwIOError == WAIT_TIMEOUT)) { // Thread timed out
         // There isn't much for the server to do, and this thread
         // can die even if it still has outstanding I/O requests
         bStayInPool = FALSE;
     }
     if (bOk || (po != NULL)) {
         // Thread woke to process something; process it
         . . .
         if (GetCPUUsage() > 90) {
                                         // CPU usage is above 90%
             if (g_nThreadsCrnt > g_nThreadsMin)) { // Pool above min
                bStayInPool = FALSE; // Remove thread from pool
           }
        }
     }
  }
```

}

```
// Thread is leaving pool
InterlockedDecrement(&g_nThreadsBusy);
InterlockedDecrement(&g_nThreadsCurrent);
return(0);
```

This pseudocode shows how creative you can get when using an I/O completion port. The **Get CPUUsage** function is not part of the Windows API. If you want its behavior, you'll have to implement the function yourself. In addition, you must make sure that the thread pool always contains at least one thread in it, or clients will never get tended to. Use my pseudocode as a guide, but your particular service might perform better if structured differently.



Note Earlier in this chapter, in "Canceling Queued Device I/O Requests" on page 309, I said that the system automatically cancels all pending I/O requests issued by a thread when that thread terminates. Before Windows Vista, when a thread issued an I/O request against a device associated with a completion port, it was mandatory that the thread remain alive until the request completed; otherwise, Windows canceled any outstanding requests made by the thread. With Windows Vista, this is no longer necessary: threads can now issue requests and terminate; the request will still be processed and the result will be queued to the completion port.

Many services offer a management tool that allows an administrator to have some control over the thread pool's behavior—for example, to set the minimum and maximum number of threads, the CPU time usage thresholds, and also the maximum concurrency value used when creating the I/O completion port.

Simulating Completed I/O Requests

I/O completion ports do not have to be used with device I/O at all. This chapter is also about interthread communication techniques, and the I/O completion port kernel object is an awesome mechanism to use to help with this. In "Alertable I/O" on page 315, I presented the **QueueUserAPC** function, which allows a thread to post an APC entry to another thread. I/O completion ports have an analogous function, **PostQueuedCompletionStatus**:

```
BOOL PostQueuedCompletionStatus(
HANDLE hCompletionPort,
```

DWORD	dwNumBytes,
ULONG_PTR	CompletionKey,
OVERLAPPED*	pOverlapped);

This function appends a completed I/O notification to an I/O completion port's queue. The first parameter, **hCompletionPort**, identifies the completion port that you want to queue the entry for. The remaining three parameters—**dwNumBytes**, **CompletionKey**, and **pOverlapped**—indicate the values that should be returned by a thread's call to **GetQueuedCompletionStatus**. When a thread pulls a simulated entry from the I/O completion queue, **GetQueuedCompletionStatus** returns **TRUE**, indicating a successfully executed I/O request.

The **PostQueuedCompletionStatus** function is incredibly useful—it gives you a way to communicate with all the threads in your pool. For example, when the user terminates a service application, you want all the threads to exit cleanly. But if the threads are waiting on the completion port and

no I/O requests are coming in, the threads can't wake up. By calling **PostQueuedCompletion Status** once for each thread in the pool, each thread can wake up, examine the values returned from **GetQueuedCompletionStatus**, see that the application is terminating, and clean up and exit appropriately.

You must be careful when using a thread termination technique like the one I just described. My example works because the threads in the pool are dying and not calling **GetQueuedCompletion Status** again. However, if you want to notify each of the pool's threads of something and have them loop back around to call **GetQueuedCompletionStatus** again, you will have a problem because the threads wake up in a LIFO order. So you will have to employ some additional thread synchronization in your application to ensure that each pool thread gets the opportunity to see its simulated I/O entry. Without this additional thread synchronization, one thread might see the same notification several times.



Note In Windows Vista, when you call **CloseHandle** passing the handle of a completion port, all threads waiting in a call to **GetQueuedCompletionStatus** wake up and **FALSE** is returned to them. A call to **GetLastError** will return **ERROR_INVALID_HANDLE**; the threads can use this to know that it is time to die gracefully.

The FileCopy Sample Application

The FileCopy sample application (10-FileCopy.exe), shown at the end of this chapter, demonstrates the use of I/O completion ports. The source code and resource files for the application are in the 10-FileCopy directory on the companion content Web page. The program simply copies a file specified by the user to a new file called FileCopy.cpy. When the user executes FileCopy, the dialog box shown in Figure 10-2 appears.

En FileCopy		- • ×
Pathname		
Сору	File size:	0

Figure 10-2 The dialog box for the FileCopy sample application

The user clicks the Pathname button to select the file to be copied, and the Pathname and File Size fields are updated. When the user clicks the Copy button, the program calls the **FileCopy** function, which does all the hard work. Let's concentrate our discussion on the **FileCopy** function.

When preparing to copy, **FileCopy** opens the source file and retrieves its size, in bytes. I want the file copy to execute as blindingly fast as possible, so the file is opened using the **FILE_FLAG_NO_BUFFERING** flag. Opening the file with the **FILE_FLAG_NO_BUFFERING** flag allows me to access the file directly, bypassing the additional memory copy overhead incurred when allowing the system's cache to "help" access the file. Of course, accessing the file directly means slightly more work for me: I must always access the file using offsets that are multiples of the disk volume's sector size, and I must read and write data that is a multiple of the sector's size as well. I chose to transfer the file's data in **BUFFSIZE** (64 KB) chunks, which is guaranteed to be a multiple of the sector size. This is why I round up the source file's size to a multiple of **BUFFSIZE**. You'll also notice that the source file is opened with the **FILE_FLAG_OVERLAPPED** flag so that I/O requests against the file are performed asynchronously.

332 Windows via C/C++

The destination file is opened similarly: both the **FILE_FLAG_NO_BUFFERING** and **FILE_FLAG_ OVERLAPPED** flags are specified. I also pass the handle of the source file as **CreateFile**'s **hFile Template** parameter when creating the destination file, causing the destination file to have the same attributes as the source.



Note Once both files are open, the destination file size is immediately set to its maximum size by calling **SetFilePointerEx** and **SetEndOfFile**. Adjusting the destination file's size now is extremely important because NTFS maintains a high-water marker that indicates the highest point at which the file was written. If you read past this marker, the system knows to return zeros. If you write past the marker, the file's data from the old high-water marker to the write offset is filled with zeros, your data is written to the file, and the file's high-water marker is updated. This behavior satisfies C2 security requirements pertaining to not presenting prior data. When you write to the end of a file on an NTFS partition, causing the high-water marker to move, NTFS must perform the I/O request synchronously even if asynchronous I/O was desired. If the **FileCopy** function didn't set the size of the destination file, none of the overlapped I/O requests would be performed asynchronously.

Now that the files are opened and ready to be processed, **FileCopy** creates an I/O completion port. To make working with I/O completion ports easier, I created a small C++ class, CIOCP, that is a very simple wrapper around the I/O completion port functions. This class can be found in the IOCP.h file discussed in Appendix A, "The Build Environment." **FileCopy** creates an I/O completion port by creating an instance (named **iocp**) of my CIOCP class.

The source file and destination file are associated with the completion port by calling the CIOCP's **AssociateDevice** member function. When associated with the completion port, each device is assigned a completion key. When an I/O request completes against the source file, the completion key is **CK_READ**, indicating that a read operation must have completed. Likewise, when an I/O request completes against the destination file, the completion key is **CK_WRITE**, indicating that a write operation must have completed.

Now we're ready to initialize a set of I/O requests (**OVERLAPPED** structures) and their memory buffers. The **FileCopy** function keeps four (**MAX_PENDING_IO_REQS**) I/O requests outstanding at any one time. For applications of your own, you might prefer to allow the number of I/O requests to dynamically grow or shrink as necessary. In the FileCopy program, the **CIOReq** class encapsulates a single I/O request. As you can see, this C++ class is derived from an **OVERLAPPED** structure but contains some additional context information. **FileCopy** allocates an array of **CIOReq** objects and calls the **AllocBuffer** method to associate a **BUFFSIZE**-sized data buffer with each I/O request object. The data buffer is allocated using the **VirtualAlloc** function. Using **VirtualAlloc** ensures that the block begins on an even allocation-granularity boundary, which satisfies the requirement of the **FILE_FLAG_NO_BUFFERING** flag: the buffer must begin on an address that is evenly divisible by the volume's sector size.

To issue the initial read requests against the source file, I perform a little trick: I post four **CK_WRITE** I/O completion notifications to the I/O completion port. When the main loop runs, the thread waits on the port and wakes immediately, thinking that a write operation has completed. This causes the thread to issue a read request against the source file, which really starts the file copy.

The main loop terminates when there are no outstanding I/O requests. As long as I/O requests are outstanding, the interior of the loop waits on the completion port by calling CIOCP's **GetStatus** method (which calls **GetQueuedCompletionStatus** internally). This call puts the thread to sleep until an I/O request completes to the completion port. When **GetQueuedCompletionStatus** returns, the returned completion key, **CompletionKey**, is checked. If **CompletionKey** is **CK_READ**, an I/O request against the source file is completed. I then call the **CIOReq**'s **Write** method to issue a write I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file. If **CompletionKey** is **CK_WRITE**, an I/O request against the destination file is completed. If I haven't read beyond the end of the source file, I call **CIOReq**'s **Read** method to continue reading the source file.

When there are no more outstanding I/O requests, the loop terminates and cleans up by closing the source and destination file handles. Before **FileCopy** returns, it must do one more task: it must fix the size of the destination file so that it is the same size as the source file. To do this, I reopen the destination file without specifying the **FILE_FLAG_NO_BUFFERING** flag. Because I am not using this flag, file operations do not have to be performed on sector boundaries. This allows me to shrink the size of the destination file to the same size as the source file.

```
Module: FileCopy.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
#include "stdafx.h"
#include "Resource.h"
// Each I/O request needs an OVERLAPPED structure and a data buffer
class CIOReq : public OVERLAPPED {
public:
  CIOReq() {
    Internal = InternalHigh = 0;
    Offset = OffsetHigh = 0;
    hEvent = NULL;
    m_nBuffSize = 0;
    m_pvData = NULL;
  }
  ~CIOReq() {
    if (m_pvData != NULL)
      VirtualFree(m_pvData, 0, MEM_RELEASE);
  }
  BOOL AllocBuffer(SIZE_T nBuffSize) {
    m_nBuffSize = nBuffSize;
    m_pvData = VirtualAlloc(NULL, m_nBuffSize, MEM_COMMIT, PAGE_READWRITE);
```

return(m_pvData != NULL);

}

```
BOOL Read(HANDLE hDevice, PLARGE_INTEGER plioffset = NULL) {
      if (pli0ffset != NULL) {
         Offset
                  = pli0ffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
      return(::ReadFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }
   BOOL Write(HANDLE hDevice, PLARGE_INTEGER plioffset = NULL) {
      if (pli0ffset != NULL) {
         Offset
                  = pli0ffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
     }
      return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }
private:
   SIZE_T m_nBuffSize;
   PVOID m_pvData;
};
```

```
#define BUFFSIZE (64 * 1024) // The size of an I/O buffer
#define MAX_PENDING_IO_REQS 4 // The maximum # of I/Os
```

```
// The completion key values indicate the type of completed I/0.
#define CK_READ 1
#define CK_WRITE 2
```

```
BOOL FileCopy(PCTSTR pszFileSrc, PCTSTR pszFileDst) {
```

```
BOOL f0k = FALSE; // Assume file copy fails
LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst;
```

```
try {
```

```
{
    // Open the source file without buffering & get its size
    CEnsureCloseFile hFileSrc = CreateFile(pszFileSrc, GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
        FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);
    if (hFileSrc.IsInvalid()) goto leave;
```

```
// Get the file's size
GetFileSizeEx(hFileSrc, &liFileSizeSrc);
```

```
// Nonbuffered I/O requires sector-sized transfers.
// I'll use buffer-size transfers since it's easier to calculate.
liFileSizeDst.QuadPart = chROUNDUP(liFileSizeSrc.QuadPart, BUFFSIZE);
// Open the destination file without buffering & set its size
CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
   0, NULL, CREATE_ALWAYS,
   FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hFileSrc);
if (hFileDst.IsInvalid()) goto leave;
// File systems extend files synchronously. Extend the destination file
// now so that I/Os execute asynchronously improving performance.
SetFilePointerEx(hFileDst, liFileSizeDst, NULL, FILE_BEGIN);
SetEndOfFile(hFileDst);
// Create an I/O completion port and associate the files with it.
CIOCP iocp(0);
iocp.AssociateDevice(hFileSrc, CK_READ); // Read from source file
iocp.AssociateDevice(hFileDst, CK_WRITE); // Write to destination file
// Initialize record-keeping variables
CIOReq ior[MAX_PENDING_IO_REQS];
LARGE_INTEGER liNextReadOffset = { 0 };
int nReadsInProgress = 0;
int nWritesInProgress = 0;
// Prime the file copy engine by simulating that writes have completed.
// This causes read operations to be issued.
for (int nIOReq = 0; nIOReq < _countof(ior); nIOReq++) {</pre>
   // Each I/O request requires a data buffer for transfers
   chVERIFY(ior[nIOReg].AllocBuffer(BUFFSIZE));
  nWritesInProgress++;
   iocp.PostStatus(CK_WRITE, 0, &ior[nIOReq]);
}
// Loop while outstanding I/O requests still exist
while ((nReadsInProgress > 0) || (nWritesInProgress > 0)) {
   // Suspend the thread until an I/O completes
   ULONG_PTR CompletionKey;
   DWORD dwNumBytes;
   CIOReg* pior;
   iocp.GetStatus(&CompletionKey, &dwNumBytes, (OVERLAPPED**) &pior, INFINITE);
   switch (CompletionKey) {
   case CK_READ: // Read completed, write to destination
      nReadsInProgress--;
      pior->Write(hFileDst); // Write to same offset read from source
      nWritesInProgress++;
      break;
```

```
case CK_WRITE: // Write completed, read from source
          nWritesInProgress--;
          if (liNextReadOffset.QuadPart < liFileSizeDst.QuadPart) {</pre>
             // Not EOF, read the next block of data from the source file.
             pior->Read(hFileSrc, &liNextReadOffset);
             nReadsInProgress++;
             liNextReadOffset.QuadPart += BUFFSIZE; // Advance source offset
          }
          break;
       }
     }
     fOk = TRUE;
     }
  leave:;
  }
  catch (...) {
  }
  if (f0k) {
     // The destination file size is a multiple of the page size. Open the
     // file WITH buffering to shrink its size to the source file's size.
     CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
       0, NULL, OPEN_EXISTING, 0, NULL);
     if (hFileDst.IsValid()) {
       SetFilePointerEx(hFileDst, liFileSizeSrc, NULL, FILE_BEGIN);
       SetEndOfFile(hFileDst);
     }
  }
  return(f0k);
}
BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM 1Param) {
  chSETDLGICONS(hWnd, IDI_FILECOPY);
  // Disable Copy button since no file is selected yet.
  EnableWindow(GetDlgItem(hWnd, IDOK), FALSE);
  return(TRUE);
}
void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {
```

```
TCHAR szPathname[_MAX_PATH];
```

```
switch (id) {
   case IDCANCEL:
     EndDialog(hWnd, id);
     break;
   case IDOK:
     // Copy the source file to the destination file.
     Static_GetText(GetDlgItem(hWnd, IDC_SRCFILE),
        szPathname, sizeof(szPathname));
     SetCursor(LoadCursor(NULL, IDC_WAIT));
     chMB(FileCopy(szPathname, TEXT("FileCopy.cpy"))
        ? "File Copy Successful" : "File Copy Failed");
     break;
  case IDC PATHNAME:
     OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
     ofn.hwndOwner = hWnd;
     ofn.lpstrFilter = TEXT("*.*\0");
     lstrcpy(szPathname, TEXT("*.*"));
     ofn.lpstrFile = szPathname;
     ofn.nMaxFile = _countof(szPathname);
     ofn.lpstrTitle = TEXT("Select file to copy");
     ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
     BOOL fOk = GetOpenFileName(&ofn);
     if (f0k) {
        // Show user the source file's size
        Static_SetText(GetDlgItem(hWnd, IDC_SRCFILE), szPathname);
        CEnsureCloseFile hFile = CreateFile(szPathname, 0, 0, NULL,
           OPEN_EXISTING, 0, NULL);
        if (hFile.IsValid()) {
           LARGE_INTEGER liFileSize;
           GetFileSizeEx(hFile, &liFileSize);
           // NOTE: Only shows bottom 32 bits of size
           SetDlgItemInt(hWnd, IDC_SRCFILESIZE, liFileSize.LowPart, FALSE);
        }
     }
     EnableWindow(GetDlgItem(hWnd, IDOK), f0k);
     break;
  }
}
```

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

```
switch (uMsg) {
chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
chHANDLE_DLGMSG(hWnd, WM_COMMAND, Dlg_OnCommand);
}
return(FALSE);
```

}

}

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

