

Microsoft

Foreword by Ben Fathi

Corporate Vice President, Windows Core Development, Microsoft Corporation

Windows[®] Internals

5
FIFTH
EDITION

Covering Windows Server[®] 2008
and Windows Vista[®]



Mark E. Russinovich
and David A. Solomon
with Alex Ionescu

Microsoft®

Windows® Internals, Fifth Edition

*Mark E. Russinovich
David A. Solomon
with Alex Ionescu*

PUBLISHED BY

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

Copyright © 2009 by David Solomon (all); Mark Russinovich (all)

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: 2009927697

Printed and bound in the United States of America.

5 6 7 8 9 10 11 12 13 14 QGT 6 5 4 3 2 1

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, Microsoft Press, Access, Active Directory, ActiveSync, ActiveX, Aero, Authenticode, BitLocker, DirectX, Excel, Hyper-V, Internet Explorer, MS, MSDN, MS-DOS, Outlook, PowerPoint, ReadyBoost, ReadyDrive, SideShow, SQL Server, SuperFetch, Visual Basic, Visual C++, Visual Studio, Win32, Windows, Windows Media, Windows NT, Windows Server, Windows Vista, and Xbox are either registered trademarks or trademarks of the Microsoft group of companies. 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 Editor: Devon Musgrave

Project Editor: John Pierce

Editorial Production: Curtis Philips, Publishing.com

Cover: Tom Draper Design

Body Part No. X14-95072

To Jim Allchin, our OS and rock star

Table of Contents

Forewordxix
Acknowledgmentsxxi
Introductionxxiii
1 Concepts and Tools.	1
Windows Operating System Versions	1
Foundation Concepts and Terms	2
Windows API	2
Services, Functions, and Routines	4
Processes, Threads, and Jobs	5
Virtual Memory	14
Kernel Mode vs. User Mode	16
Terminal Services and Multiple Sessions	19
Objects and Handles	21
Security	22
Registry	23
Unicode	23
Digging into Windows Internals	24
Reliability and Performance Monitor	25
Kernel Debugging	26
Windows Software Development Kit	31
Windows Driver Kit	31
Sysinternals Tools	32
Conclusion	32

 **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/

2	System Architecture	33
	Requirements and Design Goals	33
	Operating System Model	34
	Architecture Overview	35
	Portability	38
	Symmetric Multiprocessing	39
	Scalability	43
	Differences Between Client and Server Versions	43
	Checked Build	47
	Key System Components	49
	Environment Subsystems and Subsystem DLLs	50
	Ntdll.dll	57
	Executive	58
	Kernel	61
	Hardware Abstraction Layer	65
	Device Drivers	68
	System Processes	74
	Conclusion	83
3	System Mechanisms	85
	Trap Dispatching	85
	Interrupt Dispatching	87
	Exception Dispatching	114
	System Service Dispatching	125
	Object Manager	133
	Executive Objects	136
	Object Structure	138
	Synchronization	170
	High-IRQL Synchronization	172
	Low-IRQL Synchronization	177
	System Worker Threads	198
	Windows Global Flags	200
	Advanced Local Procedure Calls (ALPCs)	202
	Kernel Event Tracing	207
	Wow64	211
	Wow64 Process Address Space Layout	211
	System Calls	212
	Exception Dispatching	212

User Callbacks	212
File System Redirection	212
Registry Redirection and Reflection	213
I/O Control Requests	214
16-Bit Installer Applications	215
Printing	215
Restrictions	215
User-Mode Debugging	216
Kernel Support	216
Native Support	217
Windows Subsystem Support	219
Image Loader	220
Early Process Initialization	222
Loaded Module Database	223
Import Parsing	226
Post Import Process Initialization	227
Hypervisor (Hyper-V)	228
Partitions	230
Root Partition	230
Child Partitions	232
Hardware Emulation and Support	234
Kernel Transaction Manager	240
Hotpatch Support	242
Kernel Patch Protection	244
Code Integrity	246
Conclusion	248
4 Management Mechanisms	249
The Registry	249
Viewing and Changing the Registry	249
Registry Usage	250
Registry Data Types	251
Registry Logical Structure	252
Transactional Registry (TxR)	260
Monitoring Registry Activity	262
Registry Internals	266
Services	281
Service Applications	282
The Service Control Manager	300

Service Startup	303
Startup Errors	307
Accepting the Boot and Last Known Good	308
Service Failures	310
Service Shutdown	311
Shared Service Processes	313
Service Tags	316
Service Control Programs	317
Windows Management Instrumentation	318
Providers	319
The Common Information Model and the Managed Object Format Language	320
Class Association	325
WMI Implementation	327
WMI Security	329
Windows Diagnostic Infrastructure	329
WDI Instrumentation	330
Diagnostic Policy Service	330
Diagnostic Functionality	332
Conclusion	333
5 Processes, Threads, and Jobs	335
Process Internals	335
Data Structures	335
Kernel Variables	342
Performance Counters	343
Relevant Functions	344
Protected Processes	346
Flow of CreateProcess	348
Stage 1: Converting and Validating Parameters and Flags	350
Stage 2: Opening the Image to Be Executed	351
Stage 3: Creating the Windows Executive Process Object (<i>PspAllocateProcess</i>)	354
Stage 4: Creating the Initial Thread and Its Stack and Context	359
Stage 5: Performing Windows Subsystem–Specific Post-Initialization	360
Stage 6: Starting Execution of the Initial Thread	362
Stage 7: Performing Process Initialization in the Context of the New Process	363