```
DialogBox(hInstExe, MAKEINTRESOURCE(IDD_FILECOPY), NULL, Dlg_Proc);
return(0);
```

Index

Symbols

"." or ".." characters in pszDLLPathName, 557 % (percent signs), 82 \\?\, pathname providing with, 294 =::=::\ string, 77

Numbers

0 priority, for zero page thread, 190 0 wait, of mutexes, 268 0xfd value, 24 16-bit applications, porting to Win32 easily, 612 16-bit Windows applications running in, 94 backward compatibility with, 17 first function designed for, 85 32-bit applications, 397, 398 32-bit error code number, translating, 4 32-bit error code set, returning, 4 32-bit processes address space for, 371 running under 64-bit operating system, 94 32-bit Windows kernel, 372 64-bit applications accessing full user-mode partition, 375 running on 64-bit Windows, 398 64-bit processes, address space for, 371 64-bit values, 299 64-bit Windows 8-TB user-mode partition, 375 fully supporting AWE, 442 getting 2-GB user-mode partition, 374 kernel getting room, 373 64-bit Windows kernel, 372 64-KB boundary, 376

Ą

abandoned mutexes. 267 AbnormalTermination intrinsic function, 671, 672 abort function, 705 ABOVE_NORMAL_PRIORITY_CLASS, 95, 191 above normal priority class, 189 above normal thread priority, 190, 193 absolute time, 347 abstract layer, 188 AC (alignment check) flag, 391 AcceptEx, 291 access control entry (ACE), 122 access flags, 293 access masks, 62 AccessChk tool, 123 ACE (access control entry), 122 AcquireSRWLockExclusive function, 224, 234 AcquireSRWLockShared function, 225, 236

active processes, switching, 607 ActiveProcesses member, 138 ActiveProcessLimit member, 131 AddElement function, 231 addFileType parameter, 745 address space determining state of, 408-417 fragmentation, 526 modules in, 539 in a process, 67 regions, 375 regions reserving, 376, 419 reserved for thread's stack, 451 separate for each process, 605 total number of bytes private in, 407 Address Space Layout Randomization (ASLR), 417 address space map for a process, 383-387 showing regions and blocks, 388-390 address space sandbox, running applications in, 375 address window, 439, 440 Address Windowing Extensions. See AWE AddSIDToBoundaryDescriptor function, 59 AddText helper function, 238 AddVectoredContinueHandler function, 727 AddVectoredExceptionHandler function, 726 administrative tasks, executing, 116 Administrator account checking if application is running, 117 debugging process, 116 users logging on to Windows with, 110 Administrator SID, checking for, 118 Advanced Local Procedure Call (ALPC) blocked, 714 blocking thread execution, 710 tracked by WCT, 281 AdvAP132.dll. 537 AeDebug registry subkey, 713, 715 affinities, 203-206 of processes, 83 affinity masks, 204 Affinity member, 131 affinity restriction, changing, 130 alertable I/O, 315-320 bad and good of, 318 using, 310 alertable state calling thread in an, 260 placing thread in, 319 thread entering, 326 thread in, 316 thread waiting in, 714 _aligned_malloc function, 210

alignment check, 391 alignment errors, 258 Alignment Fixups/sec counter, 393 allocate declaration, 471 Allocated Ranges edit control, 506 AllocateUserPhysicalPages function, 441 allocation granularity, 376, 483 allocation granularity boundary regions beginning on, 375 regions reserved on, 420 AllocationBase member, 409 AllocationProtect member, 409 AllocConsole function, 94 ALPC. See Advanced Local Procedure Call (ALPC) /analyze switch, 14 anonymous pipe, 290, 291 ANSI functions, 15-17, 520 ANSI strings, 15, 16 ANSI versions of DLLs, 17 of entry-point functions, 150 of functions, 22 APC (asynchronous procedure call) queue, 315, 316, 318 APC entries, waitable timers queuing, 260-261 APCFunc function, 319 API hooking example, 634 manipulating module's import section, 636-639 overwriting code, 635 API macros in WindowsX.h, 776 AppCompat.txt file, 736 Append Method, 270 application code, reading or writing to kernel mode partition, 375 "Application has stopped working" dialog box, 664 Application Instances sample application (17-AppInst.exe), 472-474 application recovery, support for, 756-757 application restart, flags restricting, 755 ApplicationRecoveryFinished function, 756 ApplicationRecoveryInProgress function, 756 applications allocating, 439 converting to Unicode-ready, 26 determining version of Windows, 85 developing using Unicode, 17 directory forcing loader to always check, 585 DLLs extending features of, 537 forcing keystrokes into, 278 improving load time, 551 loading DLL and linking to symbol, 553-554 localizing, 11 mapping code and data into, 465 monitoring kernel objects used by, 41 paging file increasing RAM, 378 running, 374, 375, 540, 571 shutting down, 253 subclassing windows created in other processes, 607 theming not supporting by default, 766

tuning, 408 types of, 68, 69 users starting, 99 windows locating on screen, 97 writing, 68-104 AppName.local file, 585 aRAMPages parameter, 442 arbitrary wait, of mutexes, 268 archive file, indicating, 297 _argc variable, 73, 76 __argv variable, 73, 76 ASCII Unicode character set, 13 asInvoker value, 114 Ask Me To Check If Problem Occurs option, 712 ASLR (Address Space Layout Randomization), 417 assertion dialog boxes, 20 AssignProcessToJobObject function, 136 AssociateDevice member function, 332 AssociateDeviceWithCompletionPort function, 322 asynchronous device I/O, 277 basics of, 305-310 caveats, 307-309 asynchronous I/O operations, 304 asynchronous I/O requests issuing, 327 queuing, 305 system handling, 316 asynchronous procedure call entries. See APC entries asynchronous session, opening, 284 ATL class library, 18 atomic access of sophisticated data structures, 215 for threads, 208-213 atomic manipulation, 217 atomic test and set operation, 212 atomically-manipulated object set, 238 attribute(s) associated with sections, 467 changing for sections, 471 passing with lpAttributeList, 99 attribute flags, 297 attribute keys, in STARTUPINFOEX, 100 attribute list, 100-102 Attribute parameter, 101 Auto value, 715 AutoExclusionList key, 715 automatic application restart, 755-757 auto-reset event(s) automatically reset to nonsignaled state, 249, 253 calling PulseEvent, 251 demonstrating use of, 252 signaling, 247 using instead of manual-reset event, 250 auto-reset timer, signaling, 257 AvailPageFile, 407 AvailPhys, 406 AvailVirtual, 407 AWE (Address Windowing Extensions), 439, 441 AWE application (15-AWE.exe), 442

В

background colors, used by child's console window, 97 background process, 195 background processing state (BPS), 365 Backup/Restore File and Directories privileges, 296 bAlertable flag, 279 bAllUser parameter, 737 base address for executable file's image, 74 for memory-mapped files, 496-497 of region, 386 returning executable or DLL's file image, 74 /BASE option, 464 base priority level, 194 /BASE:address linker switch, 74 BaseAddress member, 409 ___based keyword, 497 based pointers, 497 BaseThreadStart function, 706 basic limit restriction, 129 basic limits, superset of, 132 basic UI restrictions, 129 batch, detecting end of, 346 Batch application (11-Batch.exe), 342-346 bCancelPendingCallbacks parameter of WaitForThreadpoolIoCallbacks function, 355 of WaitForThreadpoolWorkCallbacks, 342 BCD (boot configuration data), 206 configuring, 374 current value of parameters, 374 programmatic configuration of, 206 BCDEdit.exe, 374 beginthread function, 168 _beginthreadex function, 151, 160 calling CreateThread, 162, 168 compared to _beginthread, 168 overriding, 451 problems with, 764 source code for, 161 using, 763 BELOW_NORMAL_PRIORITY_CLASS, 95, 191 below normal priority class, 189 below normal thread priority, 190, 193 bFirstInTheList parameter of AddVectoredContinueHandler, 727 of AddVectoredExceptionHandler, 726 bGrant parameter, 134 binary (.exe or DLL) file, supporting all languages, 26 Bind.exe utility, 593, 594 BindImageEx function, 593 binding modules, 592-595 bInheritHandle member, 44 bInheritHandle parameter of DuplicateHandle, 62 of Open* functions, 51 bInheritHandles parameter, 44, 91-92 bInitialOwned parameter, 265

bInitialOwner parameter, 266 bit 29, of error code, 7 bit flags, 133 bitmask, indicating CPUs for threads, 203 bitwise mask, of Label ACE, 124 BlkSize member, 411 block of memory allocating from heap, 525 allocating properly aligned, 210 guarding, 265 blocks defining, 387 displaying inside regions, 388 number contained within region, 411 within reserved region, 387 Blue Screen of Death, 713 bManualReset parameter of CreateEvent, 248 of CreateWaitableTimer, 257 BOOL return type, 3 boot configuration data. See BCD boundary descriptor associating SID with, 59 creating, 54 protecting namespace name itself, 53 BPS_CONTINUE state, 366 BPS_DONE state, 366 BPS_STARTOVER state, 366 bReadDone, 311 breakpoint, forcing, 765 bResume parameter, 259 bRgnIsAStack member, 411 .bss section, 467, 470 buffer manipulation functions, 27 buffer overrun errors, 11 buffer overruns, automatically detecting, 27 BUFFSIZE, 331 build environment, for sample applications in this book, 761-767 build number, of current system, 87 Button_SetElevationRequiredState macro, 118 bWaitAll parameter, 245, 246

C

C/C++ compiler defining built-in data type, 13 for IA-64, 393 C/C++ language comma (,) operator, 700 never calling CreateThread, 151 programs, error checking in, 372 resources not destroyed by ExitThread, 154 C/C++ programming, 19 C/C++ run-time global variables, 73 library functions, 160, 168 startup code, 105 C/C++ run-time, continued startup functions, 69, 71 startup routine, 90 C/C++ run time, global variables, 71 C/C++ run-time libraries DllMain function, 570 initializing, 69, 570 multithreaded version, 167 not originally designed for multithreaded applications, 160 present in single address space, 539 shipping with Microsoft Visual Studio, 159 stack-checking function, 456-457 startup assistance from, 570 startup code, 146 supplied in DLL, 167 using TLS, 597 C/C++ source code module, 541 C compiler, mangling C functions, 546 C header file, 14 C language, char data type, 13 C run-time function, 17 C run-time library invented around 1970, 160 with multithreaded applications, 160 prefixing identifiers with underscores, 18 secure string functions in, 18-25 string manipulation functions, 19 Unicode and ANSI functions in, 17 C run-time memory allocation functions, 71 C run-time startup code, 105 C++ heaps with, 528-531 with message crackers, 773 C++ applications, using C++ exceptions, 727 C++ classes avoiding exporting, 542 creating, 443 C++ compilers. See also Microsoft Visual C++ compiler C4532 warning, 690 equivalent structured exception handling, 728 mangling function and variable names, 544 running from command shell, 93 C++ exception handling compared to structured exception handling, 660 only when writing C++ code, 727 C++ exceptions, vs. structured exceptions, 727 C++ object address, storing in hEvent, 307 C++ objects, destroying, 104, 106, 154 C++ throw statement, 728 C++ try block, compiler generating SEH _try block for, 728 C2 security requirements, 332 C4532 warning, 690 CACHE_DESCRIPTOR structure, 214 Cache field, 214 cache flags, in CreateFile, 295-297 cache lines, 214-215

cache manager, buffering data, 295 CAddrWindow class, 443 CAddrWindowStorage class, 443 call tree, 453 callback environment, initializing, 357 callback functions defining, 346 registering to WER, 756 required by alertable I/O, 318 callback instance, functions applying to, 356 callback method, 355 callback routine, 260 callback termination functions, 355 CallbackMayRunLong function, 356 calling thread checking signaled state of kernel objects, 244 as forever blocked, 244 placing in wait state, 221 putting to sleep, 325 transitioning from user mode to kernel mode, 241 _callthreadstartex function, 164, 165 CancelIo function, 309 CancelIoEx function, 310 CancelSynchronousIo function, 304-305 CancelThreadpoolIo function, 354 CancelWaitableTimer function, 259 CAPIHook C++ class, 640 case sensitivity, of reserved keywords, 53 casting, 773 catch C++ keyword, 660 catch handlers, 726 Cb (count of bytes), 22 cb member, 575 cb parameter, 32 cb value, 97 cbMultiByte parameter, 28 cbReserved2 value, 98 cbSize parameter, 101 cbStackSize parameter, 151 Cch, in method names, 22 cch1 parameter (CompareString), 25 cch2 parameter (CompareString), 25 cchValue parameter, 81 cchWideChar parameter, 28 ccPath parameter, 560 CD-ROMs, image files copying to RAM, 380 CellData matrix, 505 CELLDATA structure, 504 committing storage for single, 425 implementing, 424 changed data, preserving, 486 ChangeDisplaySettings function, 133 char data types, 13 character encodings, 12-13 character sets, 12 characters, 26 chBEGINTHREADEX macro, 161, 763, 765 _chdir C run-time function, 85

CheckInstances function, 54, 59 CheckTokenMembership function, 118 chHANDLE DLGMSG macro, 766 child control macros, 776 child process(es), 108-110 current directories, inheriting parent's current, 85 environment variables, inheriting, 81 error mode flags, inheriting, 392 getting copy of parent's environment block, 81 handle table, example, 45 kernel object access, documenting expectation of, 45 kernel object, handle value of, 46 parent console, inheriting, 46 parent termination, continuing to execute, 125 primary thread, controlling code executed by, 634 priority class, changing, 191 process affinity, inheriting, 203 root directory, current directories defaulting to, 85 running detached, 110 spawning, 43-45, 108 chINRANGE macro, 763 chmalloc, 27 chMB macro, 765 chMSG macro, 763 chSETDLGICONS macro, 766 chSize parameter, 82 chVERIFY macro, 765 CIOCP class, 332 CIOReq C++ class, in FileCopy sample application, 307 CK_FILE completion key, 322 CK_WRITE completion key, 332 CK_WRITE I/O completion notifications, 332 class, allocating instance of, 528 cleanup groups, 358, 360 guaranteed without using try-finally, 686 localizing code in one place, 669 of mutexes vs. critical sections, 268 client machines, communicating with many servers, 261 client session, namespace in Terminal Services, 51 client thread(s) created by Queue, 229 as WriterThread, 232-234 Client Threads list box, 269 client/server threads, stopping, 236 Clipboard, preventing processes from reading, 133 CloseHandle function, 39, 103 calling twice, 487 clearing out entry in process' handle table, 40 closing devices, 292, 354 closing job object handle, 128 committed storage, reclaiming, 500 decrementing parent thread object's usage count, 171 destroying all kernel objects, 253 event kernel object is no longer required, 249 forgetting to call, 40 passing handle of completion port, 331

process' statistical data, no longer interested in, 108 protected handle, thread attempting to close, 47 ClosePrivateNamespace function, 60 closesocket, 292 CloseThreadpool function, 357, 360 CloseThreadpoolCleanupGroup function, 360 CloseThreadpoolCleanupGroupMembers function, 359 CloseThreadpoolIo function, 355 CloseThreadpoolTimer function, 347, 348 CloseThreadpoolWait function, 353 CloseThreadpoolWork, 342 CloseThreadWaitChainSession, 284 CMD.EXE command prompt, 68 CMMFSparse object, 508 CmnHdr.h header file, 761-767 *cmp string comparison functions, 25 code injecting with CreateProcess, 633 maintaining with termination handlers, 672 running within catch or finally blocks, 664 Code Analysis option, 14 code page associated with newly converted string, 28 number associated with multibyte string, 27 code points. See Unicode code points code policies, possible, 124 code section, 466 code-point comparison, 25 coherence, memory-mapped files and, 495 COM easily integrating with, 26 initialization and calls, 281 interface methods, 17 object DLLs, 585 ComCtl32.dll, 537 ComDig32.dll, 537 comma (,) operator, in exception filter, 700 command console, regaining control of, 305 command line parsing into separate tokens, 76 of process, 76, 121 command shell. 192 command-line buffer, copying, 76 CommandLineToArgvW function, 76, 77 commit argument, with /STACK switch, 151 committed pages, disabling caching of, 382 committing physical storage, 376, 424 communication flags, in CreateFile, 295-298 communication protocol building complete two-way, 639 related to uploading problem reports, 745 CompareString function, 24 CompareString(Ex) function, 24, 27 CompareStringOrdinal function, 24, 25, 27 comparison, linguistically correct, 24 compatibility rules, reorganized by operating system, 114

compiler differences among implementations, 659 guaranteeing finally block execution, 662 vendors following Microsoft's suggested syntax, 659 completion key value set, 141 completion notification, calling thread responding to, 318 completion port. See I/O completion port(s) completion routine, 315 CompletionKey parameter as CK_READ, 333 of PostQueuedCompletionStatus, 330 CompletionRoutine function, 315 component protection, creating heaps for, 521 components, of a process, 67 computers, supporting suspend and resume, 259 concurrent model, 320 CONDITION_VARIABLE_LOCKMODE_SHARED, 227 condition variables, 227-240 in conjunction with lock, 228 signaling, 237 Configuration Properties section, 761 consent dialog box, customizing, 746 consent parameter, 746 Consent setting, 739 console application, forcing full-screen mode, 98 console, use of, 290, 291 console window, 94 buffer identifying, 98 specifying width and height of child's, 97 console-based applications. See CUI-based process constant string, copying to temporary buffer, 90 ConstructBlkInfoLine, 417 constructors for C++ class objects, 71 ConstructRgnInfoLine, 417 ConsumeElement function, 234 container applications, 715 content indexing service, 297 contention, 223, 226 context(s) of thread, 157 types of, 185 CONTEXT_CONTROL section, 185 CONTEXT_DEBUG_REGISTERS section, 185 CONTEXT_EXTENDED_REGISTERS section, 185 CONTEXT_FLOATING_POINT section, 185 CONTEXT_FULL identifier, 186 CONTEXT INTEGER section, 185 CONTEXT_SEGMENTS section, 185 CONTEXT structure, 183-187, 699 changing members in, 186 ContextFlags member, 185 as CPU-specific, 183 defined in WinNT.h header file, 157 pointing to, 702 role in thread scheduling, 183 saving, 699 sections, 185

stack pointer register, 157 for x86 CPU, 183 context structure, of every thread, 173 context switch, 173, 607 ContextFlags member, 185, 186 ContextRecord member, 702 continue handler function registering, 727 removing from internal list, 727 returning EXCEPTION_CONTINUE_EXECUTION, 727 returning EXCEPTION_CONTINUE_SEARCH, 727 ContinueDebugEvent, 176 control window messages, 611 convenience data types, 14 ConvertStringSecurityDescriptorToSecurityDescriptor function, 59 ConvertThreadToFiber function, 362, 365 ConvertThreadToFiberEx function, 362 cookie, storing stack state as, 457 copy-on-write mechanism, 382 of memory management system, 466 pages backing system's paging file, 588 turning off, 468 copy-on-write protection, 382 core components, isolating from malware, 52 core functions, requiring Unicode strings, 15 corruption, of internal heap structure, 524 count of bytes. See Cb Count of characters. See Cch Counter application (12-Counter.exe), 365-367 countof macro defined in stdlib.h, 19 getting Cch value, 22 _countof (szBuffer), instead of sizeof (szBuffer), 26 counts, functions returning, 3 CPU(s). See also x86 CPUs aligned data, accessing properly, 391 cache coherency, communicating to maintain, 225 cache line, changing bytes in, 214 cache line, determining size of, 214 exceptions, raising, 679 indicating active, 396 limiting number system will use, 206 load balancing threads over multiple, 68 number in machine, 396 process address spaces for different, 377 registers for each thread, 157 specifying subset of, 131 tasks, giving varied to perform, 146 threads, controlling running on, 203 threads, selecting, 206 CPU architectures, 373 CPU cache lines. See cache lines CPU time accounting information, 130 exceeding allotted, 142 scheduling for threads, 67

CPU-dependent code, writing, 186 CPU-specific code, writing, 633 COueue C++ class, 229 constructor, 270 thread-safe, 269 CrashDmp.sys, 713 CREATE_ALWAYS value, 294 CREATE_BREAKAWAY_FROM_JOB flag, 95, 136 CREATE_DEFAULT_ERROR_MODE flag, 84, 94 CREATE_EVENT_INITIAL_SET flag, 248 CREATE_EVENT_MANUAL_RESET flag, 248 CREATE_FORCEDOS flag, 94 CREATE_MUTEX_INITIAL_OWNER, 265 CREATE_NEW_CONSOLE flag, 94 CREATE_NEW_PROCESS_GROUP flag, 94 CREATE_NEW value, 294 CREATE_NO_WINDOW flag, 94 CREATE SEPARATE WOW VDM flag, 94 CREATE_SHARED_WOW_VDM flag, 94 CREATE_SUSPENDED flag, 174 checking for, 175 correct use of, 153 of CreateProcess fdwCreate, 93 passing to CreateThread, 157, 193 when calling CreateProcess, 136 CREATE_UNICODE_ENVIRONMENT flag, 94 Create* functions, 51 CreateBoundaryDescriptor function, 59 CreateConsoleScreenBuffer function, 291 CreateDesktop function, 133 CreateEvent function, 5, 248, 312 CreateEventEx function, 248 CreateFiber function, 362 CreateFiberEx function, 363 CreateFile function, 291-294, 478 cache flags, 295-297 checking return value of, 500 FileReverse calling, 488 not calling methods asynchronously, 304 returning INVALID_HANDLE_VALUE, 39 specifying exclusive access to file, 496 CreateFileMapping function, 479, 488, 500 creating file mapping, 34 passing INVALID_HANDLE_VALUE, 499 CreateFileMappingNuma function, 485 CreateFileW function, 294 CreateIcon function, 37 CreateIoCompletionPort function, 321, 322 CreateMailslot function, 291 CreateMutex function, 49, 265 CreateNamedPipe function, 291 CreateNewCompletionPort function, 321 CreatePipe function, 291 CreatePrivateNamespace function, 59, 60 CreateProcess function, 44, 89-104, 136, 464 checking for CREATE_SUSPENDED flag, 175 initializing suspend count, 175 invoking debugger process, 714

more control over cursor, 99 spawning WerFault.exe, 710 CreateProcessWithLogonW function, 17 CreateRemoteThread function, 621, 623, 625 CreateSemaphore function, 263 CreateSemaphoreEx function, 49 CreateThread function, 150-153, 157 called by beginthreadex, 162 checking for CREATE_SUSPENDED flag, 175 compared to CreateRemoteThread, 621 creating new thread with normal priority, 193 initializing suspend count, 175 instead of _beginthreadex, 168 never needing to call, 340 prototype for, 764 thread stack storage initially committed, 451 CreateThreadpool function, 356 CreateThreadpoolCleanupGroup function, 359 CreateThreadpoolIo function, 354 CreateThreadpoolTimer function, 262, 346, 348 CreateThreadpoolWait function, 352 CreateThreadpoolWork function, 341 CreateToolhelp32Snapshot function, 177 CreateWaitableTimer function, 256 CreateWellKnownSid function, 118 CreateWindow function, 97 CreateWindowEx function, 15 CreateWindowExW function, 16 creation time, returned by GetThreadTimes, 179 critical sections, 217-224, 238 changing spin count for, 223 compared to mutexes, 265, 267 compared to SRWLock, 225 ensuring access to data structures, 218 minimizing time spent inside, 239 reading volatile long value, 226 spinlocks incorporated on, 222 tracked by WCT, 281 using interlocked functions, 219 CRITICAL_SECTION structure, 218, 220 critical-error-handler message box, not displaying, 83 CriticalSectionTimeout data value, 221 .CRT section, 470 _CrtDumpMemoryLeaks function, 72 crtexe.c file, 71 _CrtSetReportMode, 20 CSparseStream C++ class, 508 CSystemInfo class, 443 CToolhelp C++ class, 120 Ctrl+Break, 94 Ctrl+C, 94, 304 CUI applications, 68-69 CUI subsystem, selecting, 71 CUI-based process, output to new console window, 93 current drives and directories, of process, 84 current priority level, of thread, 194 current resource count, in semaphore kernel object, 262-263

786 Currently Running Fiber field

Currently Running Fiber field, 365 CurrentVersion\Windows\ key, 608 custom prefix, for named objects, 53 custom problem report, generating, 748 customized report description, in WER console, 742 Customized WER sample application (26-CustomizedWER.exe), 748–751

CustomUnhandledExceptionFilter function, 749, 750 CVMArray templated C++ class, 718 CWCT C++ class, 282 __CxxUnhandledExceptionFilter, 705, 706, 708

D data

accessing via single CPU, 215 aligning, 391-393 communicating among processes, 463 flushing to devices, 303 passing from one thread to another, 599 protecting from threads accesses, 230 sharing using memory-mapped files, 109, 498 data blocks, 738, 739 data buffer, not moving or destroying, 309 Data Execution Prevention. See DEP data files accessing, 463 mapping to address space, 380, 476 data misalignment exceptions, 392 .data section, 467, 470 data structure, example of poorly designed, 214 deadlock(s) avoiding, 238 dangers faced when synchronizing content, 237 detecting, 281-287 example, 569 issues when stopping threads, 236-238 prevented with WaitForMultipleObjects, 247 typical, 282 DEBUG builds, 72 debug event, 279 Debug menu option, in Task Manager, 715 DEBUG_ONLY_THIS_PROCESS flag, 93 DEBUG_PROCESS flag, 93 .debug section, 470 DebugActiveProcess function, 714 DebugActiveProcessStop function, 633 DebugApplications, adding as new subkey, 715 DebugBreak function, 765 debugger calling SetEvent, 715 connecting to any process, 715 dynamically attaching, 713 exceptions and, 729-732 injecting DLL as, 633 notifying of unhandled exception, 708 positioning at exact location, 735 starting from command shell, 128 debugging built into Windows operating system, 279

elevated/filtered processes, 116 going across process boundaries, 605 just-in-time, 713-715 monitoring thread's last error code, 5 one instance of application, 467 unpredictability of first-in/first-out, 247 debugging-related exceptions, 696 DebugSetProcessKillOnExit function, 633 declaration, of mutexes vs. critical sections, 268 __declspec(align(#)) directive, 214 __declspec(dllexport) modifier, 544 in executable's source code files, 544 in header file, 548 __declspec(dllimport) keyword, 548 _declspec(thread) prefix, 602 decommitting physical storage, 377, 426 .def file, 547 default debugger, WER locating and launching, 713 default heap, 519, 520 DefaultConsent DWORD value, 710 DefaultSeparate VDM value, 94 #define directives, creating set of macros, 773 DefWindowProc function, 775 Delay Import section, 573 /DELAY linker switch, 572 Delay Loaded DLLs option, setting, 572 delay-load DLLs, 571-582 unloading, 575 delay-load hook function, 577 /DELAYLOAD linker switch, 572, 573 DelayLoadApp application, 576–582 DelayLoadDllExceptionFilter function, 574 _delayLoadHelper2 function, 573, 574 DelayLoadInfo structure, 574 /Delay:nobind switch, 575 /Delay:unload linker switch, 575 delete operator, 528 delete operator function, 530 DeleteBoundaryDescriptor function, 59, 60 DeleteCriticalSection function, 220, 223 DeleteFiber function, 364, 366 DeleteProcThreadAttributeList method, 102 Denial of Service (DoS) attacks, 53 DEP (Data Execution Prevention) detecting code from a nonexecutable memory page, 701 enabled, 381 dependency loop, 563 DependencyWalker utility, 576 deprecated functions, replacing, 19 desktop(s) naming to start application, 97 preventing processes from creating or switching, 133 Desktop Item Position Saver utility, 610-616 destination buffer, 23 destination file, size of, 332, 333 DestroyThreadpoolEnvironment, 358, 360 destructors, 72 detached process, 110

DETACHED_PROCESS flag, 93 Detours hooking API, 641 device(s), 290 associating with I/O completion ports, 322-324 communicating with asynchronously, 297 flushing data to, 303 opening and closing, 290-298 device driver executing I/O requests out of order, 308 running in kernel mode, 190 device I/O canceling queued requests, 309 as slow and unpredictable, 305 device kernel objects, signaling, 310, 311-312 device objects in nonsignaled state, 277 as synchronizable kernel objects, 277 device-sharing privileges, 293 dialog boxes, message crackers with, 766 DialogBox, calling, 252 .didata section, 470, 573 DIPS.exe application, 610-616 directories, 290 obtaining and setting current, 84 opening, 291 tracking for multiple drives, 84 DisableThreadLibraryCalls function, 569, 570 DisassociateCurrentThreadFromCallback function, 356 disk defragmenting software, 147 disk image, 486 disk space, looking like memory, 377 display resolution, 611 DliHook function, 576 DLLs (dynamic-link libraries) building, 540 calling LoadLibrary to load, 464 containing functions called by .exe, 464 as cornerstone of Microsoft, 537 created and implicitly linked by applications, 540 creating, 17, 546 delay-loading, 571-582 entry-point function, 562-571 explicitly loaded, 564 exporting variables, functions, or C++ classes, 542 in full 4-TB user-mode partition, 375 functions for specialized tasks, 537 injecting as debugger, 633 injecting using registry, 608-609 injecting using remote threads, 621-633 injecting using Windows hooks, 609-616 injecting with Trojan DLL, 633 in large 2+ GB user-mode partition, 374 loading at preferred base addresses, 464 loading from floppy disk, 380 loading from high-memory addresses, 590 making difficult for hackers to find, 417 mapping into processes using User32.dll, 608 preventing processes from dying, 564

preventing threads from dying, 567 process address space and, 538-539, 564 reasons for using, 537 receiving no notification with TerminateThread, 155 reserving region of address space for, 464 serving special purposes, 538 spreading out loading, 571 telling to perform per-thread cleanup, 567 unmapping, 564, 610 uploading when callback function returns, 355 usage count, 559 using DllMain functions to initialize, 563 .dll extension, 584 DLL functions, 29-30 DLL image file, producing, 541 DLL injection, 605-607 DLL modules allocating and freeing memory, 539 building, 541-546 explicitly loading, 555 explicitly unloading, 558 loading in user-mode partition, 373 rebasing, 586-592 DLL_PROCESS_ATTACH notification, 563, 608 DLL_PROCESS_DETACH notification, 564-566 DLL redirection, 585-586 /DLL switch, 538 DLL_THREAD_ATTACH notification, 566 DLL_THREAD_DETACH notification, 168, 567 DllMain function calling DLL's, 564 C/C++ run-time library and, 570 implementing, 562 mapping file image without calling, 555 serialized calls to, 567-570 DllMain function name, case sensitivity of, 562 _DllMainCRTStartup function calling destructors, 570 calling DllMain function, 570 handling DLL_PROCESS_ATTACH notification, 570 inside C/C++ run time's library file, 570 DLL's file image, mapping, 538, 555 dlp member, of DelayLoadInfo, 574 DOMAIN_ALIAS_RID_ADMINS parameter, 59 DONT_RESOLVE_DLL_REFERENCES flag, 555, 556 DontShowUI value, 710 _dospawn, 98 double-byte character sets (DBCSs), 12 drive-letter environment variables, 85 driver, 305 dt command, in WinDbg, 121 due times, 262 dump files, naming of, 735 DumpBin utility running on executable file, 468 using preferred base addresses, 586 viewing DLL's export section, 545 viewing module's import section, 549-550 dumpType parameter, 745

DUPLICATE_CLOSE_SOURCE flag, 61 DUPLICATE_SAME_ACCESS flag, 61, 62 DuplicateHandle function, 60, 60-63, 64, 170 dw1ConditionMask parameter, 88 dwActiveProcessorMask member, 396 dwAllocationGranularity member, 396 dwBlkProtection member, 411 dwBlkStorage member, 411 dwBuildNumber member, 87 dwBytes parameter (HeapAlloc), 525 dwBytes parameter (HeapReAlloc), 527 dwConditionMask parameter, 88 dwCount parameter, 245 dwCreateFlags parameter, 153 dwCreationDisposition parameter, 294 dwData parameter, 319 dwDesiredAccess parameter accepted by kernel object creation functions, 49 of CreateEventEx. 248 of CreateFile, 293, 478 of CreateMutexEx, 265 of CreateSemaphore, 263 of DuplicateHandle, 61 DuplicateHandle ignoring, 62 of MapViewOfFile, 482 of Open* functions, 50 dwExceptionCode parameter, 703 dwExceptionFlags parameter, 703 dwExitCode parameter of ExitThread, 154 of TerminateThread, 155 dwFileFlags parameter, 745 dwFileOffsetHigh parameter, 483 dwFileOffsetLow parameter, 483 dwFillAttribute value, 97 dwFlags in STARTUPINFO, 97, 98 dwFlags parameter, 23 of CreateEventEx, 248 of CreateMutexEx, 265 of CreateSemaphore, 263 of GetThreadWaitChain, 285 of LoadLibraryEx, 555 with MultiByteToWideChar function, 27 in OpenThreadWaitChainSession, 284 of RegisterApplicationRestart, 755 as reserved. 101 of SetHandleInformation, 47 of UpdateProcThread, 101 of WerRegisterFile, 739 of WerReportAddDump, 745 of WerReportSubmit, 747 of WideCharToMultiByte function, 28 dwFlagsAndAttributes parameter, 294 dwIdealProcessor parameter, 205 dwInitialSize parameter, 525 dwl1ConditionMask parameter, 87 dwLastError member, 574 dwMajorVersion member, 87

dwMask parameter, 46 dwMaximumSize parameter, 525 dwMaximumSizeHigh parameter, 481 dwMaximumSizeLow parameter, 481 dwMilliseconds parameter, 243 of GetQueuedCompletionStatus, 325 of SignalObjectAndWait, 279 of Sleep* functions, 177, 227 of WaitForMultipleObjects, 245 dwMinorVersion member, 87 dwMoveMethod parameter, 301 dwNumberOfBytesToFlush parameter, 486 dwNumberOfBytesToMap parameter, 483 dwNumberOfProcessors member, 396 dwNumBytes parameter, 330 dwOptions parameter, 61 DWORD, fields of, 703 DWORD return type, 3 dwOSVersionInfoSize member, 87 dwPageSize member, 396 dwParamID parameter, 743 dwPingInterval, 756 dwPlatformId member, 87 dwPreferredNumaNode parameter of CreateFileMappingNuma, 485 of MapViewOfFileExNuma, 485 of VirtualAllocExNuma, 421 dwProcessAffinityMask parameter, 203 dwProcessorType member, 396 dwRgnBlocks member, 411 dwRgnGuardBlks member, 411 dwRgnProtection member, 411 dwRgnStorage member, 411 dwShareMode parameter of CreateFile, 293, 478 specifying exclusive access, 496 dwSize parameter, 435 of ReadProcessMemory, 624 of WerRegisterMemoryBlock, 738 of WriteProcessMemory, 624 dwSpinCount parameter, 223 of InitializeCriticalSectionAndSpinCount, 222 of SetCriticalSectionSpinCount, 223 dwStackCommitSize parameter, 363 dwStackReserveSize parameter, 363 dwStackSize parameter, 362 dwThreadAffinityMask parameter, 204 dwTlsIndex parameter, 599 dwTypeBitMask parameter, 88 dwTypeMask parameter, 87, 88 dwX value, 97 dwXCountChars value, 97 dwXSize value, 97 dwY value, 97 dwYCountChars value, 97 dwYSize value, 97 dynamic boosts, 194 dynamic priority range, 194
dynamic TLS, 598–602 /dynamicbase linker switch, 417 dynamic-link libraries. *See* DLLs

E

E_INVALIDARG, 744 .edata section, 470 efficiency, for applications with Unicode, 26 EFLAGS register, 391 element, adding on top of stack, 213 ELEMENT structure of CQueue class, 269 inside CQueue class, 230 elevated account, credentials of, 112 elevated privileges, child process getting, 116 empty stack, 213 emulation layer, for 32-bit applications, 397 EnableAutomaticJITDebug function, 750 encrypted file, 297 end of file marker, 509 end user. See user(s) endless recursion, 151 _endthread function, 168 _endthreadex function, 166 compared to _endthread, 169 using, 154 EnterCriticalSection function, 218, 219-220, 223 /ENTRY switch, 570 -entry:command-line option, 69 entry-point function calling, 71, 146 for every thread, 149 implementing, 562 for Windows application, 69 EnumProcesses function, 119 env parameter, 72, 79 _environ global variable, 73 environment block, 77 containing ANSI strings by default, 94 spaces significant in, 80 environment strings, obtaining from registry keys, 80 environment variables, 77-83 accessing, 72 multiple for process, 84 Environment Variables dialog box, 80 ERANGE, returning, 21 \$err,hr, in Watch window, 5, 6 errno C run-time global variable, 160 defining in standard C headers, 166 internal C/C++ run-time library function, 167 macro, 167 setting, 19 errno_t value, 19, 20 error(s) checking with ReadFile and WriteFile, 308 returning via exceptions, 703 trapping and handling, 683

ERROR_ALREADY_EXISTS, 5 ERROR_CANCELLED, 116 error code(s) 32-bit number with fields, 7 composition of, 697 returned from GetLastError, 308 setting thread's last, 7 Windows functions returning with, 3 ERROR_ELEVATION_REQUIRED, 116 error handling, 659 critical sections and, 223 performed by Windows functions, 3 simplified by termination handlers, 672 using exceptions for, 702 ERROR_INVALID_HANDLE, 39, 331 ERROR_INVALID_USER_BUFFER, 308 ERROR_IO_PENDING, 308 Error Lookup utility in Visual Studio, 6, 9 ERROR_MOD_NOT_FOUND, 557 error mode, 83, 94 ERROR_NOT_ENOUGH_MEMORY, 308 ERROR_NOT_ENOUGH_QUOTA, 308 ERROR_NOT_OWNER, 267 ERROR_OLD_WIN_VERSION, 88 ERROR_OPERATION_ABORTED, 310 error reporting, enabling, 738 ERROR_SUCCESS, 5 ERROR_USER_MAPPED_FILE, 489 ErrorShow sample application, 6, 7-9 European Latin Unicode character set, 13 event(s) allowing threads to synchronize execution, 43 changing to nonsignaled state, 249 changing to signaled state, 249 initializing signaled or nonsignaled, 248 as most primitive of all kernel objects, 247 signaled and immediately nonsignaled, 251 signaling that operation has completed, 247 event handle, creating with reduced access, 248 event kernel objects, 247-253 creating, 248 identifying, 312 signaling, 310, 312-313 synchronizing threads, 249 types of, 247 used by critical sections, 223 EVENT_MODIFY_STATE, 248 __except filter, 728 __except keyword, 679 exception(s), 679, 695-696 debugger and, 696, 729-732 handling, 684 raised by HeapAlloc, 525 system processing, 682 EXCEPTION_ACCESS_VIOLATION, 695, 698, 701 EXCEPTION_ARRAY_BOUNDS_EXCEEDED, 695 EXCEPTION_BREAKPOINT, 696 exception codes, 697, 703

EXCEPTION_CONTINUE_EXECUTION, 681, 691-693, 706, 707 EXCEPTION_CONTINUE_SEARCH, 681, 693-694, 706, 707 filter returning, 574 FilterFunc function returning, 458 EXCEPTION_DATATYPE_MISALIGNMENT, 695 raised by other processors, 258 transforming misalignment fault into, 391 EXCEPTION_EXECUTE_HANDLER, 681, 683-690, 706,707 FilterFunc function returning, 458 returning, 691 triggering global unwind, 709 exception filters committing more storage to thread's stack, 692 debuggee's thread searching for, 730 executed directly by operating system, 680 GetExceptionInformation calling only in, 699 identifiers, 681 producing, 701 return values of top-level, 706 thread searching for, 731 understanding by example, 680-682 writing your own unhandled, 738 EXCEPTION_FLT_DENORMAL_OPERAND, 696 EXCEPTION_FLT_DIVIDE_BY_ZERO, 696 EXCEPTION_FLT_INEXACT_RESULT, 696 EXCEPTION_FLT_INVALID_OPERATION, 696 EXCEPTION_FLT_OVERFLOW, 696 EXCEPTION_FLT_STACK_CHECK, 696 EXCEPTION_FLT_UNDERFLOW, 696 EXCEPTION_GUARD_PAGE, 695 exception handlers added into internal list of functions, 726 executed directly by operating system, 680 registering in special table in image file, 381 syntax for, 679 understanding by example, 680-682 exception handling, 662 EXCEPTION_ILLEGAL_INSTRUCTION, 695 EXCEPTION_IN_PAGE_ERROR, 695 exception information, accessing, 699 EXCEPTION_INT_DIVIDE_BY_ZERO, 696 EXCEPTION_INT_OVERFLOW, 696 EXCEPTION_INVALID_DISPOSITION, 695 EXCEPTION_INVALID_HANDLE, 223, 696 EXCEPTION MAXIMUM PARAMETERS, 704 EXCEPTION_NESTED_CALL, 707 EXCEPTION_NONCONTINUABLE_EXCEPTION, 695,704 EXCEPTION_NONCONTINUABLE flag, 703 always used for C++ exceptions, 728 used by HeapAlloc, 704 EXCEPTION_POINTERS structure, 699 EXCEPTION_PRIV_INSTRUCTION, 695 EXCEPTION_RECORD structure, 699 pointing to, 700, 704 saving, 699

walking linked list, 704 EXCEPTION_SINGLE_STEP, 696 EXCEPTION_STACK_OVERFLOW, 454, 695 ExceptionAddress member, 701 ExceptionCode member, 700 ExceptionFlags member, 700 ExceptionInformation member, 701, 704, 708, 735 ExceptionRecord member of EXCEPTION_POINTERS structure, 700, 704 of EXCEPTION_RECORD structure, 700 exception-related exceptions, 695 Exceptions dialog box, 729, 731 excluded application, 737, 738 exclusive access to ensuring queue, 270 opening binary file for, 556 specifying to file, 496 .exe file building, 540 CreateProcess searching for, 90 executing startup code, 465 loading and executing DLL files, 463 loading from floppy disk, 380 locating, 464 multiple mappings, 467 passing instance of, 73 reserving region of address space for, 464 .exe module. See executable module executable and DLL modules, rebasing and binding, 551 executable code, storing in heap, 524 executable file. See .exe file executable image file, 541 executable module building, 541, 547-550 loading in user-mode partition, 373 mapped into new process' address space, 542 running, 550-552 with several DLL modules, 539 executable source code files, 548 EXECUTE attribute, 468 execution content, containing user-defined value, 364 context, of fiber, 362 picking up with first instruction following except block. 684 resuming after failed CPU instruction, 683 times of threads, 179-183 exit code, setting, 154 exit function, 72 exit time, returned by GetThreadTimes, 179 ExitProcess function, 104, 105-106, 155 causing process or thread to die, 105 explicit calls to as common problem, 106 getting address of real, 638 hooking, 634 to terminate process, 564 trapping all calls to, 638