Thread Internals	370
Data Structures	370
Kernel Variables	379
Performance Counters	379
Relevant Functions	380
Birth of a Thread	380
Examining Thread Activity	381
Limitations on Protected Process Threads	384
Worker Factories (Thread Pools)	386
Thread Scheduling	391
Overview of Windows Scheduling	391
Priority Levels	393
Windows Scheduling APIs	395
Relevant Tools	396
Real-Time Priorities	399
Thread States	400
Dispatcher Database	404
Quantum	406
Scheduling Scenarios	413
Context Switching	418
Idle Thread	418
Priority Boosts	419
Multiprocessor Systems	434
Multiprocessor Thread-Scheduling Algorithms	442
CPU Rate Limits	444
Job Objects	445
Conclusion	450
6 Security	451
Security Ratings	451
Trusted Computer System Evaluation Criteria	451
The Common Criteria	453
Security System Components	454
Protecting Objects	458
Access Checks	459
Security Descriptors and Access Control	484
Account Rights and Privileges	501
Account Rights	502

Privileges	503
Super Privileges	509
Security Auditing	511
Logon	513
Winlogon Initialization	515
User Logon Steps	516
User Account Control	520
Virtualization	521
Elevation	528
Software Restriction Policies	533
Conclusion	535
7 I/O System	537
I/O System Components	537
The I/O Manager	539
Typical I/O Processing	540
Device Drivers	541
Types of Device Drivers	541
Structure of a Driver	547
Driver Objects and Device Objects	550
Opening Devices	555
I/O Processing	562
Types of I/O	563
I/O Request to a Single-Layered Driver	572
I/O Requests to Layered Drivers	578
I/O Cancellation	587
I/O Completion Ports	592
I/O Prioritization	598
Driver Verifier	604
Kernel-Mode Driver Framework (KMDF)	606
Structure and Operation of a KMDF Driver	607
KMDF Data Model	608
KMDF I/O Model	612
User-Mode Driver Framework (UMDF)	616
The Plug and Play (PnP) Manager	619
Level of Plug and Play Support	620
Driver Support for Plug and Play	621

Driver Loading, Initialization, and Installation	623
Driver Installation.	632
The Power Manager.	636
Power Manager Operation.	638
Driver Power Operation	639
Driver and Application Control of Device Power	643
Conclusion.	644
8 Storage Management.	645
Storage Terminology	645
Disk Drivers	646
Winload	646
Disk Class, Port, and Miniport Drivers	647
Disk Device Objects.	650
Partition Manager	651
Volume Management	652
Basic Disks	653
Dynamic Disks	656
Multipartition Volume Management	661
The Volume Namespace.	667
Volume I/O Operations.	674
Virtual Disk Service	675
BitLocker Drive Encryption	677
BitLocker Architecture.	677
Encryption Keys	679
Trusted Platform Module (TPM)	681
BitLocker Boot Process	683
BitLocker Key Recovery.	684
Full Volume Encryption Driver.	686
BitLocker Management.	687
Volume Shadow Copy Service	688
Shadow Copies	688
VSS Architecture.	688
VSS Operation	689
Uses in Windows	692
Conclusion.	698

9	Memory Management	699
	Introduction to the Memory Manager	699
	Memory Manager Components	700
	Internal Synchronization	701
	Examining Memory Usage	701
	Services the Memory Manager Provides	704
	Large and Small Pages	705
	Reserving and Committing Pages	706
	Locking Memory	707
	Allocation Granularity	708
	Shared Memory and Mapped Files	709
	Protecting Memory	711
	No Execute Page Protection	713
	Copy-on-Write	718
	Address Windowing Extensions	719
	Kernel-Mode Heaps (System Memory Pools)	721
	Pool Sizes	722
	Monitoring Pool Usage	724
	Look-Aside Lists	728
	Heap Manager	729
	Types of Heaps	730
	Heap Manager Structure	731
	Heap Synchronization	732
	The Low Fragmentation Heap	732
	Heap Security Features	733
	Heap Debugging Features	734
	Pageheap	735
	Virtual Address Space Layouts	736
	x86 Address Space Layouts	737
	x86 System Address Space Layout	740
	x86 Session Space	740
	System Page Table Entries	744
	64-Bit Address Space Layouts	745
	64-Bit Virtual Addressing Limitations	749
	Dynamic System Virtual Address Space Management	751
	System Virtual Address Space Quotas	756
	User Address Space Layout	757

Address Translation	761
x86 Virtual Address Translation	762
Translation Look-Aside Buffer	768
Physical Address Extension (PAE)	769
IA64 Virtual Address Translation	772
x64 Virtual Address Translation	773
Page Fault Handling	774
Invalid PTEs	775
Prototype PTEs	776
In-Paging I/O	778
Collided Page Faults	779
Clustered Page Faults	779
Page Files	780
Stacks	784
User Stacks	785
Kernel Stacks	786
DPC Stack	787
Virtual Address Descriptors	787
Process VADs	788
Rotate VADs	790
NUMA	791
Section Objects	792
Driver Verifier	799
Page Frame Number Database	803
Page List Dynamics	807
Page Priority	809
Modified Page Writer	812
PFN Data Structures	814
Physical Memory Limits	818
Windows Client Memory Limits	819
Working Sets	822
Demand Paging	823
Logical Prefetcher	823
Placement Policy	827
Working Set Management	828
Balance Set Manager and Swapper	831
System Working Set	832
Memory Notification Events	833