ExitThread function, 153, 154, 166, 364, 558 avoiding, 166 causing process or thread to die, 105 explicit calls to as common problem, 106 to kill thread. 567 kills thread, 154 ExitWindowsEx function, 133 ExpandEnvironmentStrings function, 82 explicit data types, 26 explicit DLL module loading and symbol linking, 553-561 explicitly loaded DLLs, 564 Explorer.exe's address space, injecting DLL into, 610 export section, in DLL file, 545-546 exported symbol, explicitly linking, 561 EXPORTS section, 547 -exports switch (DumpBin), 545 expression parameter, 20 extended basic limit restriction, 129 extended (Ex) version functions, 23 Extended Latin Unicode character set, 13 extended limits, on job, 132 EXTENDED_STARTUPINFO_PRESENT flag, 95 extended versions of kernel object creation functions, 49 extern "C" modifier in C++ code, 544 using to mix C and C++ programming, 546 extern keyword, importing symbol, 548 external manifest, 114

F

/F option, 451 facility codes, 698 Facility field, in error code, 7 failure, indicating for function, 7 Fast User Switching, 52 faulting process, all threads suspended, 715 fdwAllocationType parameter, 420 fdwCreate parameter of CreateProcess, 93-95, 191 specifying priority class, 95 fdwFlags parameter of HeapAlloc, 525 of HeapFree, 528 of HeapReAlloc, 527 of HeapSize, 527 fdwOptions parameter, 523 fdwPriority parameter, 191 fdwProtect parameter of CreateFileMapping, 479 of VirtualAlloc, 420 fdwReason parameter, 562 fiber(s), 361-367 FIBER_FLAG_FLOAT_SWITCH, 362 Fiber Local Storage. See FLS functions FiberFunc function, 366 fields in error code, 7

FIFO algorithm, 247 file(s), 290 adding to problem reports, 738, 745 contents broken into sections, 466 copying other media in background, 148 direct access, 331 extremely large, 296 getting size of, 299 locating in Windows Vista, 146 mapping into two address spaces, 499 memory-mapping, 487 opening, 291 pointer, setting beyond end of file's current size, 302 reversing contents of, 488 setting end of, 302 types added to problem reports, 745 unmapping data, 485–486 unregistering, 739 FILE_ATTRIBUTE_ARCHIVE flag, 297 FILE_ATTRIBUTE_ENCRYPTED flag, 297 FILE_ATTRIBUTE_HIDDEN flag, 297 FILE_ATTRIBUTE_NORMAL flag, 297 FILE_ATTRIBUTE_NOT_CONTENT_INDEXED flag, 297 FILE_ATTRIBUTE_OFFLINE flag, 297 FILE_ATTRIBUTE_READONLY flag, 297 FILE_ATTRIBUTE_SYSTEM flag, 297 FILE_ATTRIBUTE_TEMPORARY, 296, 297 file attribute flags, 297-298 FILE_BEGIN value, 301 file caching, 477 FILE_CURRENT value, 301 file data, mapping, 482-485 file desired access rights, 478 file devices, 299-302 FILE END value, 301 FILE_FLAG_BACKUP_SEMANTICS flag (CreateFile), 291, 296, 298 FILE_FLAG_DELETE_ON_CLOSE flag, 296 FILE_FLAG_NO_BUFFERING flag, 295 accessing extremely large files, 296 FileCopy opening using, 331 FILE_FLAG_OPEN_NO_RECALL flag, 296 FILE_FLAG_OPEN_REPARSE_POINT flag, 296 FILE_FLAG_OVERLAPPED flag (CreateFile), 297, 305 not specifying, 303 opening device for asynchronous I/O, 311 source file opened with, 331 FILE_FLAG_POSIX_SEMANTICS flag, 296 FILE_FLAG_RANDOM_ACCESS flag, 295 FILE_FLAG_SEQUENTIAL_SCAN flag, 295 FILE_FLAG_WRITE_THROUGH flag, 296, 486 file handle, checking for invalid, 298 file image accessing pages in, 467 locating, 555 used as physical storage, 379 file kernel object, creating or opening, 477, 478-479

FILE_MAP_ALL_ACCESS access right, 483 FILE_MAP_COPY flag, 483, 486 FILE MAP EXECUTE access right, 483 FILE_MAP_READ access right, 483 FILE_MAP_READ parameter, 36 FILE_MAP_WRITE access right, 483 file objects, closing, 486-487 file pointers, 300-302, 489 file resources, required for processes, 146 File Reverse application (17-FileRev.exe), 487-490 File Save As dialog box, in Notepad, 31 FILE_SHARE_DELETE value, 293 FILE_SHARE_READ | FILE_SHARE_WRITE value, 293 FILE_SHARE_READ value, 293, 478 FILE_SHARE_WRITE value, 293, 478 FILE_SKIP_COMPLETION_PORT_ON_SUCCESS flag, 327 FILE_SKIP_SET_EVENT_ON_HANDLE flag, 313 FILE_TYPE_CHAR value, 292 FILE_TYPE_DISK value, 292 FILE_TYPE_PIPE value, 292 FILE_TYPE_UNKNOWN value, 292 FileCopy function, 331, 332 FileCopy sample application (10-FileCopy.exe), 331 file-mapping kernel object, 477, 479-482 file-mapping objects backed by a single data file, 495 creating, 488 multiple processes sharing, 499 naming, 482 sharing blocks of data between processes, 43 FileRev, cleaning up, 489 FileReverse function, 488 FILETIME structure, 258 filler value (0xfd), 21 filtered process, debugging, 116 filtered token, 110 FilterFunc function, 458 fImpLoad parameter (DllMain), 562 finally block(s) allowing to always execute, 748 ensuring execution, 664 executing code in, 689 forcing to be executed, 671 localizing cleanup code in, 669 not guaranteed to execute for any exception, 664 premature exit in try block, 662 putting return statement inside, 690 situations precluding execution of, 665 __finally keyword, 660 FindWindow, 607 first in, first out algorithm, 247 first-chance notification, 729, 731 fixed module, 120 /FIXED switch of linker, 464 passing to linker, 589

FLAG_DISABLE_THREAD_SUSPENSION, 736 flags for SetErrorMode, 83 stored in process' handle table, 44 used by CompareString, 25 Flags parameter (Sleep functions), 227 flat address space, 465 flNewProtect, 435 floating point-related exceptions, 696 floating-point operations, performed by fiber, 362 flow of control, from try block into finally block, 663 FLS functions, 364 FlsAlloc function, 364 FlsFree function, 364 FlsGetValue function, 364 FlsSetValue function. 364 FlushFileBuffers function, 303 FlushViewOfFile function, 486 ForceClose method, of g_mmf object, 508 ForceQueue value, 712 foreground process, 195 foreign node, using RAM from, 405 FORMAT_MESSAGE_ALLOCATE_BUFFER flag, 9 FORMAT_MESSAGE_FROM_SYSTEM flag, 9 FORMAT_MESSAGE_IGNORE_INSERTS flag, 9 FormatMessage function, 6, 9 FORWARD_WM_* macro, 775 forwarded functions, 583 frame-based mechanism, SEH as, 726 free address space, 375 FREE flag, locating, 599 Free memory region type, 386 free regions, no storage committed within, 387 FreeEnvironmentStrings function, 79, 95 freeing, memory blocks, 527 FreeLibrary function, 558 called by thread, 564 called by _FUnloadDelayLoadedDLL2, 575 decrementing library's per-process usage count, 559 to free DLL containing static TLS variables, 603 from inside DllMain, 563 steps performed when called by thread, 565, 566 thread pool calling, 355 FreeLibraryAndExitThread function, 558, 559, 564 FreeLibrary-WhenCallbackReturns function, 355 FreeUserPhysicalPages function, 442 fuErrorMode parameter, 83 fuExitCode parameter, 105, 106 full pathname, passed to LoadLibraryEx, 557 full thread stack region, 453 FuncaDoodleDoo example code, 665 Funcarama1 example code, 667 Funcarama2 example code, 668 Funcarama3 example code, 668 Funcarama4 example code, 669 Funcenstein1 example code, 660 Funcenstein2 example code, 661-662 Funcenstein3 example code, 663

Funcenstein4 example code, 666 Funcfurter1 example code, 663 Funcfurter2 example code, 672 Funcmeister1 coding example, 680 Funcmeister2 coding example, 681 function forwarders, 583, 633 function names, exporting without mangling, 547 functions allowing one process to manipulate another, 621 for backward compatibility with 16-bit Windows, 17 for C run time to call for invalid parameter, 20 calling asynchronously, 340-346 calling at timed intervals, 346-351 calling when single kernel object becomes signaled, 351-353 creating kernel objects, 37, 38 easily calling nondeprecated, 26 hooking, 640 manipulating kernel objects, 34 for opening devices, 291-292 placing thread in alertable state, 317 with problems in multithreaded environments, 160 protecting code against buffer overruns, 11 from psapi.h, 144 reading environment variables, 85 requiring default heap, 520 returning INVALID_HANDLE_VALUE, 39 succeeding for several reasons, 5 used full stack space for, 459 validating parameters, 3 _FUnloadDelayLoadedDLL2 function, 575, 577

G

g_fResourceInUse spinlock variable, 217 g_fShutdown, 234 g mmf object, 508 g_x variable, 587 garbage collection function, 427, 428 garbage data, 24 GarbageCollect function, 428 GD132.dll. 537 general protection (GP) fault errors, 83 GenerateWerReport function, 741, 751 generic data types, 26 GENERIC_READ access right, 478 GENERIC_READ flag, 303 GENERIC_READ value, 293 GENERIC_WRITE access right, 293, 478 GENERIC_WRITE flag, 303 GENERIC_WRITE value, 293 GET_MODULE_HANDLE_EX_FLAG_FROM_ ADDRESS parameter, 74 Get*CycleTime functions, 182 GetCommandLine function, 73, 76 GetCompressedFileSize function, 300 GetCPUFrequencyInMHz function, 182 GetCPUUsage function, 330 GetCurrentDirectory function, 84 GetCurrentFiber function, 364, 365

GetCurrentProcess function, 63, 169, 744 GetCurrentProcessId function, 104, 170, 625 GetCurrentThread function, 169 GetCurrentThreadId function, 104, 170 GetDiskFreeSpace function, 295 GetDlgItem function, 776 GetEnvironmentStrings function, 95 retrieving complete environment block, 77 using instead of _environ, 73 GetEnvironmentVariable function calling by child process, 46 determining existence and value of, 81 using instead of _wenviron, 73 GetExceptionCode intrinsic function, 695-697 GetExceptionInformation intrinsic function, 699-702 GetExitCodeProcess function, 108, 109 GetExitCodeThread function, 156, 458 GetFiberData function, 364 GetFileSize function, 488 GetFileSizeEx function, 299 GetFileType function, 292 GetFreeSlot private function, 230 GetFullPathName function, 85 GetHandleInformation function, 47 GetLargePageMinimum function, 423 GetLastError function, 4, 88 for additional information, 5 calling immediately after call to Create* function, 50 calling right away, 5 returning 2 (ERROR_FILE_NOT_FOUND), 50 returning 6 (ERROR_INVALID_HANDLE), 50 returning ERROR_ACCESS_DENIED, 54 returning ERROR ALREADY EXISTS, 51 returning ERROR_INVALID_HANDLE, 39, 40 GetLogicalProcessorInformation function, 214, 397 GetMappedFileName function, 387 GetMessage, 613 GetMessage loop, 149 GetModuleFileName function, 73, 560 GetModuleFileNameEx. 144 GetModuleHandle function, 74, 75, 559 GetModuleHandleEx function, 74 GetModulePreferredBaseAddr function, 120 GetMsgProc function, 610 GetMsgProc parameter (SetWindowsHookEx), 609 GetNativeSystemInfo function, 398 GetNewElement functions, 232 GetNextSlot helper function, 232 GetNextSlot private helper function, 231 GetNumaAvailableMemoryNode function, 405 GetNumaHighestNodeNumber function, 406 GetNumaNodeProcessorMask function, 406 GetNumaProcessorNode function, 405 GetOverlappedResult function, 314 GetPriorityClass function, 192 GetProcAddress function, 561 exact memory location of LoadLibraryW, 623 hooking, 639 GetProcessAffinityMask function, 204

794 GetProcessElevation helper function

GetProcessElevation helper function, 117 GetProcessHeaps function, 531 GetProcessId function, 104 GetProcessIdOfThread function, 104 GetProcessImageFileName function, 144 GetProcessIntegrityLevel function, 123 GetProcessIoCounters function, 138 GetProcessMemorvInfo function, 407 GetProcessPriorityBoost function, 195 GetProcessTimes function, 138, 169, 180 _getptd_noexit function, 166 GetQueuedCompletionStatus function, 141, 324 GetQueuedCompletionStatusEx function, 317, 326 GetQueueStatus, 367 GetStartupInfo function, 102 GetStatus method, of CIOCP, 333 GetStdHandle, opening console, 291 GetSystemDirectory function, 550 GetSystemInfo function, 203, 395-396, 466 GetThreadContext function, 185, 187 GetThreadId function, 104 GetThreadLocale function, 24 GetThreadPriority function, 193 GetThreadPriorityBoost function, 195 GetThreadTimes function, 169, 179 GetThreadWaitChain function, 285 GetTickCount64 function, 179, 181 Getting System Version page, 86 GetTokenInformation function, 118, 123 GetVersion function, 85 GetVersionEx function, 73, 86 /GF compiler switch, 90 /Gf compiler switch, 90 Global, as reserved keyword, 53 global atom table, giving job, 133 global data variable, 688 global namespace, 51–52 Global\ prefix, 52 global replaces, performing, 26 global task count, 345, 346 global unwinds, 671, 687-690 global variables in 08-Queue.exe application, 232 altering, 466 available to programs, 73 avoiding, 597 as better choice, 599 in DLL, 538 reflecting number of instances running, 467 globally set filter function, notifying, 708 GlobalMemoryStatus function, 404 to get total amount of RAM, 437 listing results of call, 406 GlobalMemoryStatusEx function, 404, 441 GP fault errors, handling, 83 grandchild process, spawning, 45 granularity. See also allocation granularity; page granularity of reserved region, 396

graphical applications, allowing fine-tuning, 81 Graphical Device Interface (GDI) objects, 37 growable heap, 525 /GS compiler switch in Microsoft Visual C++, 457 security features provided by, 570 taking advantage of, 27 gt_ prefix, for global TLS variables, 602 guard page, 451, 454 GUI-based applications, 68, 69, 608 GUID, 51

н

handle(s) closing to child process and thread, 103 counting in Windows Task Manager, 41 getting to device, 291 identifying kernel object, 34 measuring as undocumented, 39 obtaining default heap, 520 setting for existing heaps, 531 HANDLE_FLAG_INHERIT flag, 46 HANDLE_FLAG_PROTECT_FROM_CLOSE flag, 47 HANDLE_MSG macro defined in WindowsX.h, 774 not using, 766 HANDLE return type, 3 handle table flags stored in, 44 for kernel objects, 37 undocumented, 37 handle values passing as command-line argument, 46 as process-relative, 35 HANDLE_WM_* macros, 775 HANDLE_WM_COMMAND macro, 775 Handles column, in Select Process Page Columns dialog box, 41 Handshake (09-Handshake.exe) application, 252 hard affinities, 203, 205 hardware exceptions, 679 HasOverlappedIoCompleted macro, 307 hAutoResetEvent1 object, 246 hAutoResetEvent2 object, 246 hCompletionPort parameter of GetQueuedCompletionStatus, 325 of GetQueuedCompletionStatusEx function, 326 of PostQueuedCompletionStatus, 330 header annotation, 14 "Header Annotations" documentation on MSDN, 14 header file coding, 542 creating, 541 establishing, 542 including DLL's, 548 including in each DLL's source code files, 543 /headers switch (DumpBin), 468, 586 heap(s), 419, 519 advantage of, 519

allocating blocks of memory, 525-526 coalescing free blocks within, 532 creating additional, 523-531 destroying, 528-531 disadvantage of, 519 inheritance of, 531 reasons to create additional, 520-523 related functions, 531-533 rules for committing and decommitting storage, 519 serializing access to itself, 523 using multiple, 531 using with C++, 528-531 validating integrity of, 532 HEAP_CREATE_ENABLE_EXECUTE flag, 523, 524 heap dump, custom, 745 heap functions, 703 HEAP_GENERATE_EXCEPTIONS flag, 523, 524, 525, 527 passing to heap functions, 703 specifying, 525 /HEAP linker switch, 519 heap manager, raising exeption, 524 HEAP_NO_SERIALIZE flag, 523, 525, 526, 527, 528 absence of, 524 avoiding, 523 safe use of, 524 HEAP_REALLOC_IN_PLACE_ONLY flag, 527 HEAP_ZERO_MEMORY flag, 525, 527 HeapAlloc function actions of, 523 causing to raise software exception, 525 exceptions raised by, 525 HeapCompact function, 532 HeapCreate function, 523 HeapDestroy function, 528 HeapEnableTerminationOnCorruption parameter, 524 HeapFree function, 527 called by delete operator function, 530 calling to free memory, 77 HeapLock function, 532 HeapReAlloc function, 526 HeapSetInformation function, 524 HeapSize function, 527 HeapUnlock function, 532 HeapValidate function, 532 HeapWalk function, 532 hEvent member, of OVERLAPPED structure, 307 hEventMoreWorkToBeDone event, 280 hFile parameter, 301 of CancelIoEx, 310 of CreateFileMapping, 479 of GetFileSizeEx, 299 of LoadLibraryEx, 555 of ReadFile and WriteFile, 303 SetFileCompletionNotificationModes function, 313 hFileMappingObject parameter, 482 hFileTemplate parameter, 298, 332 hHeap parameter of HeapAlloc, 525

of HeapReAlloc, 527 of HeapSize, 527 hidden file, indicating, 297 high 32 bits, 481 High level of trust, 122 HIGH_PRIORITY_CLASS, 95, 191 high priority class, 189 high priority level, 191 higher-priority threads, 187 highest memory address, 396 highest thread priority, 190, 193 highestAvailable value, 114 high-resolution performance functions, 181 high-water marker, maintained by NTFS, 332 HINSTANCE type, 74 hInstanceExe parameter, 73 hInstDll parameter containing instance handle of DLL, 562 of GetProcAddress, 561 of SetWindowsHookEx, 609 hInstModule parameter, 560 hint value, of symbol, 550 hJob parameter of AssignProcessToJobObject function, 136 of UserHandleGrantAccess function, 134 HKEY_PERFORMANCE_DATA root key, 118 hmodCur member, of DelayLoadInfo, 574 HMODULE type, parameter of, 74 HMODULE value, 555, 558 hObject parameter of SetHandleInformation, 46 of SetThreadpoolWait function, 352 of WaitForSingleObject function, 109, 243 hObjectToSignal parameter, 279 hObjectToWaitOn parameter, 279 hook functions, 576, 635 Hook_MessageBoxA function, 640 hooks injecting DLL using, 609-616 owned by thread, 155 host machine, 395 host system, version numbers, 87 hPrevInstance parameter, 75 hProcess parameter of AllocateUserPhysicalPages, 441 of AssignProcessToJobObject function, 136 of CreateRemoteThread, 621 of GetExitCodeProcess, 108 of GetProcessMemoryInfo, 407 of ReadProcessMemory, 624 in SetPriority, 191 of SetProcessAffinityMask function, 203 of TerminateProcess, 106 of VirtualAllocExNuma, 421 of WaitForInputIdle, 278 of WerGetFlags, 737 of WerReportAddDump, 744 WriteProcessMemory, 624

hReport parameter of WerReportAddDump, 744 of WerReportAddFile, 745 of WerReportSetParameter, 743 of WerReportSetUIOption, 746 of WerReportSubmit, 746 HRESULT, values for safe string functions, 22 hSourceHandle parameter, 61 hSourceProcessHandle parameter, 61 hStdError value, 98 hStdInput value, 98 hStdOutput value, 98 hTargetProcessHandle parameter, 61 hThread parameter, 204 of CancelSynchronousIo, 304 of GetExitCodeThread. 156 of QueueUserAPC function, 319 of SetThreadIdealProcessor function, 205 of SetThreadPriority function, 193 of TerminateThread, 155 of WerReportAddDump, 744 hTimer parameter, 257 hUserObj parameter, 134 hWctSession parameter, 285 hyper-threading, 178

I

IA-64, 8-KB page size, 376 IA-64 CPU, 391 IAT (Import Address Table), updating, 638 icacls.exe Vista console mode tool, 123 icons manually repositioning on desktop, 611 setting for dialog boxes, 766 ID(s) of current process, 104 getting set of those currently in job, 139 reused immediately by system, 103 of running thread, 104 used mostly by utility applications, 103 idata section, 470. ideal CPU, setting for thread, 205 IDLE_PRIORITY_CLASS, 95, 191 idle priority class, 188, 189 idle thread priority, 190, 193 #ifdefs, putting into code, 702 IMAGE_FILE_RELOCS_STRIPPED flag, 589 image files, executing from floppies, 380 Image memory region type, 386 image of .exe file, 379 image regions, displaying pathnames of files, 387 IMAGE_THUNK_DATA structures, 638 Image Walk DLL, 631-633 __ImageBase pseudo-variable, 74 ImageDirectoryEntryToData, 637 implicit linking, 540 Import Address Table (IAT), updating, 638

import section of each DLL checked during loading, 551 embedded in executable module, 548-550 in executable module, 541 manipulating by API hooking, 636-639 parsed by executable module, 542 virtual addresses of all imported symbols, 593 -imports switch (DumpBin), 549 Increase Scheduling Priority privilege, 189 IncreaseUserVA parameter, 374 Indexed Locations, searching, 147 indexing, by Windows Indexing Services, 146 INFINITE passed to WaitForSingleObject, 244 passing for dwMilliseconds, 177 infinite loop, generating, 692 infinite wait, of mutexes vs. critical sections, 268 information, storing on per-fiber basis, 364 inheritable handle, 44, 92 inheritance, 81. See also child process(es); kernel object handles; object handle inheritance heaps, 531 kernel object handle, 91 inheritance flag, 46 Inherit.cpp program, 91, 92 initial state, of mutex, 266 initialization of mutexes vs. critical sections, 268 value, passing to thread function, 152 InitializeCriticalSection function, 220, 223 InitializeCriticalSection parameter, 222 InitializeCriticalSectionAndSpinCount function, 222, 223 InitializeProcThreadAttributeList function, 101 InitializeSListHead function, 213 InitializeSRWLock function, 224 InitializeThreadpoolEnvironment function, 358 Inject Library sample application, 625-626 injecting DLL into process' address space, 607-634 InjectLib, 626 INNER_ELEMENT structure, 230 instance handle, 75 instruction pointer register, 157 integer-related exceptions, 696 integrity level assigning to securable resources, 122 setting for file system resource, 123 intercepted Windows messages, 124 interlocked family of functions, 209 interlocked functions, executing extremely quickly, 210 interlocked helper functions, 213 Interlocked Singly Linked List, 213 interlocked (user-mode) functions, 277 InterlockedCompareExchange function, 212 InterlockedCompareExchange64 function, 213 InterlockedCompareExchangePointer function, 212 InterlockedDecrement function, 213, 346

InterlockedExchange function, 211 InterlockedExchange64 function, 211 InterlockedExchangeAdd function, 209, 210, 213 InterlockedExchangeAdd64 function, 209 InterlockedExchangePointer function, 211 InterlockedFlushSList function, 213 InterlockedIncrement function, 213, 345 atomically incrementing value by, 210 incrementing volatile long value, 226 InterlockedPopEntrySList function, 213 InterlockedPushEntrySList function, 213 Internal member origin of, 307 of OVERLAPPED structure, 307, 312, 326 InternalHigh member origin of, 307 of OVERLAPPED structure, 307, 312 international markets, targeting, 11 Internet Explorer, user canceling Web request, 304 interprocess communication, with DuplicateHandle, 62 interthread communication techniques, 330 intrinsic function, 671 INUSE flag, 600 invalid file handle, checking for, 298 invalid handle, 39, 40 INVALID_HANDLE_VALUE CreateFile returning, 482 functions returning, 3 returned by CreateFile, 479 returned if CreateFile fails, 298 returning for, 479 invalid memory attempting to access, 674 occurring inside finally block, 704 invalid parameter, C run time detecting, 20 I/O accounting information, querving, 138 I/O completion notifications, receiving, 310 I/O completion port(s), 289, 320-333 architecting around, 324-327 associating device with, 321, 322-324 creating, 143, 321-322 designed to work with pool of threads, 321 internal workings of, 323 as interthread communication mechanism, 289 keeping CPUs saturated with work, 328 kernel object, creating, 141 managing thread pool, 327-328 mechanism used by most applications, 262 threads assigned to specified, 328 threads monitoring, 141 for use within single process only, 322 using, 310 waking threads in waiting thread queue, 325 waking up waiting threads, 327 I/O completion queue, 324 IO_COUNTERS structure, 138 I/O notification, 330

I/O request(s) cancel pending, 310 issuing to device driver, 315 issuing without completion entry, 324 outstanding at any one time, 332 passing to device driver, 305 priorities, 196 queuing for device driver, 305 receiving completed notifications, 310-333 simulating completed, 330 I/O-bound task, using separate thread for, 148 IOCP.h file, CIOCP class found in, 332 IoInfo member, 132 IsDebuggerPresent function, 708 IsModuleLoaded function, 577 isolated applications, taking advantage of, 586 IsOS function, 398 IsProcessInJob function, 127, 136 IsProcessorFeaturePresent function, 396 IsTextUnicode function, 31, 488 IsThreadAFiber function, 364, 367 IsThreadpoolTimerSet function, 347 IsUserAnAdmin function, 118 IsWow64Process function, 94, 398

J

JAC-compliant applications, 115 Japanese kanji, 12 JIT debugging. See just-in-time debugging job(s) notifications, 140-142 placing processes in, 136 placing restrictions on, 129-135 terminating all processes in, 136 job event notifications, 142 job kernel object, 125 creating, 128 fitting into I/O completion port model, 289 Job Lab application (05-JobLab.exe), 143-144 correct use of, 153 demonstrating I/O completion ports and job objects working together, 289 JOB_OBJECT_LIMIT_ACTIVE_PROCESS flag, 131 JOB_OBJECT_LIMIT_AFFINITY flag in LimitFlags member, 131 setting, 130 JOB_OBJECT_LIMIT_BREAKAWAY_OK flag, 136 JOB OBJECT LIMIT BREAKAWAY OK limit flag, 136 JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_ EXCEPTION limit flag, 132 JOB_OBJECT_LIMIT_JOB_MEMORY flag, 132 JOB_OBJECT_LIMIT_JOB_TIME flag, 130, 131 JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME flag, 130, 137 JOB_OBJECT_LIMIT_PRIORITY_CLASS flag, 130, 131 JOB_OBJECT_LIMIT_PROCESS_MEMORY flag, 132

JOB_OBJECT_LIMIT_SCHEDULING_CLASS flag, 131

JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK flag, 136 JOB_OBJECT_LIMIT_WORKINGSET flag, 131 JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS notification, 142 JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT notification, 142 JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO notification, 142 JOB_OBJECT_MSG_END_OF_JOB_TIME notification, 142 JOB_OBJECT_MSG_END_OF_PROCESS_TIME notification, 142 JOB_OBJECT_MSG_EXIT_PROCESS notification, 142 JOB_OBJECT_MSG_JOB_MEMORY_LIMIT notification, 142 JOB_OBJECT_MSG_NEW_PROCESS notification, 142 JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT notification, 142 JOB_OBJECT_TERMINATE_AT_END_OF_JOB notification, 142 JOB_OBJECT_UILIMIT_DESKTOP flag, 133 JOB_OBJECT_UILIMIT_DISPLAYSETTINGS flag, 133 JOB_OBJECT_UILIMIT_EXITWINDOWS flag, 133 JOB_OBJECT_UILIMIT_GLOBALATOMS flag, 133 JOB_OBJECT_UILIMIT_HANDLES flag, 133, 134 JOB_OBJECT_UILIMIT_READCLIPBOARD flag, 133 JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS flag, 133 JOB_OBJECT_UILIMIT_WRITECLIPBOARD flag, 133 job object, creating, 143 job statistics, querying, 137-144 Job tab, of Process Explorer box, 139 Job.h file, 144 jobless process, 143 JobMemoryLimit member, 132 JOBOBJECT_BASIC_ACCOUNTING_INFORMATION structure, 137 JOBOBJECT_BASIC_AND_IO_ACCOUNTING_ INFORMATION structure, 138 JOBOBJECT_BASIC_LIMIT_INFORMATION structure, 129, 132 allocating, 129 members of, 131 SchedulingClass member, 130 JOBOBJECT_BASIC_PROCESS_ID_LIST structure, 139 JOBOBJECT_BASIC_UI_RESTRICTIONS structure, 129, 132 JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure, 129, 132 JOBOBJECT_SECURITY_LIMIT_INFORMATION structure, 129, 135 JobObjectBasicAccountingInformation parameter, 137 JobObjectBasicAndIoAccountingInformation parameter, 138 JobToken member, 135 JUMP instructions, as CPU-dependent, 635 just-in-time debugging, 713-715, 750

K

kanji, 12 kernel address space, 374 kernel interprocess communication (IPC) mechanism, 281 kernel mode, 185 kernel object(s) applications leaking, 40 binding to thread pool wait object, 352 closing, 39-43, 107, 486 containing flag (wave-in-the-air), 242 creating, 38-39 creation functions, 49 gaining access to existing, 36 getting up-to-date list of, 42 handle table for, 37 identifying, 279 kernel address space, content stored in, 45 keyed event type of, 224 managing thread, 145 manipulating, 34 monitoring number used by any application, 41 multiple namespaces for in Terminal Services, 51 naming, 48-60 outliving process, 35 owned by kernel, 35 performance of, 241 in process, 67 process boundaries, sharing across, 43-64 security descriptor, protecting, 35 sharing by name, 50 sharing names, 50 in signaled or nonsignaled state, 242 signaling and waiting for single atomic operation, 279 thread synchronization, behaving with respect to, 276-277 threads, using to synchronize, 241 types of, 33 usage count, incrementing, 45 kernel object data structures, 34 kernel object handle inheritance, 91 kernel object handles controlling which child processes to inherit, 46 inherited by child process, 100 as process-relative, 43 kernel time, 179 Kernel32 methods, 27 Kernel32.dll, 537, 571 kernel-mode CPU time, 137 kernel-mode partition, 373, 375 KEY_ALL_ACCESS, 37 KEY_QUERY_VALUE, 36 keyboard buffer, identifying, 98 keyed event type, of kernel objects, 224 keystrokes, forcing into application, 278 key/value pair, adding to attribute list, 101 known DLLs, 584-585

L

L, before literal string, 14 language identifier, in error code, 9 large blocks, allocating, 526 LARGE_INTEGER structure, 258 LARGE_INTEGER union, 299 large pages of RAM, 480 large user-mode address space, 374 /LARGEADDRESSAWARE linker switch, 374, 375 large-page support, in Windows, 422 Last MessageBox Info sample application, 639-655 last-chance notification, 729, 730 last-in first-out (LIFO), threads awakened, 325 Latin1 characters Unicode character set, 13 LB_GETCOUNT message, 776 LB_GETTEXT message, 612 LCID (locale ID), 24 lComparand parameter, 212 __leave keyword, 662 added to Microsoft's C/C++ compiler, 669 using in try block, 670 LeaveCriticalSection function, 218, 222 forgetting calls to, 219 thread pool calling, 355 LeaveCriticalSection-WhenCallbackReturns function, 355 legacy replacement functions, 20 level attribute, in <trustInfo>, 114 levels of trust, 122 .lib file passing to linker, 548 producing, 541, 545 /Lib switch, 573 LibCMtD.lib, 159 LibCMt.lib, 159 liDistanceToMove parameter, 301 LIFO order, threads waking up in, 331 LimitFlags member, 130, 131 LineSize field, 214 LINGUISTIC_IGNORECASE flag, 25 LINGUISTIC_IGNOREDIACRITIC flag, 25 lInitialCount parameter, 263 linked lists implementing spreadsheets, 424 memory-mapped files and, 496 linker creating executable module, 548 forcing to look for entry point function, 766 linker defined pseudo-variable, 72 linker switches embedding inside your source code, 472 set up by Visual Studio, 69 list boxes, appending strings into, 238 ListBox_AddString, 237 ListBox_SetCurSel, 237 ListView control, 612 lMaximumCount parameter, 263 LOAD_IGNORE_AUTHZ_LEVEL flag, 558

LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE flag, 556 LOAD_LIBRARY_AS_DATAFILE flag, 556 LOAD_LIBRARY_AS_IMAGE_RESOURCE flag, 556 LOAD_WITH_ALTERED_SEARCH_PATH flag, 556-558 loader creating virtual address space, 542, 550 fixing up all references to imported symbols, 551 mapping executable module, 550 search order, 550 searching disk drives for DLL, 550 LoadIcon function, 73 LoadLibrary function, 555 any program can call, 472 calling for DLLs, 464 calling to load desired DLL, 621 incrementing per-process usage count, 559 linking to DLL containing static TLS variables, 603 mixing with LoadLibraryEx, 560 steps performed when called by thread, 565 thread calling, 622 LoadLibrary macro, 622 LoadLibraryA function, 622 LoadLibraryEx function, 555, 564 avoiding calls to from inside DllMain, 563 incrementing per-process usage count, 559 mixing with LoadLibrary, 560 LoadLibraryW function, 622 LoadStringA function, 17 local access, creating heaps for, 522 Local, as reserved keyword, 53 .local file, 585 Local\ prefix, 52 Local Procedure Call, 281 local unwind, 662, 671 locale ID. 24 LocalFileTimeToFileTime function, 258 localization DLLs facilitating, 537 made easier by Unicode, 26 localizing applications, 11 lock(s) creating, 570 not holding for long time, 239 tips and techniques, 238-240 Lock Pages in Memory user right, 423 LockCop application, 09-LockCop.exe, 281-287 LockCop tool, 287 LockObject view, 287 logical disk drive opening, 291 use of, 290 logical resources accessing multiple simultaneously, 238 each with lock, 238 logical size, returning file's, 300 LONG return type, 3

long value, constructing, 773 LONGLONG variable, 405 Lovell, Martyn, 19 low 32 bits, 481 Low integrity level processes, 124 Low level of trust, 122 low priority level, thread with, 191 lower-priority thread, 427 lowest thread priority, 190, 193 low-fragmentation heap algorithm, 526 low-priority services, executing long-running, 196 low-priority thread, 196 lParam parameter, 774 lpAttributeList field, 99 lpDesktop value, 97 lPeriod parameter, 257, 258 lpFile field, 115 lpMaximumApplicationAddress member, 396 lpMinimumApplicationAddress member, 396 lpParameters field, 116 lpReserved value, 97 lpReserved2 value, 98 lPreviousCount variable, 270 lpSecurityDescriptor, 36 lpTitle value, 97 lpVerb field, 115 lReleaseCount parameter, 264 lstrcat, not using, 27 lstrcpy, not using, 27

Μ

macros, in WindowsX.h, 773 mailslot, 43, 290 mailslot client, opening, 291 mailslot server, opening, 291 main function, changing to WinMain, 70 mainCRTStartup function, 69, 70 mainfestdependency switch, 766 maintability, in code, 15 MAKELONG macro, from WinDef.h, 773 MAKESOFTWAREEXCEPTION macro, 765 malloc, 26, 167 malware creating same boundary descriptor, 54 exploiting unsafe string manipulation, 18 inheriting high privileges of Administrator, 110 isolating core components from, 52 writing code into areas of memory, 381 Mandatory Integrity Control, 122 .manifest suffix, 114 manual-reset events calling PulseEvent, 251 creating auto-reset event, 248 no successful wait side effect for, 249 signaling, 247 manual-reset timer, signaling, 257 Mapped memory region type, 386 MapUserPhysicalPages function, 441

MapViewOfFile function, 482-484 requirements for file offset parameters, 495 requiring process to call, 497 reserving different address space regions, 498 MapViewOfFileEx function, 496, 497 MapViewOfFileExNuma function, 485 master list, of error codes, 6-7 MAX PATH characters long, 48, 294 MAX_PATH constant, defined in WinDef.h, 84 _MAX_PATH constant, in various source code, 294 MaxArchiveCount, 740 maximum number of threads, setting, 328 MAXIMUM_PROCESSORS value, 205 maximum resource count, in semaphore kernel object, 262 MAXIMUM_SUSPEND_COUNT times, 175 MAXIMUM_WAIT_OBJECTS value, 245 MaxQueueCount for WER store, 740 Medium level of trust, 122 meeting-planner type of application, 259 MEM_COMMIT flag, 422 MEM_COMMIT identifier, 421 MEM_DECOMMIT identifier, 426 MEM_LARGE_PAGE flag, 423 MEM_PHYSICAL flag, 440 MEM_RELEASE, 426 MEM_RESERVE flag, 440 MEM_RESERVE identifier, 420 MEM_RESET flag, 436, 437 MEM_TOP_DOWN flag, 420 memcpy intrinsic function, 671 memory allocating for fiber's execution context, 362 copying pages of to paging file, 378 corruption from string manipulation, 18 determining state of regional address space, 429 DLLs conserving, 537 localizing accessses to, 522 reserving and committing all at once, 423 memory address, choosing for reserved region, 420 memory alignment faults, 83 memory architecture, of Windows, 371-393 MEMORY_BASIC_INFORMATION structure, 409, 410 memory block(s) changing size of, 526 freeing, 527 in kernel object, 34 obtaining size of, 527 schedulable thread with exclusive access to, 250 memory linkage, avoiding, 166 Memory Load, in VMStat, 406 memory management copy-on-write. See copy-on-write mechanism more efficient with heaps, 521 on NUMA machines, 405-408 memory pages, protection attributes for, 381 memory regions, types of, 386 Memory.hdmp file, 735, 736

memory-mapped file desired access rights, 483 memory-mapped files, 379, 419, 476-477 backed by paging file, 499 cleaning up, 477 coherence and, 495 creating, 479 growable, 509 implementation details of, 497 processing big file using, 494-495 purposes of, 463 reversing contents of ANSI or Unicode text file, 487 sharing data among processes, 498 sharing data in, 109 sharing data with other processes, 496 sparsely committed, 504-515 specifying base address of, 496-497 using, 477-490 Memory-Mapped File Sharing application (17-MMFShare.exe), 500-501 memory-mapping, file, 487 memory-related exceptions, 695 MEMORYSTATUS structure, 404 MEMORYSTATUSEX structure, 405 MemReset.cpp listing, 437 message crackers, 773-775 cracking apart parameters, 774 using with dialog boxes, 766 in WindowsX.h file, 773 message ID, for error, 5 message queue, of a thread, 613 message text, describing error, 5 MessageBox function, hooking all calls to, 639 MessageBoxA, hooking of, 640 MessageBoxW, hooking of, 640 messages, sending to child controls, 776 MFC class library, 18 microdump, custom, 745 Microsoft C compiler. See C compiler Microsoft Spy++. See Spy++ Microsoft Visual C++ compiler. See also C++ compilers building DLL, 570 C++ exception handling, 728 syntax, 659 Microsoft Visual C++, supporting C++ exception handling, 660 Microsoft Visual Studio. See Visual Studio Microsoft Windows. See Windows operating system minidump file adding to problem report, 744 custom, 745 MiniDump.mdmp file, 736 minidumps, information about, 735 minimum memory address, 396 misaligned data accesses, 392 misaligned data, code accessing, 391 MMF Sparse application (17-MMFSparse.exe), 505-515 Modified letters Unicode character set, 13

modules binding, 592-595 determining preferred base addresses, 120 import section, strings written in ANSI, 638 Modules! menu item, in ProcessInfo, 120 Monitor class, 228 mouse cursor, control over, 98 MoveMemory function, 486 MSDN Web site, Getting System Version page, 86 MS-DOS application, forcing system to run, 94 MsgBoxTimeout function, 348 MsgWaitForMultipleObjects function, 262, 279 MsgWaitForMultipleObjectsEx function, 279, 317 MSIL code, 159 msPeriod, 347 MSVCMRt.lib, 159 MSVCR80.dll library, 159 MSVCRtD.lib, 159 MSVCRt.lib, 159 MSVCURt.lib, 159 msWindowLength parameter, 347 multibyte-character string, converting, 28 MultiByteToWideChar function, 27, 28 Multimedia Class Scheduler service, 174 multiple paging files, 380 multiple processes, sharing data with each other, 463 multiple threads accessing shared resource, 207 communicating with each other easily, 150 scheduling, 174 using judiciously, 149 multiprocessor environment, 214 multithreaded applications, debugging, 281 multithreaded environment tcstok s function in, 597 working asynchronously, 207 multithreading, allowing simplification, 148 multithreading operating system, benefits of, 148 mutex kernel objects, 265-271 checks and changes performed atomically, 266 creating, 265 rules for, 265 mutex object, creating, 44 mutexes abandoned, 267 compared to critical sections, 267 ensuring exclusive access, 265 as slower than critical sections, 265 special exception to normal kernel object rules, 266 thread ownership concept for, 267 threads in different processes synchronizing execution, 43 tracked by WCT, 281 violating normal rules, 265 as worst performing, 226 MWMO_ALERTABLE flag, 317

Ν

name uniqueness, ensuring, 51 named kernel objects, functions creating, 48 named objects forcing to go into global namespace, 52 preventing multiple instances of applications, 51 named pipe client, 291 named pipes, 43, 290 named pipe server, 291 namespaces closing with existing objects, 60 private, 53-60 nanoseconds, measuring, 258 nCmdShow, 99 nested exceptions, 700, 707 .NET framework easily integrating with, 26 encoding all characters and strings, 12 NetMsg.dll module, 9 network drives, 380 network share, passed to LoadLibraryEx, 557 new operator, calling in C++, 528 new operator function, 530 nMainRetVal, 72 nNumberOfArguments parameter, 704 nNumBytesToRead parameter, 303 nNumBytesToWrite parameter, 303 non-constant string, 89 none-gleton application, 53 nonfiltered security token, 111 nonfiltered token, grabbing, 118 non-real-time priority class, thread in, 190 nonreentrant function, 153 nonsignaled state, 241 changing event to, 249 object set to, 157 Non-Uniform Memory Access. See NUMA non-Visual C++ tools, creating DLLs for use with, 546 No-Read-Up, 124 NORM_IGNORECASE flag, 25 NORM_IGNOREKANATYPE flag, 25 NORM_IGNORENONSPACE flag, 25 NORM_IGNORESYMBOLS flag, 25 NORM_IGNOREWIDTH flag, 25 NORMAL_PRIORITY_CLASS, 95, 191 normal priority class, 189 normal thread priority, 190, 193 Notepad command line, 91 opening both Unicode and ANSI files, 31 property page for shortcut running, 99, 100 spawning instance of, 489 notifications about jobs, 140-142 more advanced, 141 nPriority parameter, 193 NTFS creating file on, 294

support for sparse files, 505 NtQueryInformationProcess, 122 NULL return value, 3 NULL-pointer assignment partition, 372 __nullterminated prefix, 14 NUMA machines applications running on, 485 forcing virtual memory to particular node, 421 memory management on, 405–408 NUMA (Non-Uniform Memory Access), 203 Number Of Processors check box, 206 number pool, for system IDs, 103 NumberParameters member, 701, 704

0

.obj module, producing, 541 object handle inheritance, 43-47 object handles, duplicating, 60-64 object inheritance. See object handle inheritance object names, creating unique, 51 objects creating separate heaps for, 521 managing linked list of, 218 process termination cleaning up, 40 ObjectStatus field, 287 ObjectType field, 286 offline storage, 296, 297 Offset member, 306 OffsetHigh member, 306 OLE applications, 188 one-file one-buffer method, 476 one-file two-buffer method, 476 one-file zero-buffer method, 477 _onexit function, 72 OPEN_ALWAYS value, 294 Open command, 278 OPEN_EXISTING value, 294 Open* functions, 50 OpenEvent function, 249 OpenFile function, 17 OpenFileMapping function, 36, 501 OpenJobObject function, 128 OpenMutex function, 266 OpenPrivateNamespace function, 60 OpenSemaphore function, 263 OpenThread function, 177 OpenThreadWaitChainSession function, 284 OpenWaitableTimer function, 256 operating system file, indicating, 297 freeing memory, 104, 154 locating free page of memory in RAM, 378 products, identifying, 87 scheduling threads, 67, 68 upgrades, effects on bound modules, 595 operations always performed synchronously, 308 performing at certain times, 256