Proactive Memory Management (SuperFetch)	836
Components	836
Tracing and Logging	838
Scenarios	840
Page Priority and Rebalancing	840
Robust Performance	843
ReadyBoost	844
ReadyDrive	845
Conclusion	847
10 Cache Manager	849
Key Features of the Cache Manager	849
Single, Centralized System Cache	850
The Memory Manager	850
Cache Coherency	850
Virtual Block Caching	852
Stream-Based Caching	852
Recoverable File System Support	853
Cache Virtual Memory Management	854
Cache Size	855
Cache Virtual Size	855
Cache Working Set Size	856
Cache Physical Size	858
Cache Data Structures	859
Systemwide Cache Data Structures	860
Per-File Cache Data Structures	862
File System Interfaces	868
Copying to and from the Cache	869
Caching with the Mapping and Pinning Interfaces	870
Caching with the Direct Memory Access Interfaces	872
Fast I/O	873
Read Ahead and Write Behind	875
Intelligent Read-Ahead	875
Write-Back Caching and Lazy Writing	877
Write Throttling	885
System Threads	886
Conclusion	887

11 File Systems	889
Windows File System Formats	890
CDFS	890
UDF	891
FAT12, FAT16, and FAT32	891
exFAT	894
NTFS	895
File System Driver Architecture	895
Local FSDs	896
Remote FSDs	897
File System Operation	901
File System Filter Drivers	907
Troubleshooting File System Problems	908
Process Monitor Basic vs. Advanced Modes	908
Process Monitor Troubleshooting Techniques	909
Common Log File System	910
NTFS Design Goals and Features	918
High-End File System Requirements	918
Advanced Features of NTFS	920
NTFS File System Driver	934
NTFS On-Disk Structure	937
Volumes	937
Clusters	937
Master File Table	938
File Reference Numbers	942
File Records	942
File Names	945
Resident and Nonresident Attributes	948
Data Compression and Sparse Files	951
The Change Journal File	956
Indexing	960
Object IDs	961
Quota Tracking	962
Consolidated Security	963
Reparse Points	965
Transaction Support	965

NTFS Recovery Support	974
Design	975
Metadata Logging	976
Recovery	981
NTFS Bad-Cluster Recovery	985
Self-Healing	989
Encrypting File System Security	990
Encrypting a File for the First Time	993
The Decryption Process	998
Backing Up Encrypted Files	999
Conclusion	1000
12 Networking	1001
Windows Networking Architecture	1001
The OSI Reference Model	1001
Windows Networking Components	1003
Networking APIs	1006
Windows Sockets	1006
Winsock Kernel (WSK)	1012
Remote Procedure Call	1014
Web Access APIs	1018
Named Pipes and Mailslots	1021
NetBIOS	1027
Other Networking APIs	1030
Multiple Redirector Support	1033
Multiple Provider Router	1034
Multiple UNC Provider	1037
Name Resolution	1039
Domain Name System	1039
Windows Internet Name Service	1039
Peer Name Resolution Protocol	1039
Location and Topology	1042
Network Location Awareness (NLA)	1042
Link-Layer Topology Discovery (LLTD)	1043
Protocol Drivers	1044
Windows Filtering Platform (WFP)	1047
NDIS Drivers	1053
Variations on the NDIS Miniport	1057
Connection-Oriented NDIS	1057

Remote NDIS	1060
QoS.....	1062
Binding.....	1064
Layered Network Services	1066
Remote Access	1066
Active Directory	1066
Network Load Balancing	1068
Distributed File System and DFS Replication.....	1069
Conclusion.....	1071
13 Startup and Shutdown.....	1073
Boot Process	1073
BIOS Preboot	1073
The BIOS Boot Sector and Bootmgr.....	1077
The EFI Boot Process	1086
Initializing the Kernel and Executive Subsystems.....	1088
Smss, Csrs, and Wininit	1094
ReadyBoot.....	1099
Images That Start Automatically.....	1100
Troubleshooting Boot and Startup Problems	1101
Last Known Good.....	1101
Safe Mode.....	1101
Windows Recovery Environment (WinRE).....	1106
Solving Common Boot Problems	1109
Shutdown.....	1115
Conclusion.....	1118
13 Crash Dump Analysis	1119
Why Does Windows Crash?	1119
The Blue Screen	1120
Troubleshooting Crashes	1124
Crash Dump Files	1125
Crash Dump Generation.....	1130
Windows Error Reporting.....	1131
Online Crash Analysis	1133
Basic Crash Dump Analysis.....	1134
Notmyfault	1134
Basic Crash Dump Analysis.....	1135
Verbose Analysis	1137

Using Crash Troubleshooting Tools.	1139
Buffer Overrun, Memory Corruptions, and Special Pool	1140
Code Overwrite and System Code Write Protection	1143
Advanced Crash Dump Analysis	1144
Stack Trashes.	1145
Hung or Unresponsive Systems.	1147
When There Is No Crash Dump.	1150
Conclusion.	1152
Glossary	1153
Index	1183



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/

Foreword

It's both a pleasure and an honor for me to write the foreword for this latest edition of *Windows Internals*. Many significant changes have occurred in Windows since the last edition of the book, and David, Mark, and Alex have done an excellent job of updating the book to address them. Whether you are new to Windows internals or an old hand at kernel development, you will find lots of detailed analysis and examples to help improve your understanding of the core mechanisms of Windows as well as the general principles of operating system design.

Today, Windows enjoys unprecedented breadth and depth in the computing world. Variants of the original Windows NT design run on everything from Xbox game consoles to desktop and laptop computers to clusters of servers with dozens of processors and petabytes of storage. Advances such as hypervisors, 64-bit computing, multicore and many-core processor designs, flash-based storage, and wireless and peer-to-peer networking continue to provide plenty of interesting and innovative areas for operating system design.

One such area of innovation is security. Over the past decade, the entire computing industry—and Microsoft in particular—has been confronted with huge new threats, and security has become the top issue facing many of our customers. Attacks such as Blaster and Sasser threatened to bring the entire Internet to its knees, and Windows was at the eye of the hurricane. It was obvious to us that we could no longer afford to do business as usual, as many of the usability and simplicity features designed into Windows were being used to attack it for nefarious reasons. At first the hackers were teenagers trying to gain notoriety by breaking into systems or adding graffiti to a corporate Web site, but pretty soon the attacks intensified and went underground. The hackers became more sophisticated and evaded inspection. You rarely see headlines about viruses and worms these days, but make no mistake—botnets and identity theft are big business today, as are industrial and government espionage through targeted attacks.

In January 2002, Bill Gates sent his now-famous “Trustworthy Computing” memorandum to all Microsoft employees. It was a call to action that resonated well and charted the course for how we would build software and conduct business over the coming years. Nearly the entire Windows engineering team was diverted to work on Windows XP SP2, a service pack dedicated almost entirely to improving the security of the operating system. The Security Development Lifecycle (SDL) was developed and applied to all Microsoft products, with particular emphasis on Windows Vista as the first version of the operating system designed from the ground up to be secure. SDL specifies strict guidelines and processes for secure software development. Sophisticated tools have been developed to scan everything from source code to system binaries to network protocols for common security vulnerabilities. Every time a new security vulnerability is discovered, it is analyzed, and mitigations are developed to address that potential attack vector. Windows Vista has now been in the market for

two years, and it is by far the most secure version of Windows. Some industry analysts have pointed out that it is, in fact, the most secure general purpose operating system shipping today.

The Windows team has continued to innovate over the past few years. Windows XP, Windows Server 2003, Windows Server 2003 R2, Windows XP SP2, Windows Vista, Windows Server 2008, and Hyper-V are all major accomplishments and great successes—as well as great additions to the Windows family of products.

Frankly, I can't think of a more exciting and challenging topic. Nor can I think of a more authoritative and well-written book. David, Mark, and Alex have done a thorough job of dissecting the Windows architecture and providing diagnostic tools for hands-on learning. I hope you enjoy reading and learning about Windows as much as we all enjoy working on it.

Ben Fathi
Corporate Vice President, Windows Core Development
Microsoft Corporation

Acknowledgments

We dedicate this edition to **Jim Allchin**, our executive sponsor and champion before he retired from Microsoft. Jim supported our book work on this and earlier editions and was instrumental in bringing Mark Russinovich to Microsoft. In addition to shepherding Windows Vista out the door, Jim also oversaw the delivery of Windows 2000, Windows XP, and Windows Server 2003.

Each edition of this book has to acknowledge **Dave Cutler**, Senior Technical Fellow and the original architect of Windows NT. Dave originally approved David Solomon's source code access and has been supportive of his work to explain the internals of Windows through his training business as well as during the writing of the editions of this book.

We also thank three developers at Microsoft for contributing content that was incorporated into this edition:

- **Christian Allred**, who wrote detailed descriptions on transactional NTFS (TxF) internals, data structures, and behaviors
- **Stone Cong**, who wrote content and created diagrams about the Common Log File System (CLFS)
- **Adrian Marinescu**, who updated his heap manager section in the memory management chapter

This book wouldn't contain the depth of technical detail or the level of accuracy it has without the input, and support of key members of the Windows development team. We want to thank the following people, who provided technical review and input to the book:

Dmitry Anipko	Kwan Hyun	Ravi Mumulla	Jon Schwartz
Eugene Bak	Mehmet Iyigun	Adi Oltean	Valerie See
Karlito Bonnevie	Philippe Joubert	Vince Orgovan	Matt Setzer
Jon Cargille	Kwan Hyun Kim	Bernard Ourghanlian	Andrey Shedel
Dean DeWhitt	Kinshuman Kinshumann	Alexey Pakhunov	Neeraj Singh
Apurva Doshi	Alex Kirshenbaum	Milos Petrbock	Vikram Singh
Joseph East	Norbert Kusters	Daniel Pravat	Paul Sliwowicz
Tahsin Erdogan	Jeff Lambert	Ravi Pudipeddi	John Stephens
Cenk Ergan	Paul Leach	Melur Raghuraman	Deepu Thomas
Osman Ertugay	Scott Lee	Ramu Ramanathan	J. R. Tipton
Tom Fout	Mark Lloyd	Vlad Sadovsky	Davis Walker
Nar Ganapathy	Karan Mehra	Dragos Sambotin	Brad Waters
Robin Giese	Derek Moore	Jamie Schwartz	Bruce Worthington

Thanks also to Daniel Pearson (who teaches Windows internals for Dave Solomon) for his review and input.

Others might have contributed by answering questions in the hallway or cafeteria or by providing technical material—if we missed you, please forgive us!

The authors would like to thank Ilfak Guilfanov of Hex-Rays (www.hex-rays.com) for the IDA Pro Advanced and Hex-Rays licenses for Alex Ionescu for his use in speeding his reverse engineering of the Windows kernel. Alex chose not to have Windows source code access (as did Mark Russinovich before he joined Microsoft) to research the information for his work on this book, and these tools greatly facilitated his work. IDA's features turn reverse engineering into a powerful tool for understanding Windows internals. Combined with the Hex-Rays Decompiler, this analysis becomes even faster and more refined, as C code is directly presented instead of assembler, including all the right types.

Thanks also to Matt Ginzton of VMware, who arranged for Alex and David to receive VMware Workstation to use in their research for the book. VMware Workstation was used instead of Microsoft Virtual PC because of its support for 64-bit guests and multiple snapshots with nonpersistent disks. (These features are now supported by Hyper-V, Microsoft's new server virtualization offering, but at the time of writing, this support was not available).

Thanks to Mike Vance of AMD for providing Dave Solomon's AMD64 laptop for use in his book research and live classes.