order of execution, in SEH, 689 ordinal number, specifying, 561 ordinal values, 546 OS_WOW6432, passing as parameter, 398 _osver global variable, 73 OSVERSIONINFO structure, 86 OSVERSIONINFOEX structure, 86, 87 OVERLAPPED ENTRY, 326 OVERLAPPED structure, 306-307 allocating and initializing unique, 309 C++ class derived from, 307 members initialized, 311 members of, 306 not moving or destroying, 309 OverlappedCompletionRoutine callback function, 354 over-the-shoulder logon, 112 overwriting code, hooking by, 635 overwriting, protecting against accidental, 455

Ρ

page(s), 376 altering rights of, 435 committing physical storage in, 376 swapping in and out of memory, 378 writable, 382 page commit information, tracking, 425 PAGE_EXECUTE_* protections, 381 PAGE_EXECUTE_READ protection attribute, 381, 480 PAGE_EXECUTE_READWRITE protection attribute, 381, 382, 480 PAGE_EXECUTE_WRITECOPY protection attribute, 381, 382 PAGE_EXECUTE protection attribute, 381 page faults, 137, 378 page frame number, 441 page granularity, 425, 426 PAGE_GUARD protection attribute, 383, 390, 411 PAGE_NOACCESS protection attribute, 381, 435 PAGE_NOCACHE protection attribute flag, 382, 390 PAGE_NOCACHE section attribute, 479 PAGE_READONLY protection attribute, 381, 480 PAGE_READWRITE protection attribute, 381, 421, 422, 480 assigned to region and committed storage, 422 assigning, 382 page protection for stack's region, 451 passing to VirtualAlloc, 440 passing with CreateFileMapping, 482 page size address space reserved as multiple of, 376 showing CPU's, 396 PAGE_WRITECOMBINE protection attribute flag, 383, 390 PAGE_WRITECOPY protection attribute, 381, 480 changing to PAGE_READWRITE, 484 conserving RAM usage and paging file space, 382 page-locked storage requirement, 308 page-sized threshold, controlling, 457

paging file(s) backing memory-mapped files, 499 on disk, 377 maximum number of bytes, 407 no storage required from, 594 physical storage not maintained, 379-380 size of system's, 378 using multiple, 380 parallel port opening, 291 use of, 290 parameter not referenced warning, 76 parent process adding environment variable, 46 debugging child process, 93 determining, 104 forgetting to close handle to child process, 108 handle table, copying, 45 obtaining its current directories, 85 passing handle value as command-line argument, 46 preventing child process from inheriting error mode, 83 setting child process' current drive and directory, 95 spawning child process, 44, 278 waiting for child to complete initialization, 46 parent-child relationship, of processes, 43 parent's thread, 278 pArguments parameter, 704 ParseThread method executing for each thread of, 286 as heart of wait chain traversal, 284 ParseThreads function, 282, 284 partition tables, 291 partitions, 372-375 pathname, with CreateFile, 294 pAttributeList parameter, 101 PAUSE assembly language instruction, 178 paused thread list, moving thread's ID to, 327 pbIsCycle parameter, 285 pcchRemaining parameter, 23 pCompletionKey parameter, 141 pCompletionPortEntries array parameter, 326 pConditionVariable parameter, 227 pContext parameter, 285 pde parameter, 279 pDefaultChar parameter, 29 pDueTime parameter, 257, 258 pDumpCustomOptions parameter, 745 pdwExitCode parameter of GetExitCodeProcess, 108 of GetExitCodeThread, 156 pdwFlags parameter, 47 pdwNumBytes parameters, 303 pdwNumBytesRead parameter, 624 pdwNumBytesWritten parameter, 624 pdwProcessAffinityMask parameter, 204 pdwReturnSize parameter, 135 pdwSystemAffinityMask parameter, 204

pdwThreadID parameter, 153 PE file, mapping, 387 PE header, 451 PeakJobMemoryUsed member, 132 PeakProcessMemoryUsed member, 132 PeakWorkingSetSize field, 408 PEB (process environment block), 121, 376 pei parameter, 745 Pentium floating-point bug, 204 percent signs (%), 82 performance counters, 139 Performance Data database, 118 Performance Data Helper function library (PDH.dll), 139 Performance Data Helper set of functions, 118 performance, of mutexes vs. critical sections, 268 Performance Options dialog box, 195 periodic timer, setting, 351 PerJobUserTimeLimit member, 131 PerProcessUserTimeLimit member, 131 persistent mechanism for communication, 104 pExceptionParam parameter, 744 pflOldProtect parameter, 435 pfnAPC parameter, 319 pfnCallback parameter, 340 pfnCur member, 574 PfnDliHook type, 576 pfnHandler parameter, 726 pfnRecoveryCallback parameter, 756 pfnStartAddr parameter of CreateRemoteThread, 621 of CreateThread, 152, 157 pfnStatusRoutine parameter, 593 pfnTimerCallback parameter, 346 pfnWorkHandler parameter, 341 pftDueTime parameter, 347 pftTimeout parameter, 352 pfUsedDefaultChar parameter, 29 _pgmptr global variable, 73 pHandler parameter of RemoveVectoredContinueHandler, 727 of RemoveVectoredExceptionHandler, 727 phObjects parameter, 245 phTargetHandle parameter, 61 physical disk drive, 290, 291 physical memory, allocating, 406, 441 physical size, returning file's, 300 physical storage address, translating virtual address to, 379 assigning or mapping, 372 committing, 421, 424, 499, 505 contents, resetting, 435 data stored in paging file on disk drive, 378 decommitting, 426 methods for determining whether to commit, 425 paging file, not maintained in, 379-380 process, currently in use by, 407 region, reserving and committing, 437

region, within, 376 thread's stack, freeing for, 459 type of, 390, 409, 411 pidd member, of DelayLoadInfo, 575 Ping server, 320 pInstance parameter, 355 platform DLLs resolving differences, 538 supported by current system, 87 plDestination parameter, 212 pliFileSize parameter, 299 pMultiByteStr parameter of MultiByteToWideChar function, 28 of WideCharToMultiByte function, 28 pNodeCount parameter, 285 pNodeInfoArray parameter, 285 pNumArgs parameter, 77 pNumBytesTransferred parameter, 141 pointer variables, 393 pointers, using to reference memory, 605 POLICY_NEW_PROCESS_MIN security token, 124 POLICY_NO_WRITE_UP security token, 124 polling, 216 pool of threads, 321 Portable Executable file format, 546 POSIX subsystem, 91, 296 PostMessage, 124 PostQueuedCompletionStatus function, 262, 330, 340 PostThreadMessage, 613 pOverlapped parameter of CancelIoEx, 310 of GetQueuedCompletionStatus function, 141 of PostQueuedCompletionStatus, 330 of ReadFile and WriteFile, 303 ppfn member, 575 ppiProcInfo parameter, 92, 102 pPreviousValue parameter, 101 ppszDestEnd parameter, 23 #pragma data_seg line, 470 pragma directive taking advantage of function forwarders, 583 tricking by using macros, 763 pragma, forcing linker to look for entry-point function, 766 pragma message Helper macro, 763 #pragma warning directive, 762 preemptive multithreaded environment, 208 preemptive multithreading operating system, 174 preemptive multithreading system, Windows as, 152 preferred base address drawbacks when module cannot load at, 588 of every executable and DLL module, 586 importance of, 589 starting on allocation-granularity boundary, 590 pReportInformation parameter, 743 Preserve Job Time When Applying Limits check box, 143 pResult parameter, 32

pReturnSize parameter, 101 primary thread, 67 closing its handle to new thread, 458 creating, 146 creating process', 563 entry-point function returning, 104 examining summation thread's exit code, 458 safely calling any of C/C++ run-time functions, 167 waiting for server thread to die, 253 primary thread object, 109 printf family functions, avoiding, 27 priorities abstract view, 188-191 programming, 191-198 of threads, 187 priority 0, for thread, 188 priority boosting, 194 priority classes assigning for applications, 188 for processes, 191 processes attempting to alter, 192 set by fdwCreate, 95 specifying for all processes, 131 supported by Windows, 188 priority inversion, avoiding, 196 priority levels, 189 priority number, for every thread, 187 PriorityClass member, 131 private address space, 464 Private memory region type, 386 private namespace, 53-60 creating, 59 as directory to create kernel objects, 60 name only visible within process, 60 PRIVATE_NAMESPACE_FLAG_DESTROY, 60 private regions, identifying data in, 387 PrivateUsage field, 408 privileges elevation, 113 PrivilegesToDelete member, 135 problem reports closing, 748 creating and customizing, 740-751 customizing within process, 738-739 default parameters for, 744 enumerating application's, 740 submission customization, 747 submitting, 746-747 types of file added to, 739 PROC_THREAD_ATTRIBUTE_HANDLE_LIST, 100, PROC_THREAD_ATTRIBUTE_PARENT_PROCESS, 100, 101 process(es) address space, 482 affinity mask, 204 allocating memory in, 623 associated with jobs, 127 attempting to commit storage over job's limit, 142

command line in, 76 communicating with process outside of job, 134 container of, 125 converting pseudohandle to real, 172 creating child, 109 creating with CreateProcess, 89 creating with suspended primary thread, 93 customizing problem reports within, 738-739 default heap, 519 defined, 67 each getting its own partition, 373 elevating automatically, 113 elevating by hand, 115-116 enumerating running, 118 environment block associated with, 77 error mode, 83 hooking other, 605 as inert, 145 minimum and maximum, 131 no leaks after termination, 107 obtaining handle to existing mutex, 266 obtaining information about job, 135 parent-child relationship, 43 placing in job, 136 priority classes, 191 private address space for each, 605 reading from and writing to another, 624 resources required by, 146 seeing integrity level of, 123 setting its own priority class to idle, 192 sharing named kernel objects, 49 silently terminating, 709 specifying maximum concurrent, 131 suspending and resuming, 176-177 terminating, 104-108, 155 terminating all in job, 136 total number of in job, 138 tracked by WCT, 281 tracking current drive and directory, 84 transferring data between different, 109 using flat address spaces, 465 virtual address space, 371 process boundaries breaking through, 605 elevating privileges on, 111 mutexes vs. critical sections, 268 process environment block. See PEB Process Explorer tool detecting new kernel objects in, 42 finding keyed event, 224 from Sysinternals, 41-43, 123, 139 PROCESS_HEAP_ENTRY structure, 533 process IDs, 103, 625 PROCESS_INFORMATION structure, 102, 103 process instance handle, 73-75 process kernel objects becoming signaled, 109 creating, 61, 89, 91, 464

process kernel objects, continued decrementing usage count, 105 living at least as long as process, 107 never changing back to nonsignaled, 242 registering wait on, 353 rules Microsoft defined for, 242 status as signaled, 107 unique ID assigned to, 103 usage count decremented, 107 PROCESS_MEMORY_COUNTERS_EX structure, 407 PROCESS_MODE_BACKGROUND_END, 196 process priority class, 191 PROCESS_QUERY_INFORMATION access right, 407 process' time usage, thread querying its, 169 PROCESS_VM_READ access right, 407 Process32First function, 119 Process32Next function, 119 PROCESSENTRY32 structure, 104 ProcessIdToSessionId function, 52 ProcessInfo application (04-ProcessInfo.exe), 119 ProcessInfo.exe tool, 592 processing time, saved by SignalObjectAndWait, 280 ProcessMemoryLimit member, 132 processor affinity, 83, 205 architecture, 396 frequency, 182 process-relative handles, 39, 43 ProgMan window, 612 Program Compatibility Assistant, 128 programmatic strings, comparing, 27 programmatic Windows error reporting, 736-738 programming priorities, 191-198 programming problem, solved by Handshake, 252 programs methods of implementing, 476 readability improved by termination handlers, 672 project management, DLLs simplifying, 537 project properties dialog box, 159 project type, selecting wrong, 70 properties, displaying for thread, 173 property page, for shortcut running Notepad, 100 Protect member, 409 protected handle, thread closing during debugging, 47 protected processes, 122 protection attribute flags, 382 associated with block, 390 not using when reserved regions, 421 protection attributes assigning, 421, 422 associated with regions, 420 changing, 434 listing of, 421 locating hard-to-find bugs, 435 for pages of physical storage, 381-383 for region, 387 specifying, 479 psa parameter of CreateFile, 294

of CreateFileMapping, 479 of CreateSemaphore, 263 of CreateThread, 151 PSAPI functions, 387 psaProcess parameter, 91-92 psaThread parameter, 91–92 PSECURITY_ATTRIBUTES parameter, 37 pseudocode _beginthreadex source code, 161 showing what stack-checking function does, 456 pseudohandle closing, 60 converting to real handle, 170-172 converting to real process handle, 172 as handle to current thread, 170 passing to Windows function, 169 returning, 59 psiStartInfo parameter, 96-102 pSize, 101 pSubmitResult parameter, 747 pszApplicationName parameter, 89-91 pszCmdLine parameter in CommandLineToArgvW, 77 not writing into, 76 of WinMain, 76 of (w)WinMain, 91 pszCommandLine parameter, 89-91 pszCurDir parameter, 95 pszDLLPathName parameter, 556–558 pszDst parameter, 82 pszFileName parameter, 478 pszModule parameter, 74 pszName parameter of CreateFile, 292 of CreateFileMapping, 482 of CreateSemaphore, 263 in functions naming kernel objects, 48 of GetEnvironmentVariable, 81 of Open* functions, 50 of OpenEvent, 249 of SetEnvironmentVariable, 83 pszPathName parameter, 560 pszSrc parameter, 82 pszSymbolName parameter, 561 pszValue parameter of GetEnvironmentVariable, 81 of SetEnvironmentVariable, 83 pTimer parameter of SetThreadpoolTimer function, 347 of TimerCallback, 346 PTP_CALLBACK_ENVIRON parameter passing, 356 of TrySubmitThreadPoolCallback, 340 PTP_CALLBACK_ENVIRON structure, 359 PTP_CALLBACK_INSTANCE data type, 355 PTP_POOL value, 356, 358 PTP_WORK object, 342 pulNumEntriesRemoved parameter, 326 pulRAMPages parameter, 441

PulseEvent function, 251, 280, 353 Put PID In Job, in Job Lab sample, 143 pvAddress parameter, 435 of VirtualAlloc. 419 of VirtualFree, 426 of WerRegisterMemoryBlock, 738 pvAddressRemote parameter, 624 pvAddressWindow parameter, 442 pvBaseAddress parameter of MapViewOfFileEx, 496 of UnmapViewOfFile, 485 pvBlkBaseAddress member, 411 pvBoundaryDescriptor, 59 pvBuffer parameter of IsTextUnicode, 32 of ReadFile and WriteFile. 303 pvBufferLocal parameter, 624 pvContext parameter, 346 of CreateThreadpoolWork function, 341 of TrySubmitThreadPoolCallback, 340 pvEnvironment parameter, 95 pvFiberExecutionContext parameter of DeleteFiber, 364 of SwitchToFiber, 363 pvMem parameter of HeapReAlloc, 527 of HeapValidate, 532 PVOID return type, 3 pvParam parameter of CreateThread, 152, 157, 171 passed to CreateThread, 158 pvParameter parameter, 756 pvRgnBaseAddress member, 411 pvTlsValue parameter, 599 pWaitItem parameter, 352 pWideCharStr parameter with MultiByteToWideChar function, 28 of WideCharToMultiByte function, 28 pWork parameter, 341 pwzCommandLine parameter, 755 pwzEventType parameter, 742 pwzExeName parameter, 737 pwzFilename parameter of WerRegisterFile, 739 of WerReportAddFile, 745 pwzName parameter, 743 pwzValue parameter of WerReportSetParameter, 743 of WerReportSetUIOption, 746

Q

quantums, 67, 68 QueryDepthSList function, 213 QueryInformationJobObject calling at any time, 138 retrieving completion key and completion port handle, 141 QueryInformationJobObject function, 135, 137 QueryPerformanceCounter function, 181 QueryPerformanceFrequency function, 181 QueryProcessCycleTime function, 181, 183 QueryThreadCycleTime function, 181 question mark default character, 29 queue associated with thread, 315 controlling with mutex and semaphore, 268 Queue (08-Queue.exe) application, 228 Queue (09-Queue.exe) application, 268–271 queue data structure, 269 queue implementation, 229–232 queued device I/O requests, 309 QueueUserAPC function, 318, 319 quick free, with heaps, 523

R

RaiseException function, 703-704, 728 RAM. See also memory adding to improve performance, 408 allocating, never swapped, 439 finding free page of memory in, 382 forcing heavy demand on, 437 performance boost from adding, 378 portions saved to paging file, 378 swapping to system's paging file, 435 RAM blocks accessing via address windows, 440 assigning to address windows, 439, 443 freeing, 442 unassigning current, 442 .rdata section, 470 __rdtsc intrinsic function, 181 READ attribute, 468 ReadDirectoryChangesW function, 17 reader threads, waking up all, 228 readers, distinguishing from writers, 224 reader-writer lock, 238 ReadFile function, 302, 306 ReadFileEx function, 315 read-only data, separating from read-write data, 214 read-only files as good candidates for memory-mapped files, 496 indicating, 297 read-only handle, 64 ReadProcessMemory function, 624 ReadTimeStampCounter macro, 181 real-time operating system, 174 real-time priority, 197 REALTIME_PRIORITY_CLASS, 95, 191 real-time priority class, 189, 190 real-time range, 194 Rebase.exe utility, 590 ReBaseImage function, 591-592 rebasing implementing, 591-592 modules, 586-592 recalculation fiber, 366 recalculation thread. 366 recursion counter, 265

Refresh function, 416 regFileType parameter, 739 regions in address space, 375, 379 committing physical storage within, 376 displaying blocks inside, 388 number of blocks within, 387 number of bytes reserved for, 387 protection attributes for, 387 reserving, 419, 420 reserving and committing storage to, 422 RegionSize member, 409 register on host CPU, 183 RegisterApplicationRecoverCallback function, 756 RegisterApplicationRestart function, 755 registered event, signaling using PulseEvent, 353 RegisterWaitChainCOMCallback function, 283 registry injecting DLL using, 608-609 keys, continuing environment strings, 80 registry functions modifying entries, 81 settings for WER store, 740 RegNotifyChangeKeyValue function, 357 RegOpenKeyEx function, 36 RegQueryValueEx registry function, 118 relative thread priorities, 189, 190, 193. See also thread priority boosting relative virtual address, 545, 556 release build replacing assertion dialog box, 20 setting /GS compiler switch, 457 release, of mutexes vs. critical sections, 268 released thread list, 327 ReleaseMutex function, 266, 267 called by Append, 270 system calling, 570 thread pool calling, 355 ReleaseMutex-WhenCallbackReturns function, 355 ReleaseSemaphore function, 264, 270 allowing or disallowing, 49 in finally block, 664 putting call into termination handler, 662 thread pool calling, 355 ReleaseSemaphore-WhenCallbackReturns function, 355 ReleaseSRWLockExclusive function, 224, 234 ReleaseSRWLockShared function, 225, 236 releasing region, 376, 426 Reliability and Performance Monitor, 139 .reloc section, 470 relocating, executable (or DLL) module, 588 relocations, creating image without, 589 remote process, terminating, 187 remote threads, injecting DLL using, 621-633 Remove method, 270 RemoveVectoredContinueHandler function, 727 RemoveVectoredExceptionHandler function, 727 **REP NOP instruction**, 178

reparse attribute, 296 replaceable strings, 82 ReplaceIATEntryInAllMods function, 638 ReplaceIATEntryInOneMod function, 637, 638 report generation and sending, disabling for WER, 737 report parameters, setting, 743 ReportFault function, 748 repType parameter, 743 repUITypeID parameter, 746 requests server threads consuming, 234-236 submitting to thread pools, 340 requireAdministrator value, 114 reserve argument, 151 reserved keywords Global, 53 Local, 53 Session, 53 reserved region committing storage in, 421 releasing, 426 reserved space, for thread's stack, 151 reserving, region of address space, 375 ResetEvent, 249 _resetskoflow function, 454, 459 resetting, of physical storage, 435 resource(s) counting, 262 easily manipulating, 26 leaking, 40 resource compiler, output file, 17 resource handles, initializing, 670 resource leaks avoiding, 103 eliminating potential, 487 resource locks, entering in exactly same order, 239 resource policies, 124 resource sharing, DLLs facilitating, 537 resource-only DLLs, loading, 589 restart aware applications, 754 Restart Manager API, 755 RestrictedSids member, 135 restrictions placing on job's processes, 129-135 querying job's, 135 on server clients, 125 types of, 129 ResumeThread function, 93, 175, 193 resumptive exception handling, 729 return code of 6 (ERROR_INVALID_HANDLE), 49 return statements avoiding putting into try block, 669 in finally block, 690 return value data types for Windows functions, 3 indicating error, 4 reversed file, forcing to end, 489 RgnSize member, 411

RobustHowManyToken function, 685 RobustMemDup function, 686 root directory, as current directory, 84 .rsrc section, 470 RT_MANIFEST, 114 /RTC switches, 457 /RTCs compiler switch, 27 /RTCsu compiler switch, 457 /RTCx flags, 22 RTL_SRWLOCK, 224 RTL_USER_PROCESS_PARAMETERS structure, 121 RtlUserThreadStart function calling C/C++ run-time library's startup code, 158 calling ExitProcess, 158 calling ExitThread, 158 exported by NTDLL.dll module, 157-158 prototyped as returning VOID, 158 thread's instruction pointer set to, 157 rule of thumb, for creating threads, 324 Run As Administrator command, 111 run time, detecting stack corruptions, 457 runnable threads, in concurrent model, 320 running processes, enumerating, 118 running thread, operating system memory hidden from, 371 run-time checks, 22 run-time library, 159 Russinovich, Mark, 113 RVA (relative virtual address), 545, 556