Finally, we want to thank the team at Microsoft Press who helped turn this book from idea into reality:

- Ben Ryan (acquisitions editor at Microsoft Press) for shepherding another edition of this great book
- Kathleen Atkins (project editor) and Devon Musgrave (developmental editor) for launching and overseeing the project
- Andrea Fox (proofreader), Curtis Philips (project and production manager), and John Pierce (project editor and copyeditor) for laboriously going through all our chapters to tighten up text, find inconsistencies, and keep the manuscript to the high standards of Microsoft Press

Alex Ionescu, Mark Russinovich, and David Solomon
May 2009

Introduction

Windows Internals, Fifth Edition is intended for advanced computer professionals (both developers and system administrators) who want to understand how the core components of the Windows Vista and Windows Server 2008 operating systems work internally. With this knowledge, developers can better comprehend the rationale behind design choices when building applications specific to the Windows platform. Such knowledge can also help developers debug complex problems. System administrators can benefit from this information as well, because understanding how the operating system works “under the covers” facilitates understanding the performance behavior of the system and makes troubleshooting system problems much easier when things go wrong. After reading this book, you should have a better understanding of how Windows works and why it behaves as it does.

Structure of the Book

The first two chapters (“Concepts and Tools” and “System Architecture”) lay the foundation with definitions and explanations of terms and concepts used throughout the rest of the book. The next two chapters—“System Mechanisms” and “Management Mechanisms”—describe key underlying mechanisms in the system. The next eight chapters explain the core components of the operating system: processes, threads, and jobs; security; the I/O system; storage management; memory management; the cache manager; file systems; and networking. The last two chapters cover startup and shutdown process and crash dump analysis.

History of the Book

This is the fifth edition of a book that was originally called *Inside Windows NT* (Microsoft Press, 1992), written by Helen Custer (prior to the initial release of Microsoft Windows NT 3.1). *Inside Windows NT* was the first book ever published about Windows NT and provided key insights into the architecture and design of the system. *Inside Windows NT, Second Edition* (Microsoft Press, 1998) was written by David Solomon. It updated the original book to cover Windows NT 4.0 and had a greatly increased level of technical depth. *Inside Windows 2000, Third Edition* (Microsoft Press, 2000) was authored by David Solomon and Mark Russinovich. It added many new topics, such as startup and shutdown, service internals, registry internals, file system drivers, and networking. It also covered kernel changes in Windows 2000, such as the Windows Driver Model (WDM), Plug and Play, power management, Windows Management Instrumentation (WMI), encryption, the job object, and Terminal Services. *Windows Internals, Fourth Edition* was the Windows XP and Windows Server 2003

update and added more content focused on helping IT professionals make use of their knowledge of Windows internals, such as using key tools from Windows Sysinternals (www.microsoft.com/technet/sysinternals) and analyzing crash dumps.

Fifth Edition Changes

This latest edition has been updated to cover the kernel changes made in Windows Vista and Windows Server 2008. Hands-on experiments have been updated to reflect changes in tools, and newly added experiments use tools not available when the fourth edition was written. Additionally, content has been added to cover mechanisms that were not previously described, such as the image loader and user-mode debugging facility, and information about previously covered subjects has been expanded as well.

Hands-On Experiments

Even without access to the Windows source code, you can glean much about Windows internals from tools such as the kernel debugger and tools from Sysinternals and Winsider Seminars & Solutions (www.winsiderss.com). When a tool can be used to expose or demonstrate some aspect of the internal behavior of Windows, the steps for trying the tool yourself are listed in “Experiment” boxes. These appear throughout the book, and we encourage you to try these as you’re reading—seeing visible proof of how Windows works internally will make much more of an impression on you than just reading about it will.

Topics Not Covered

Windows is a large and complex operating system. This book doesn’t cover everything relevant to Windows internals but instead focuses on the base system components. For example, this book doesn’t describe COM+, the Windows distributed object-oriented programming infrastructure, or the .NET Framework, the foundation of managed code applications.

Because this is an internals book and not a user, programming, or system administration book, it doesn’t describe how to use, program, or configure Windows.

A Warning and a Caveat

Because this book describes undocumented behavior of the internal architecture and operation of the Windows operating system (such as internal kernel structures and functions), this

content is subject to change between releases. (External interfaces, such as the Windows API, are not subject to incompatible changes.)

By “subject to change,” we don’t necessarily mean that details described in this book *will* change between releases, but you can’t count on them not changing. Any software that uses these undocumented interfaces might not work on future releases of Windows. Even worse, software that runs in kernel mode (such as device drivers) and uses these undocumented interfaces might experience a system crash when running on a newer release of Windows.

Find Additional Content Online

As new or updated material becomes available that complements this book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web content is available at <http://technet.microsoft.com/en-us/sysinternals/bb963901> and is updated periodically.

Support

Every effort has been made to ensure the accuracy of this book. Should you run into any problems or issues, please refer to the sources listed below.

From the Authors

This book isn’t perfect. No doubt it contains some inaccuracies, or possibly we’ve omitted some topics we should have covered. If you find anything you think is incorrect, or if you believe we should have included material that isn’t here, please feel free to send e-mail to winint@solsem.com. Updates and corrections will be posted on the Web site <http://technet.microsoft.com/en-us/sysinternals/bb963901.aspx>.

From Microsoft Press

Microsoft Press provides corrections for books through the World Wide Web at the following address:

www.microsoft.com/mspress/support

Questions and Comments

In addition to sending feedback directly to the authors, if you have comments, questions, or ideas regarding the presentation or use of this book, you can send them to Microsoft using either of the following methods:

Postal mail:

*Microsoft Press
Attn: Windows Internals Editor
One Microsoft Way
Redmond, WA 98052-6399*

E-mail:

mspinput@microsoft.com

Please note that product support isn't offered through these mail addresses. For support information, visit Microsoft's Web site at <http://support.microsoft.com/>.