S

_s (secure) suffix, 19 SACL, 122 safe string functions always working with, 27 of C run time, 79 HRESULT values, 22 Safer. See WinSafer /SAFESEH linker switch, 381 sample applications in this book, build environment, 761-767 sandbox running applications in, 375 setting up, 129 scalable application, 289 schedulable threads with exclusive access to memory block, 250 system scheduling only, 174 scheduler not fully documenting, 188 tweaking for foreground process, 195 scheduling algorithm applications not designing to require specific knowledge of, 193 effect on types of applications run, 188 as subject to change, 188 Scheduling Lab application (07-SchedLab.exe), 197 SchedulingClass member, 130, 131, 132

scripts, 13 search algorithm, for finding DLL files, 556 Search window, invoking, 147 searching features, in Microsoft Windows Vista, 146 SEC_COMMIT flag, 504 SEC_COMMIT section attribute, 480 SEC_IMAGE section attribute, 480 SEC_LARGE_PAGES section attribute, 480 SEC_NOCACHE section attribute, 479 SEC_RESERVE flag specifying in CreateFileMapping, 504 with VirtualAlloc, 505 SEC_RESERVE section attribute, 480 secondary threads, entry-point function for, 149 section(s) attributes associated, 467 creating, 470 in every .exe or DLL file image, 467 section attributes, 479 section names beginning with period, 467 and purposes, 470 /SECTION switch, 471 secure (_s) functions, 19 secure string functions in C run-time library, 18-25 introducing, 19-22 manipulating Unicode strings via, 11 secure use of Unicode strings, 11 security access flags, 37 security access information, 36 SECURITY_ATTRIBUTES structure, 44 containing bInheritHandle field set to TRUE, 100 example, 36 for file-mapping kernel object, 479 functions creating kernel objects with pointer to, 35 initializing, 36 passed in CreatePrivateNamespace, 59 pointer, 151 pointing to, 294 for psaProcess and psaThread parameters, 91 SECURITY_BUILTIN_DOMAIN_RID parameter, 59 security confirmation dialog boxes, 111 security descriptor, protecting kernel objects, 35 security, for kernel objects, 35-37 security identifier (SID), 54, 59 security limit restriction, 129 security restrictions, 135 SecurityLimitFlags member, 135 segment registers, identifying, 185 SEH frame in BaseThreadStart, 707 handling exceptions, 158 placing around thread function, 165 SEH (structured execution handling) available in any programming language, 727 burden falling on compiler, 659 consisting of two main capabilities, 660

SEH (structured execution handling), continued developers avoiding, 702 frame around thread function, 158 illustrating most confusing aspects of, 688 order of execution, 689 using, 425, 716 in Windows, 381 as Windows-specific, 703 SEH termination handler, adding, 668 SEHTerm application (23-SEHTerm.exe), 673-676 Select Columns dialog box, 41 Select Process Page Columns dialog box, 40 SEM_FAILCRITICALERRORS flag, 83 SEM_NOALIGNMENTFAULTEXCEPT flag of SetErrorMode, 83 setting, 392 SEM_NOGPFAULTERRORBOX flag, 83, 132 SEM_NOOPENFILERRORBOX flag, 83 semaphore(s) successfully waiting on, 270 synchronizing execution, 43 semaphore kernel objects, 262-264 creating, 263 process-relative handles to, 263 rules for, 263 tracking number of elements in queue, 270 SEMAPHORE_MODIFY_STATE, 49 SendMessage, 124, 281 sensitive user data, 745 serial model, 320 serial port opening, 291 use of, 290 serialized calls, to DllMain, 567-570 server applications, requiring more and more memory, 439 server clients, restrictions on, 125 server threads calling GetNewElement, 231 calling Remove method, 270 consuming requests by, 234-236 created by Queue, 229 Server Threads list box, 269 ServerThread function, 252 service applications models for architecting, 320 using concurrent model, 320 using single I/O completion port, 325 service pack major version number of, 87 minor version number of, 87 Session 0 isolation, 52 Session, as reserved keyword, 53 Set Affinity menu item, 206 _set_abort_behavior function, 705 _set_invalid_parameter_handler, 20 SetCommConfig, 291 SetCriticalSectionSpinCount function, 223

SetCurrentDirectory function, 85 SetDllDirectory function, 557 SetEndOfFile function, 302, 332, 489, 509 SetEnvironmentVariable function, 83, 85 SetErrorMode function, 83, 392 SetEvent function, 249, 312 called by debugger, 715 making waiting threads schedulable, 251 primary thread calling, 250 thread pool calling, 355 SetEvent-WhenCallbackReturns function, 355 SetFileCompletionNotificationModes function, 313, 327 SetFileInformationByHandle function, 196 SetFilePointer function, 509 SetFilePointerEx function, 301-302, 332 SetHandleInformation function, 46 SetInformationJobObject function, 129, 141 failing, 135 resetting job object, 140 /setintegritylevel command-line switch, 123 SetLastError function, 7 SetMailslotInfo function, 292 SetPriorityClass function, 192 to alter priority, 192 child process calling, 191 passing PROCESS_MODE_BACKGROUND_END, 196 process calling, 131 SetProcessAffinityMask function, 203 SetProcessPriorityBoost function, 194 SetProcessWorkingSetSize function, 308 SetSize member, 131 SetThreadAffinityMask function, 204 SetThreadContext function, 186, 187 SetThreadIdealProcessor function, 205 SetThreadpoolCallbackCleanupGroup function, 359 SetThreadpoolCallbackLibrary function, 358 SetThreadpoolCallbackPool function, 358 SetThreadpoolCallbackRunsLong function, 358 SetThreadpoolThreadMaximum function, 357 SetThreadpoolThreadMinimum function, 357 SetThreadpoolTimer function, 346, 347, 348 SetThreadpoolWait function, 352, 353 SetThreadPriority function, 193, 196 SetThreadPriorityBoost function, 194 SetThreadStackGuarantee function, 454 SetUnhandledExceptionFilter function, 705, 706 setup program, beginning with one floppy, 380 SetWaitableTimer function, 257-259 SetWindowLongPtr function, 605, 610 SetWindowsHookEx function, 609 SHARED attribute, 468 shared objects, protecting against hijacking, 53 shared sections, Microsoft discouraging, 472 ShellExecuteEx function, 115, 116 SHELLEXECUTEINFO structure, 115 SHGetStockIconInfo, 118

shield icon, 113, 118 ShlwApi.h file, 24 ShowWindow function, 97, 99 SID (security identifier), 54, 59 side effect, applying, 246 side-by-side assemblies, 586 SidsToDisable member, 135 signal function, 165, 167, 168 signaled object, 109 signaled process kernel object, 242 signaled state, 241, 249 SignalObjectAndWait function, 279, 317 single-CPU machines, avoiding spinlocks on, 211 single-tasking synchronization, 108 Singleton application, 03-Singleton.exe, 53 Singleton.cpp module, 54 size argument (_aligned_malloc), 210 size parameter, for C+ new operator, 529 SIZE_T variable, 101 sizeof operator, 22 sizeof(wchar_t), 28 Sleep, 177, 211 Sleep field, 197 SleepConditionVariableCS function, 227 SleepConditionVariableSRW function, 227 releasing g_srwLock passed as parameter, 234 thread blocking on, 236 SleepEx function, 317 slim reader-writer locks. See SRWLock SNDMSG macro, 776 socket, 290, 291 soft affinity, 203 software exceptions, 679, 702-704 creating codes for, 765 generating your own, 704 trapping, 703 Software Restriction Policies. See WinSafer SORT_STRINGSORT flag, 25 source code assuming pointers to be 32-bit values, 374 compiling, 15 examining samples to demonstrate SEH, 660 modules, 538 of simple program calling GetSystemInfo, 398 space-delimited tokens, 685 spaces, in environment block, 80 sparse file feature, of NTFS, 505 Sparse Memory-Mapped file sample application. See MMF Sparse application Spawn CMD, in Job button, 143 special access protection attribute flags, 382 spin count, setting, 222 spin loops, executing, 178 spinlocks assuming protected resource access, 212 critical sections and, 222 implementing, 211 as useful on multiprocessor machines, 212

using with critical sections, 223 wasting CPU time, 211 Spreadsheet sample application, 716-722 spreadsheets performing recalculations in background, 148 sharing, 504 Spy++ filtered behavior with, 124 running inside job, 133 SRWLock (slim reader-writer lock), 224-227 acquired exclusive mode, 234 acquiring, 232 acquiring in shared mode, 236 features missing in, 225 performance, 227 performance and scalability boost using, 225 reading volatile long value, 226 synchronization mechanism not tracked, 281 SRWLOCK structure, 224, 225 st_prefix, for static TLS variables, 602 stack created by CreateFiber, 362 detecting corruption at run time, 457 emptying, 213 limit, setting, 151 overflows, recovering gracefully from, 454 owned by each thread, 151 removing top element of, 213 returning number of elements stored in, 213 space required for function, 456 storage, increasing, 452 /STACK option, 451 stack pointer register of thread, 451 in thread's context, 157 stack region bottommost page always reserved, 455 for thread, 451, 452 /STACK switch, 151 stack underflow, 455 stack-based variables, 597 StackCheck, 457 stack-checking function, 456-457 Standard phonetic Unicode character set, 13 Standard User, 116 START command, 192 start glass cursor, 99 STARTF_FORCEOFFFEEDBACK flag, 98 STARTF_FORCEONFEEDBACK flag, 98, 99 STARTF_RUNFULLSCREEN flag, 98 STARTF_USECOUNTCHARS flag, 98 STARTF_USEFILLATTRIBUTE flag, 98 STARTF_USEPOSITION flag, 98 STARTF_USESHOWWINDOW flag, 97, 98 STARTF_USESIZE flag, 98 STARTF_USESTDHANDLES flag, 98 StartRestrictedProcess function, 125-128, 129, 141 StartThreadpoolIo, 354

STARTUPINFO structure members of, 97-98 obtaining copy of, 102 psiStartInfo pointing to, 96 zeroing contents of, 96 STARTUPINFOEX structure members of, 97-98 passing to psiStartInfo parameter, 95 psiStartInfo pointing to, 96 role of, 99 zeroing contents of, 96 STARTUPINFOEX variable, 102 starvation, 187 starved thread, 221 starving thread dynamically boosting priority of, 195 SwitchToThread scheduling, 178 State member, 409 static data not shared, 465-467 sharing, 467-474 static TLS, 602 static variables declaring, 153 in DLL, 538 with heaps in C++, 529 STATUS_ACCESS_VIOLATION exception, 525 STATUS_NO_MEMORY exception, 223, 525 STATUS_NO_MEMORY software exception, 703, 704 STATUS_PENDING, 307 __STDC_WANT_SECURE_LIB__symbol, 27 __stdcall (WINAPI) calling convention, 546 STILL ACTIVE exit code, 107, 156, 157 STILL_ACTIVE identifier, 156 StopProcessing function, 237 storage, committing, 421, 692 strcpy function, 18, 684 string(s) comparing, 24 compiling as Unicode string, 14 controlling elimination of duplicate, 90 kinds of, 27 translating between Unicode and ANSI, 27-29 working with, 26 string arithmetic problems, 26 string conversions, 15 string functions, 24-25 string manipulation functions, 27 string manipulations, 22-24 string pointers, 79 StringCbCopyN, 79 StringCchCopy, 22 StringCchCopyN, 79 String.h calling functions defined in, 20 included with StrSafe.h, 19 StringReverseA function, 30 StringReverseW function, 29, 30 string-termination zero, 76 strlen function, 12, 17

_strrev C run-time function, 488 STRSAFE_E_INSUFFICIENT_BUFFER, 22, 23 STRSAFE FILL BEHIND NULL, 23, 24 STRSAFE_FILL_BYTE macro, 24 STRSAFE_FILL_ON_FAILURE, 23 STRSAFE_IGNORE_NULLS, 23 STRSAFE_NO_TRUNCATION, 23, 24 STRSAFE NULL ON FAILURE, 23 StrSafe.h file, 18, 19 strtok C/C++ run-time function, 685 structure, making volatile, 217 structured exception handling. See SEH (structured exception handling) structured exceptions, vs. C++ exceptions, 727 _stscanf_s, 46 subclass procedure, 606, 607 subclassing, 605 SubmitThreadpoolWork function, 341 /SUBSYSTEM linker switch, 70 /SUBSYSTEM:CONSOLE, 5.0 switch, 589 /SUBSYSTEM:CONSOLE linker switch, 69, 70 /SUBSYSTEM:WINDOWS, 5.0 switch, 589 /SUBSYSTEM:WINDOWS linker switch, 69, 70 successful wait side effects, 246-247, 249 suites, available on system, 87 Sum function, 457, 458 Summation (16-Summation.exe) sample application, 457 SumThreadFunc, 458 SuperFetch, 197 surrogates, 12, 13 Suspend button, 198 suspend count of thread, 175 suspend mode, 259 "Suspend Process" feature, 176 suspended state, 175 suspended thread, 174 SuspendProcess function, 176 SuspendThread function, 175, 185 SW_SHOWDEFAULT, 99 SW_SHOWMINNOACTIVE, 99 SW_SHOWNORMAL, 99 /SWAPRUN:CD switch, 380 /SWAPRUN:NET switch, 380 switch statements, 773, 774 SwitchDesktop function, 133 SwitchToFiber function, 363, 366 SwitchToThread function, 178 synchronizable kernel objects, 277 synchronization kernel object mutex, 226 synchronization mechanism performance, 225 synchronization primitives, 167 synchronization, single-tasking, 108 synchronous device I/O, 302-305 synchronously performed asynchronous request, 327 SysInfo.cpp listing, 398 Sysinternals Process Explorer tool, 41-43 WinObj tool from, 33

system creating and initializing threads, 156 making sections of code execute, 689 setting process' exit code, 105 walking up chain of try blocks, 694 system DLLs, loaded at random address, 122 System Idle Process, 103 SYSTEM_INFO structure dwPageSize field of, 397 members of, 396 passing to GetSystemInfo, 395 System level of trust, 122 SYSTEM_LOGICAL_PROCESSOR_INFORMATION structures, 214 SYSTEM_MANDATORY_LABEL_NO_READ_UP resource policy, 124 SYSTEM_MANDATORY_LABEL_NO_WRITE_UP resource policy, 124 System Properties dialog box, 195 system values, retrieving, 395 System Variables list, 81 system version, determining, 85-88 SystemParametersInfo function, 133 system's affinity mask, 204 SYSTEMTIME structure, initializing, 258 szBuffer, 21 szCSDVersion member, 87 szDll member, 574

Т

_T macro, 26 Task Manager allowing user to alter process' CPU affinity, 206 determining if objects leak, 40 obtaining process' ID, 625 process ID of, 113 Processes tab, 192 TChar.h, macro defined by, 18 _tcslen, 18 _tcspy_s, 21 _tcstok_s function, 597 TEBs (thread environment blocks), 376 temporary file, 298 Terminal Services, 51-53 Terminate Processes button, in Job Lab sample, 143 TerminateJobObject function, 137 TerminateProcess function, 104, 106, 155 as asynchronous, 107 calling from WerFault.exe, 710 using only as last resort, 564 TerminateThread function, 153, 154 killing any thread, 155 threads calling, 567 terminating processes, 107-108, 155 threads, 153-156 termination handlers, 660 reasons for using, 672

simplifying programming problems, 667 understanding by example, 660-676 test runs, WER dialog boxes breaking and stopping, 712 test-and-set operation, performing atomically, 264 text determining if ANSI or Unicode, 31 used by child's console window, 97 text files line termination, 488 no hard and fast rules as to content, 31 TEXT macro, for literal characters and strings, 26 .text section, 467, 470 text strings as arrays of characters, 26 coding, 12 .textbss section, 470 th32ParentProcessID member, 104 thead synchronization overhead, 522 theme support, enabling, 766 ThisPeriodTotalKernelTime member, 137 ThisPeriodTotalUserTime member, 137 thrashing, 378 thread(s) accessing data in paging file, 378 accessing data in process' address space, 378 accessing data in RAM, 378 accessing data via single, 215 accessing memory from its process, 371 all in process dying, 107 appropriate use of, 146-149 assigning to CPUs, 148 attempting to access storage in guard page, 452 attempting to release mutex, 267 attempting to write to shared blocks, 382 calling CreateProcess, 89 calling wait function, 264 communicating with all in a pool, 330 communicating with devices, 290 components of, 145 contending for critical section, 223 creating and destroying, 328 creating and initializing, 156 creating in other processes, 621 deadlock issues when stopping, 236–238 determining number in pool, 324 ensuring mutual exclusive access, 265 executing code in processes, 67 execution times, 179-183 experiment with different numbers of, 328 forcing out of wait state, 319 forcing to specific CPU, 205 gaining access to shared resource, 266 guaranteeing to run, 174 helping partition work, 289 higher-priority preempting lower, 187 incrementing semaphore's current count, 264 issuing asynchronous I/O request to device, 305

thread(s), continued jumping from user-mode to kernel-mode code, 280 keeping busy, 289 managing creation and destruction of, 339 maximum number runnable at same time, 322 need to communicate with each other, 207 not allowing to waste CPU time, 215 with nothing to do, 174 notified only once of stack overflow exception, 459 notifying of task completion, 207 other processes gaining access from, 248 ownership tracked by mutexes, 267 pool, how many in, 328 priorities, 187 process kernel object, referring to, 169 relative time quantum difference in job, 131 releasing mutex, 266 requiring less overhead than processes, 146 running independently, 3 sharing kernel object handles, 145 sharing kernel objects in different processes, 43 sharing single address space, 145 solving some problems while creating new ones, 149 stack storage guaranteed for, 151 staying on single processor, 203 suspend count, 175 suspending and resuming, 175 suspending themselves, 278 switching to another, 178 synchronizing using event kernel objects, 249 synchronizing with kernel objects, 241 synchronous I/O operations, waiting to complete, 304 terminating, 153-156 thread kernel object, referring to, 169 thread times, querying, 169 tracked by WCT, 281 transitioning from user mode to kernel mode, 222 turning into fiber, 362 wait state, putting themselves into, 242 waiting for its own messages, 278 waiting for wait item, 353 when not to create, 148 when to create, 146-148 thread affinities, 205 thread environment blocks (TEBs), 376 thread functions, 158 naming, 150 return value from, 165 returning exit code, 150 returning upon termination, 154 using function parameters and local variables, 150 writing, 149-153 thread IDs, 103 identifying threads currently owning mutex, 265 storing, 153 thread internal, 156-158 thread kernel objects, 150 creating, 91, 157

decrementing usage count of, 154 freeing, 102 grabbing current set of CPU registers, 185 never returning to nonsignaled state, 242 rules Microsoft defined for, 242 selecting schedulable, 173 signaled state, 156 system creating, 89 unique identifiers assigned to, 103 usage count decremented, 156 usage count for, 149 Thread Local Storage. See TLS (Thread Local Storage) THREAD_MODE_BACKGROUND_BEGIN, 196 THREAD_MODE_BACKGROUND_END, 196 thread pool(s) continuously refusing threads, 340 creating and destroying, 357 creating new, 356 customized, 356-358 destroying, 357 gracefully destroying, 358-360 how many threads in, 328 internal algorithm of, 340 I/O completion port managing, 327-328 overhead of using, 339 re-architected in Windows Vista, 339 thread pool APIs, 339 thread pooling functions, 339 thread pool I/O object, 354 thread pool wait object, 352 thread priorities, 190. See also relative thread priorities thread priority boosting disabling for threads executing spinlocks, 211 dynamically boosting, 194 thread stacks address space for, 451 building, 157 maintaining, 145 setting amount of address space for, 151 system allocating memory for, 150 thread synchronization advanced, 215-217 aspects of, 207 kernel objects and, 276-277 using device kernel object for, 311 using HeapLock and HeapUnlock, 532 THREAD_TERMINATE, 304 thread/completion port assignment, breaking, 328 threading issues, of alertable I/O, 318 ThreadObject view, 287 _threadstartex function, 162, 164 throw C++ keyword, 660 thunks, to imported functions, 622 TID parameter, 285 _tiddata block association with new thread, 165 for each thread, 162 ExitThread preventing from being freed, 166 freeing, 168