Chapter 5

Processes, Threads, and Jobs

In this chapter, we'll explain the data structures and algorithms that deal with processes, threads, and jobs in the Windows operating system. The first section focuses on the internal structures that make up a process. The second section outlines the steps involved in creating a process (and its initial thread). The internals of threads and thread scheduling are then described. The chapter concludes with a description of the job object.

Where relevant performance counters or kernel variables exist, they are mentioned. Although this book isn't a Windows programming book, the pertinent process, thread, and job Windows functions are listed so that you can pursue additional information on their use.

Because processes and threads touch so many components in Windows, a number of terms and data structures (such as working sets, objects and handles, system memory heaps, and so on) are referred to in this chapter but are explained in detail elsewhere in the book. To fully understand this chapter, you need to be familiar with the terms and concepts explained in Chapters 1 and 2, such as the difference between a process and a thread, the Windows virtual address space layout, and the difference between user mode and kernel mode.

Process Internals

This section describes the key Windows process data structures. Also listed are key kernel variables, performance counters, and functions and tools that relate to processes.

Data Structures

Each Windows process is represented by an executive process (EPROCESS) block. Besides containing many attributes relating to a process, an EPROCESS block contains and points to a number of other related data structures. For example, each process has one or more threads represented by executive thread (ETHREAD) blocks. (Thread data structures are explained in the section "Thread Internals" later in this chapter.) The EPROCESS block and its related data structures exist in system address space, with the exception of the process environment block (PEB), which exists in the process address space (because it contains information that needs to be accessed by user-mode code).

In addition to the EPROCESS block and the PEB, the Windows subsystem process (Csrss) maintains a parallel structure for each process that is executing a Windows program. Finally,

the kernel-mode part of the Windows subsystem (Win32k.sys) will also maintain a per-process data structure that is created the first time a thread calls a Windows USER or GDI function that is implemented in kernel mode.

Figure 5-1 is a simplified diagram of the process and thread data structures. Each data structure shown in the figure is described in detail in this chapter.

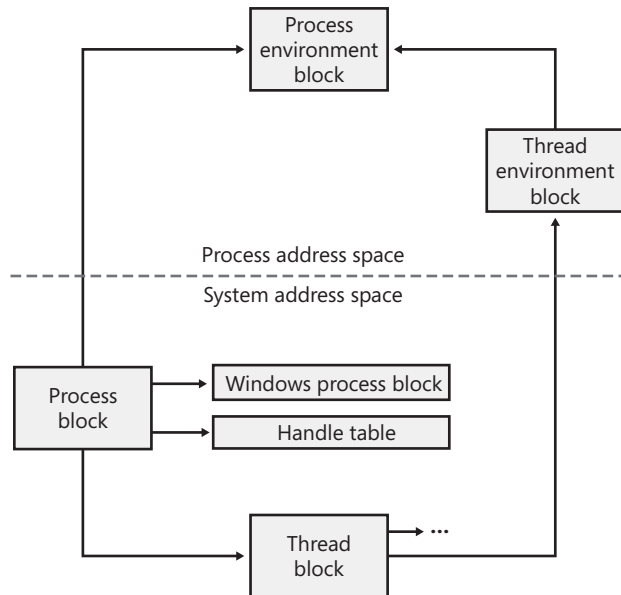


FIGURE 5-1 Data structures associated with processes and threads

First let's focus on the process block. (We'll get to the thread block in the section "Thread Internals" later in the chapter.) Figure 5-2 shows the key fields in an EPROCESS block.

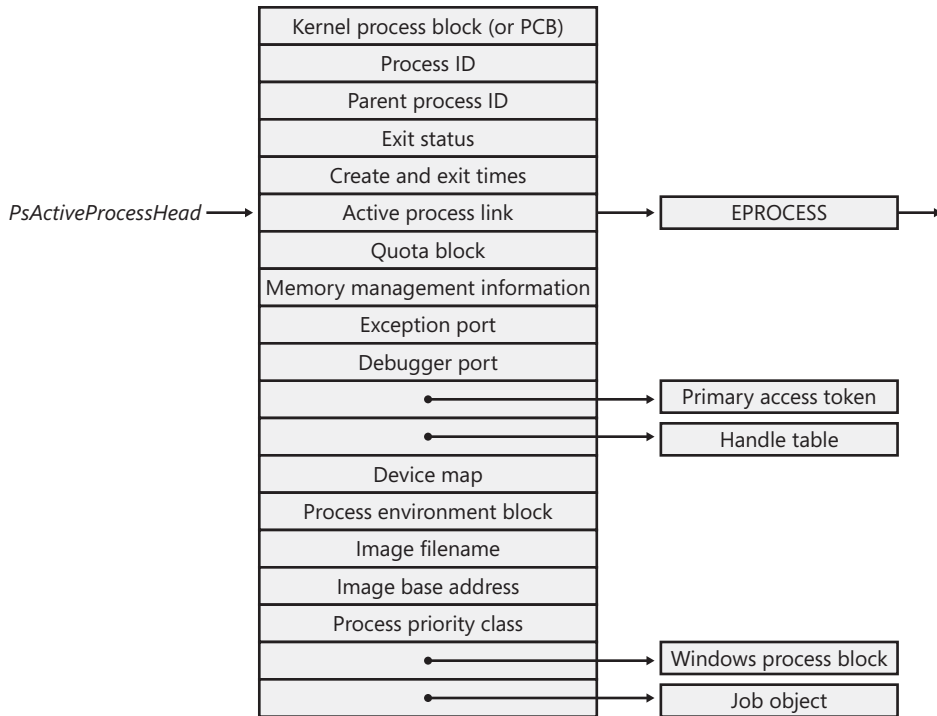


FIGURE 5-2 Structure of an executive process block



EXPERIMENT: Displaying the Format of an EPROCESS Block

For a list of the fields that make up an EPROCESS block and their offsets in hexadecimal, type `dt _eprocess` in the kernel debugger. (See Chapter 1 for more information on the kernel debugger and how to perform kernel debugging on the local system.) The output (truncated for the sake of space) on a 32-bit system looks like this:

```

1kd> dt _eprocess
nt!_EPROCESS
+0x000 Pcb           : _KPROCESS
+0x080 ProcessLock  : _EX_PUSH_LOCK
+0x088 CreateTime   : _LARGE_INTEGER
+0x090 ExitTime     : _LARGE_INTEGER
+0x098 RundownProtect : _EX Rundown_Ref
+0x09c UniqueProcessId : Ptr32 Void
+0x0a0 ActiveProcessLinks : _LIST_ENTRY
+0x0a8 QuotaUsage    : [3] Uint4B
+0x0b4 QuotaPeak     : [3] Uint4B
+0x0c0 CommitCharge : Uint4B
+0x0c4 PeakVirtualSize : Uint4B
+0x0c8 VirtualSize  : Uint4B
+0x0cc SessionProcessLinks : _LIST_ENTRY
+0x0d4 DebugPort     : Ptr32 Void
+0x0d8 ExceptionPortData : Ptr32 Void

```

```

+0x0d8 ExceptionPortValue : Uint4B
+0x0d8 ExceptionPortState : Pos 0, 3 Bits
+0x0dc ObjectTable       : Ptr32 _HANDLE_TABLE
+0x0e0 Token              : _EX_FAST_REF
+0x0e4 WorkingSetPage    : Uint4B
+0x0e8 AddressCreationLock : _EX_PUSH_LOCK
+0x0ec RotateInProgress  : Ptr32 _ETHREAD
+0x0f0 ForkInProgress    : Ptr32 _ETHREAD
+0x0f4 HardwareTrigger   : Uint4B
+0x0f8 PhysicalVadRoot   : Ptr32 _MM_AVL_TABLE
+0x0fc CloneRoot         : Ptr32 Void
+0x100 NumberOfPrivatePages : Uint4B
+0x104 NumberOfLockedPages : Uint4B
+0x108 Win32Process      : Ptr32 Void
+0x10c Job                : Ptr32 _EJOB
+0x110 SectionObject     : Ptr32 Void
+0x114 SectionBaseAddress : Ptr32 Void
+0x118 QuotaBlock        : Ptr32 _EPROCESS_QUOTA_BLOCK

```

Note that the first field (Pcb) is actually a substructure, the kernel process block (KPROCESS), which is where scheduling-related information is stored. To display the format of the kernel process block, type **dt _kprocess**:

```

1kd> dt _kprocess
nt!_KPROCESS
+0x000 Header           : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : Uint4B
+0x01c Unused0         : Uint4B
+0x020 LdtDescriptor   : _KGDTENTRY
+0x028 Int21Descriptor : _KIDTENTRY
+0x030 IopmOffset      : Uint2B
+0x032 Iopl            : UChar
+0x033 Unused         : UChar
+0x034 ActiveProcessors : Uint4B
+0x038 KernelTime     : Uint4B
+0x03c UserTime       : Uint4B
+0x040 ReadyListHead  : _LIST_ENTRY
+0x048 SwapListEntry   : _SINGLE_LIST_ENTRY
+0x04c VdmTrapHandler  : Ptr32 Void
+0x050 ThreadListHead  : _LIST_ENTRY
+0x058 ProcessLock     : Uint4B
+0x05c Affinity        : Uint4B
+0x060 AutoAlignment   : Pos 0, 1 Bit
+0x060 DisableBoost    : Pos 1, 1 Bit
+0x060 DisableQuantum  : Pos 2, 1 Bit
+0x060 ReservedFlags   : Pos 3, 29 Bits
+0x060 ProcessFlags    : Int4B
+0x064 BasePriority     : Char
+0x065 QuantumReset    : Char
+0x066 State           : UChar
+0x067 ThreadSeed      : UChar
+0x068 PowerState      : UChar
+0x069 IdealNode       : UChar

```

```

+0x06a Visited      : UChar
+0x06b Flags       : _KEXECUTE_OPTIONS
+0x06b ExecuteOptions : UChar
+0x06c StackCount  : Uint4B
+0x070 ProcessListEntry : _LIST_ENTRY
+0x078 CycleTime   : Uint8B

```

An alternative way to see the KPROCESS (and other substructures in the EPROCESS) is to use the recursion (*-r*) switch of the *dt* command. For example, typing **dt _eprocess -r1** will recurse and display all substructures one level deep.

The *dt* command shows the format of a process block, not its contents. To show an instance of an actual process, you can specify the address of an EPROCESS structure as an argument to the *dt* command. You can get the address of all the EPROCESS blocks in the system by using the *!process 0 0* command. An annotated example of the output from this command is included later in this chapter.

Table 5-1 explains some of the fields in the preceding experiment in more detail and includes references to other places in the book where you can find more information about them. As we've said before and will no doubt say again, processes and threads are such integral parts of Windows that it's impossible to talk about them without referring to many other parts of the system. To keep the length of this chapter manageable, however, we've covered those related subjects (such as memory management, security, objects, and handles) elsewhere.

TABLE 5-1 Contents of the EPROCESS Block

Element	Purpose	Additional Reference
Kernel process (KPROCESS) block	Common dispatcher object header, pointer to the process page directory, list of kernel thread (KTHREAD) blocks belonging to the process, default base priority, affinity mask, and total kernel and user time and CPU clock cycles for the threads in the process.	Thread scheduling (Chapter 5)
Process identification	Unique process ID, creating process ID, name of image being run, window station process is running on.	
Quota block	Limits on processor usage, nonpaged pool, paged pool, and page file usage plus current and peak process non-paged and paged pool usage. (Note: Several processes can share this structure: all the system processes in session 0 point to a single systemwide quota block; all other processes in interactive sessions share a single quota block.)	