_tiddata structure associated with thread, 164 in C++ source code in Mtdll.h file, 162-164 library function requiring, 168 time quantum, 130 time slices. See quantums Time Stamp Counter (TSC), 181 time values. 179 time-based notification, 346 time-critical thread priority, 190, 193 timed intervals, calling functions at, 346-351 Timed Message Box Application (11-TimedMsgBox.exe), 348-351 TimeoutCallback function, 346 timer(s) building one-shot, 351 grouping several together, 347 one-shot signaling itself, 259 pausing, 347 setting to go off, 259 timer APC routine, 260 timer kernel object, waitable, 346 TimerAPCRoutine function, 260, 261 TimerCallback function, 347 times, applying to all threads in process, 180 TLS_MINIMUM_AVAILABLE, 598, 599 TLS_OUT_OF_INDEXES, 599 .tls section, 470, 602 TLS (Thread Local Storage), 3, 165, 597 adding to application, 600 dynamic, 598-602 slots, 601 static, 602 TlsAlloc function, 599, 600 reserving index, 599 setting returned index for all threads, 601 TlsFree function, 600 TlsGetValue function, 166, 168, 599, 600 TlsSetValue operating system function, 165, 168, 599, 600 _tmain function, 72, 146, 158 _tmain (Main) function, 69 _tmain (Wmain) function, 69 TOKEN_ELEVATION_TYPE enumeration, 118 TOKEN_MANDATORY_POLICY_OFF, 124 TOKEN_MANDATORY_POLICY_VALID_MASK, 124 TokenElevationType parameter, 118 TokenElevationTypeDefault value, 118 TokenElevationTypeFull value, 118 TokenElevationTypeLimited value, 118 TokenLinkedToken, 118 TokenMandatoryPolicy, 123 ToolHelp functions added to Windows since Windows 2000, 119 allowing process to query its parent process, 104 enumerating all modules, 638 enumerating list of threads in system, 177 enumerating process' heaps, 531

utility, using to produce, 119 ToolHelp32, 282 TotalKernelTime member, 137 TotalPageFaultCount member, 137 TotalPageFile, 407 TotalPhys, 406 TotalProcesses member, 138 TotalTerminatedProcesses member, 138 TotalUserTime member, 137 TotalVirtual, 407 TP_CALLBACK_ENVIRON structure, 358, 359 TP_TIMER object, 347 tracking resources to be freed, 670 TriggerException function, 748 Trojan DLL, injecting DLL with, 633 TRUNCATE_EXISTING value, 294 truncation, 22 trust, levels of, 122 <trustInfo> section, in RT_MANIFEST, 114 try block avoiding code causing premature exits from, 662 avoiding statements causing premature exit of, 666 compiler not having to catch premature exits from, 666 followed by either finally block or except block, 679 returning prematurely from, 666 statements discouraged in, 680 __try keyword, 660 _try/_except construct, 637 TryEnterCriticalSection function, 221 try-except blocks coding examples, 680 exception filter of, 749 try-finally blocks explicitly protecting, 664 nesting inside try-except blocks, 679 TrySubmitThreadpoolCallback, 340 _tWinMain function, 69, 146, 158 with _UNICODE defined, 71 entry-point function for sample applications, 766 without _UNICODE, 72 two-file two-buffer method, 476 Type member, 409

U

UAC. See User Account Control feature uCodePage parameter of MultiByteToWideChar function, 27 of WideCharToMultiByte function, 28 uFlags parameter, 313 UI restriction, 134 UIPI (User Interface Privilege Isolation), 124 UIRestrictionsClass member, 133 ULARGE_INTEGER structure, 299, 300 ulCount parameter, 326 ullAvailExtendedVirtual member, 405 ulRAMPages parameter, 442 __unaligned keyword, 393 UNALIGNED macro, 393 unaligned references, handling silently, 258 UNALIGNED64 macro, 393 unallocated address space, 375 unelevated process, 113 unhandled exception, 705 occurring in excluded application, 737 occurring in kernel, 713 unhandled exception dialog box, turning off, 132 unhandled exception filter, 705 unhandled exception reports comparing with database of known failures, 733 saved on user's machine, 733 UnhandledExceptionFilter function, 705, 707 as default filter, 706 EXCEPTION_CONTINUE_SEARCH returned by, 710 execution steps inside, 707-713 retrieving output of, 750 WER interactions and, 710-713 UnhookWindowsHookEx function, 610 Unicode builds, enforcing consistency with, 762 character sets and alphabets, 13 consortium, 12 converting to non-Unicode equivalents, 16 full description of, 12 reasons for using, 26 standard, 12 supported in Windows Vista, 11 UNICODE and _UNICODE symbols defined in CmnHdr.h, 762 specifying both, 27 Unicode character and string, declaring, 14 Unicode code points, 13 Unicode functions in C run-time library, 17 in Windows, 15-17 Unicode strings always using, 11 converting to ANSI, 15 separating into separate tokens, 76 UNICODE symbol, 16, 27, 762 _UNICODE symbol, 18, 27, 762 Unicode Transformation Format. See UTF Unicode versions of entry point functions, 150 of functions, 17, 22 unique identifier, assigned to objects, 103 unique systemwide ID, 170 UNIX porting code to Windows, 361 threading architecture library of, 361 unmapping, from process' address space, 485-486 UnmapViewOfFile function, 485, 486 unnamed (anonymous) kernel object, 48 unnamed kernel objects, 53 unpageable memory, 423

UNREFERENCED_PARAMETER macro, 76 UnregisterApplicationRestart function, 755 unresolved external symbol error, 70 unsuccessful calls, object states never altered for, 246 Update Speed, changing to Paused, 42 UpdateProcThreadAttribute function, 101 upper bound, for number of threads, 321 usage count incrementing kernel object's, 45 of kernel objects, 35 of semaphore object, 263 user(s) changing priority class of process, 192 choosing to optimize performance for programs, 195 running Run As Administrator command, 111 starting applications, 99 User Account Control (UAC) feature, 110 forcing applications to run in restricted context, 37 security confirmation dialog boxes, 111 taking advantage of, 110 usurping WinSafer, 558 User CPU Limit, 144 user data, sensitive, 739 user interface components. See window (objects) User Interface Privilege Isolation (UIPI), 124 user interface thread, 149 user mode time, 131 User object handles, 155 User objects differentiating from kernel objects, 37 owned by thread, 155 preventing processes in job from using, 133 user rights, granting, 423 user strings, 27 user time, 179 User timers, 262 User32.dll library, 537, 608 UserHandleGrantAccess function, 134 user-interface-related events, 262 user-mode code, fibers implemented in, 361 user-mode context, as stable, 185 user-mode CPU time, used by job processes, 137 user-mode partition, 373-375 user-mode thread synchronization, limitations of, 241 USERPROFILE environment variable, 82 UTF (Unicode Transformation Format), 12, 13 utility applications, using IDs, 103 Utimers, 262

V

variables avoiding exporting, 542 creating to manage thread pool, 328 in default data section, 470 in multithreaded environments, 160 sharing, 467, 471, 472 storing handles for kernel objects, 40 in their own section, 471 VcppException (ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND), 574 VcppException (ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND), 574 VDM (Virtual DOS Machine), 94 vectored exception handling (VEH), 726 VEH exception handler function, 727 VEH list, calling functions before any SEH filter, 726 VER_BUILDNUMBER flag, 87 VER_MAJORVERSION flag, 87 VER_MINORVERSION flag, 87 VER_PLATFORM_WIN32_NT, 87 VER_PLATFORM_WIN32_WINDOWS, 87 VER_PLATFORM_WIN32s, 87 VER_PLATFORMID flag, 87 VER_PRODUCT_TYPE information, comparing, 88 VER_SERVICEPACKMAJOR flag, 87 VER_SERVICEPACKMINOR flag, 87 VER_SET_CONDITION macro, 87 VER_SUITENAME flag, 87, 88 VerifyVersionInfo function, 86-88 Version.txt file, 736 very large memory (VLM) portion, 405 view mapping for large file, 494 mapping into address space, 483 View Problem History link, 734, 741 virtual address, translating to physical storage address, 379 virtual address space, 372 bytes reserved in, 407 creating for process, 146 partitions of, 372-375 for process, 371 Virtual DOS Machine (VDM), 94 virtual memory, 419 address for DLL's g_x variable, 587 functions, using instead of heaps, 519 loading code and data for application into, 465 making available, 377 retrieving dynamic information about state of, 404 virtual memory allocation sample application. See VMAlloc.cpp listing virtual memory map, 383-387 Virtual Memory section, Change button, 380 virtual memory techniques advantages of, 424 combining with structured exception handling, 692 committing physical storage, 425 VirtualAlloc function, 375, 376, 378, 419, 422 to commit physical storage, 421, 437, 505 passing PAGE_WRITECOPY or PAGE_EXECUTE_WRITECOPY, 382 to reserve region, 428 rounding up when resetting storage, 436 using to allocate large blocks, 526 VirtualAllocEx function, 623 VirtualAllocExNuma function, 421

VirtualFree function, 376, 377, 426 decommitting storage, 427, 505 VirtualFreeEx function, 624 VirtualMemoryStatus function. See GlobalMemoryStatus function VirtualProtect function, 435 VirtualQuery function, 408, 429 determining if physical storage has been committed, 425 filling MEMORY_BASIC_INFORMATION structure, 631 limitations of, 410 VirtualQueryEx function, 409 Visual Studio creating application project, 69 debugger, 729 defining UNICODE by default, 16 Error Lookup utility, 6 linker, default base address, 74 shipping with C/C++ run-time libraries, 159 wizards, 72 Visual Studio IDE, 148 Visual Studio Project Properties dialog box, 589 VLM portion, of virtual address space, 405 VMAlloc.cpp listing, 427-429 VMMap application (14-VMMap.exe), 415-417 memory map of, 383 performing tests, 429 running, 119 using SEH to commit storage, 692 VMMap menu item, in ProcessInfo, 119 VMQuery function, 410-412, 416 VMQUERY structure, 410 VMQuery.cpp file, 411-412 VMStat application (14-VMStat.exe), 406-408 VOID return type, 3 volatile type qualifier, 216 vsjitdebugger.exe, 714, 715 _vstprintf_s function, 238

W

WAIT_ABANDONED_0 value, of WaitResult, 353 WAIT_ABANDONED, returned by wait function, 267 wait chain defined, 282 node object types, 286 session, opening, 283 traversal of, 282 Wait Chain Traversal (WCT) cancelling, 284 ParseThread method and, 284 set of functions, 281 synchronization mechanisms tracked by, 281 WAIT_FAILED return value, 244 wait function, 266 checking semaphore's current resource count, 264 returning special value of WAIT_ABANDONED, 267 Wait functions, 243

WAIT_IO_COMPLETION return value, 318 wait item, 353 WAIT_OBJECT_0 value, 244, 353 wait state forcing thread out of, 319 LeaveCriticalSection never placing thread in, 222 never allowing calling thread to enter, 221 placing thread in, 215, 221 threads placing themselves into, 243 WAIT_TIMEOUT value, 244, 353 waitable timer objects, creating, 257 waitable timers, 256 comparing with User timers, 262 creating, 256, 346 queuing APC entries, 260-261 WAITCHAIN_NODE_INFO structure defined in wct.h header file, 286 filling array of, 285 WCT filling up, 287 WaitForDebugEvent function, 176, 279 WaitForInputIdle function, 278 WaitForMultipleObjects function, 244, 311, 313 calling with array, 237 not supported, 287 performing operations atomically, 246 Remove method calling, 270 return value of, 245 successful call altering state of object, 246 using one thread per 64 kernel objects, 352 working atomically, 247 WaitForMultipleObjectsEx function, 317 WaitForSingleObject function, 107, 109, 140, 155, 243, 244, 253, 311, 312, 458, 568 not calling inside any DLL's DllMain function, 570 return value of, 244 successful call altering state of object, 246 threads calling, 250 WaitForSingleObjectEx function placing thread in alertable state, 317 suspending thread, 319 waiting on timer twice, 261 WaitForThreadpoolIoCallbacks function, 355 WaitForThreadpoolTimerCallbacks function, 347, 351 WaitForThreadpoolWaitCallbacks function, 353 WaitForThreadpoolWork function, 353 WaitForThreadpoolWorkCallbacks function, 341 waiting thread queue, 325 WaitResult parameter, 352 WaitResult values, possible, 353 WakeAllConditionVariable function, 227, 234, 237 WakeConditionVariable function, 227 with &g_cvReadyToProduce, 236 call to, 234 wargy variable, 73, 76 warning level 4, sample applications using, 762 Watch window, 5 wchart_t data type, 13 wcslen function, 17

wcspy function, 18 WCT. See Wait Chain Traversal (WCT) WCT_ASYNC_OPEN_FLAG value, 284 WCT_OBJECT_STATUS enumeration, 287 WCT_OBJECT_TYPE enumeration, 286 WCT_OUT_OF_PROC_COM_FLAG, 285 WCT_OUT_OF_PROC_CS_FLAG, 285 WCT_OUT_OF_PROC_FLAG, 285 WctAlpcType node object type, 286 WctComActivationType node object type, 286 WctComType node object type, 286 WctCriticalSelectionType node object type, 286 WctMutexType node object type, 286 WCTP_GETINFO_ALL_FLAGS, 285 WctProcessWaitType node object type, 286 WctSendMessageType node object type, 286 WctThreadType node object type, 286 WctThreadWaitType node object type, 286 WctUnknownType node object type, 286 Web browsers, communicating in background, 148 _wenviron global variable, 73 WER API, 733 WER code, running inside faulting process, 664 WER console, 733-736 listing problems, 712 opening, 710, 712 problem report displayed in, 735 showing each application crash, 734 View Problem History link, 741 WER_DUMP_NOHEAP_ONQUEUE flag, 745 WER_E_NOT_FOUND, 737 WER_FAULT_REPORTING_FLAG_QUEUE_UPLOAD flag, 737 WER_FAULT_REPORTING_FLAG_QUEUE flag, 737 WER_MAX_REGISTERED_ENTRIES, 739 WER problem report, manually generating, 751 WER_REPORT_INFORMATION structure, 743 WER_SUBMIT_OUTOFPROCESS_ASYNC flag, 747 WER_SUBMIT_RESULT variable, 747 WER (Windows Error Reporting), 710-713 creating, customizing, and submitting problem report to, 740 default parameters for noncustomized report, 744 details of files generated by, 736 generating problem reports silently, 712 notifying to not restart application, 755 restarting applications automatically, 754 shifting control to, 454 telling not to suspend other threads, 736 Wer* functions, accepting only Unicode strings, 742 WerAddExcludedApplication function, 737 WerConsentApproved value, 746 WerConsentDenied value, 746 WerConsentNotAsked value, 746 WerFault.exe application, 710, 713 WerGetFlags function, 737 WerRegisterFile function, 738, 739 WerRegisterMemoryBlock function, 738

WerRemoveExcludedApplication function, 738 WerReport* functions, 751 WerReportAddDump function, 740, 744 WerReportAddFile function, 740, 745 WerReportCloseHandle function, 740, 748 WerReportCreate function, 740, 742 WerReportSetParameter function, 740, 743 WerReportSetUIOption function, 740, 746 WerReportSubmit function, 740, 746-747 WerSetFlags function, 736 WerSvc, exception set to, 710 WerUnregisterFile function, 739 WerUnregisterMemoryBlock function, 739 WH_GETMESSAGE hook installing, 609 unhooking, 613 while loop, spinning, 211 wide characters, 16, 622 wide-character string, converting, 28 WideCharToMultiByte function, 27, 28, 29 Win32 Console Application, 70 Win32 Exceptions, 729 _WIN32_WINNT symbol, 761 WinDbg, 121 window (objects) owned by thread, 155 sharing same thread, 149 subclassing, 605 window procedures, writing with message crackers, 773 window title, for console window, 97 WindowDump utility, 124 Windows operating system data types, 15 designed to work with extremely large files, 299 devices supported by, 290 features available only to DLLs, 538 internal data structures for managing TLS, 598 memory architecture, 371-393 as preemptive multithreading system, 152 scheduling concurrent threads, 68 string functions, 24-25 Unicode and ANSI functions in, 15-17 Windows 32-bit On Windows 64-bit, 397 Windows 98, 94 Windows API backward compatibility with 16-bit Windows, 17 exposing abstract layer over system's scheduler, 188 Windows application programming interface (API), 537 Windows applications. See applications Windows Error Reporting. See WER (Windows Error Reporting) Windows Explorer process associated to dedicated job, 128 separate thread for each folder's window, 149 Windows functions. See functions Windows header file (WinNT.h), 15 CONTEXT structure, 157

defining data types, 14

Windows heap functions, 703 Windows hooks. See hooks Windows Indexing Services, 146 Windows Integrity Mechanism, 122 Windows Management Instrumentation (WMI), programmatic configuration of BCD, 206 Windows Notepad application. See Notepad Windows problems, getting in WER console, 713 Windows Quality Online Services Web site, 733 Windows Reliability and Performance Monitor, 392 Windows system directory never touching, 585 placing DLLs in, 608 Windows Task Manager. See Task Manager Windows Vista cancelling pending synchronous I/O, 304 CPU time charged for thread, 181 dialog boxes when exception is passed, 709 dialog box for unhandled exception, 674 error reporting in separate process, 664 extended mechanisms available in, 174 functions with both Unicode and ANSI versions, 17 I/O requests, expecting large number of, 326 mapping of priority classes and relative thread priorities, 190 Notepad File Save As dialog box, 31 raising security bar for end user, 110 soft affinity assigning threads to processors, 203 source code for CreateWindowExA, 16 starting applications in new session, 52 synchronous I/O, features added to improve, 304 testing whether host system is, 88 thread pool re-architected, 339 threads issuing requests and terminating, 330 unhandled exception, major rearchitecture of, 675 UNICODE macro, if application doesn't define, 17 Windows XP exception passed to UnhandledExceptionFilter, 709 message for unhandled exception, 674 WindowsX.h file, 775 WinError.h header file, 4, 5 WinExec function, replacing with CreateProcess, 17 WinMain function changing main to, 70 pszCmdLine parameter, 76 WinMainCRTStartup function, 69, 70 _winmajor global variable, 73 winminor global variable, 73 WinNT.h header file. See Windows header file (WinNT.h) WinObj tool, 33 WinSafer, 558 Wintellect Applications Suite product section, 741 _winver global variable, 73 WINVER symbol, 761 WM_COMMAND message macro for, 775 processing, 774 wParam containing two different values, 774

WM_INITDIALOG message-handling code, 118 WM_SETTINGCHANGE message, 81 WM_TIMER messages, 262 wmainCRTStartup function, 69, 70 word processor applications, background operations, 148 work item(s) adding to thread pool's queue, 340 creating, 341 explicitly controlling, 340-342 freeing, 342 functions implementing batch processing, 342 refusing to execute multiple actions, 341 worker threads, compute-bound or I/O-bound, 149 working set, 407 knowing process', 408 minimizing, 408 of process, 131 WorkingSetSize field, 408 WOW64, 397 wParam parameter, for messages, 774 _wpgmptr global variable, 73 wProcessorArchitecture member, 396 wProcessorLevel member, 396 wProcessorRevision member, 396 wProductType member, 87 wReserved member of OSVERSIONINFOEX structure, 87 of SYSTEM_INFO, 396 writable buffer, passing as parameter, 19 writable pages, 382 write access, 708 WRITE attribute, 468 Write method, of CIOReq, 333 WriteFile function, 302, 306 WriteFileEx function, 315

WriteProcessMemory function, 624 writers, distinguishing from readers, 224 wServicePackMajor member, 87 wShowWindow member, 87 wShowWindow value, 99 wShowWindow value, 97 wSuiteMask member, 87 (w)WinMain function, writing, 75 wWinMainCRTStartup function, 69, 70

Х

x64 system, 4-KB page size, 376 x86 compiler, 393 x86 CPUs. *See also* CPU(s) dealing with unaligned data references silently, 258 handling data alignment, 391 interlocked functions asserting hardware signal, 210 page size of, 387 performing, 392 x86 system, 4-KB page size, 376 x86 Windows, larger user-mode partition in, 373–374 _XcpFilter function, 708 .xdata section, 470 XML manifest, 114

Y

YieldProcessor macro, 178

Ζ

/Zc:wchar_t compiler switch, 13 zero page thread, 188, 190 ZeroMemory, 437 zero-terminated string passing address of, 48 passing address of in pszName, 50 /ZI switch, 90

Jeffrey Richter

Jeffrey Richter is a co-founder of Wintellect (*http://www.Wintellect.com/*), a training, debugging, and consulting company dedicated to helping companies produce better software faster. Jeff has written many books, including *CLR via C#* (Microsoft Press, 2005). Jeff is also a contributing editor for *MSDN Magazine*, where he has written several feature articles and is a columnist. Jeff also speaks at various trade conferences worldwide, including VSLive!, and Microsoft's TechEd and PDC. Jeff has consulted for many companies, including AT&T, DreamWorks, General Electric, Hewlett-Packard, IBM, and Intel. For Microsoft, Jeff's code has shipped in many products including TerraServer, Visual Studio, the .NET Framework, Office, and various versions of Windows. On the personal front, Jeff holds both airplane and helicopter pilot licenses, though he never gets to fly as often as he'd like. He is also a member of the International Brotherhood of Magicians and enjoys showing friends sleight-of-hand card tricks from time to time. Jeff's other hobbies include music, drumming, model railroading, boating, traveling, and the theater. He and his family live in Kirkland, Washington.

Christophe Nasarre

Christophe Nasarre works as a software architect and development lead for Business Objects, a multinational software company that is focused on helping organizations gain better insight into their business, improving decision-making and enterprise performance through business intelligence solutions. He has worked as a technical editor on numerous Addison-Wesley,