Element	Purpose	Additional Reference
Virtual address descriptors (VADs)	Series of data structures that describes the status of the portions of the address space that exist in the process.	Virtual address descriptors (Chapter 9)
Working set information	Pointer to working set list (MMWSL structure); current, peak, minimum, and maximum working set size; last trim time; page fault count; memory priority; outswap flags; page fault history.	Working sets (Chapter 9)
Virtual memory information	Current and peak virtual size, page file usage, hardware page table entry for process page directory.	Chapter 9
Exception legacy local procedure call (LPC) port	Interprocess communication channel to which the process manager sends a message when one of the process's threads causes an exception.	Exception dispatching (Chapter 3)
Debugging object	Executive object through which the user-mode debugging infrastructure sends notifications when one of the process's threads causes a debug event.	User-mode debugging (Chapter 3)
Access token (TOKEN)	Executive object describing the security profile of this process.	Chapter 6
Handle table	Address of per-process handle table.	Object handles and the process handle table (Chapter 3)
Device map	Address of object directory to resolve device name references in (supports multiple users).	Object names (Chapter 3)
Process environment block (PEB)	Image information (base address, version numbers, module list), process heap information, and thread-local storage utilization. (Note: The pointers to the process heaps start at the first byte after the PEB.)	Chapter 5
Windows subsystem process block (W32PROCESS)	Process details needed by the kernel-mode component of the Windows subsystem.	

The kernel process (KPROCESS) block, which is part of the EPROCESS block, and the process environment block (PEB), which is pointed to by the EPROCESS block, contain additional details about the process object. The KPROCESS block (which is sometimes called the PCB or process control block) is illustrated in Figure 5-3. It contains the basic information that the Windows kernel needs to schedule the threads inside a process. (Page directories are covered in Chapter 9, and kernel thread blocks are described in more detail later in this chapter.)

The PEB, which lives in the user process address space, contains information needed by the image loader, the heap manager, and other Windows system DLLs that need to access it from user mode. (The EPROCESS and KPROCESS blocks are accessible only from kernel mode.) The basic structure of the PEB is illustrated in Figure 5-4 and is explained in more detail later in this chapter.

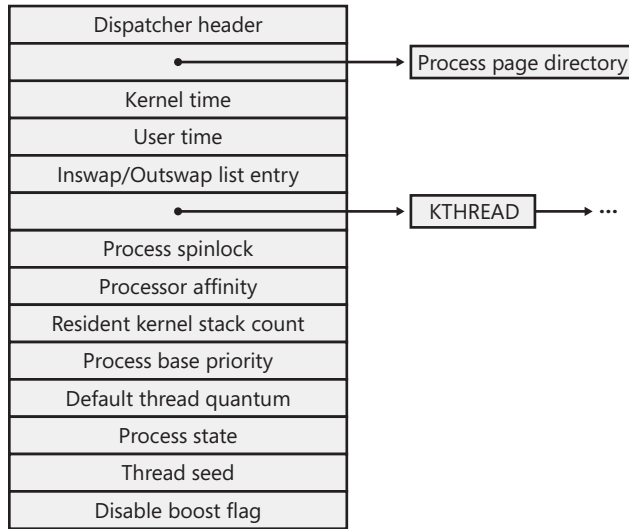


FIGURE 5-3 Structure of the executive process block

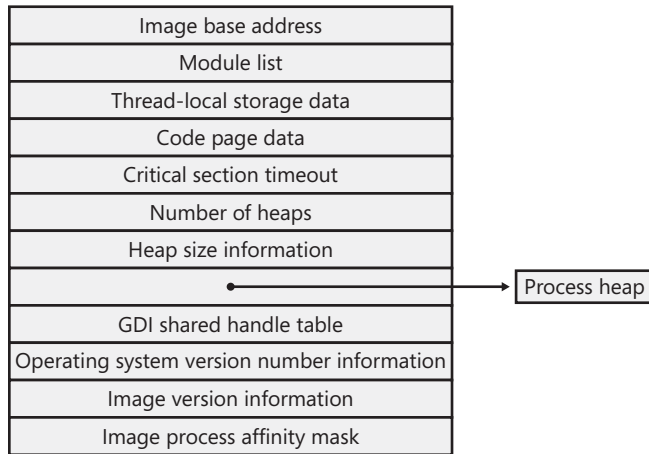


FIGURE 5-4 Fields of the process environment block



EXPERIMENT: Examining the PEB

You can dump the PEB structure with the *!peb* command in the kernel debugger. To get the address of the PEB, use the *!process* command as follows:

```
1kd> !process
PROCESS 8575f030 SessionId: 1 Cid: 08d0 Peb: 7ffd9000 ParentCid: 0360
DirBase: 1a81b000 ObjectTable: e12bd418 HandleCount: 66.
Image: windbg.exe
```

Then specify that address to the *!peb* command as follows:

```
1kd> !peb 7ffd9000
PEB at 7ffd9000
InheritedAddressSpace: No
ReadImageFileExecOptions: No
BeingDebugged: No
ImageBaseAddress: 002a0000
Ldr 77895d00
Ldr.Initialized: Yes
Ldr.InInitializationOrderModuleList: 00151c38 . 00191558
Ldr.InLoadOrderModuleList: 00151bb8 . 00191548
Ldr.InMemoryOrderModuleList: 00151bc0 . 00191550
Base TimeStamp Module
2a0000 4678a41e Jun 19 23:50:54 2007 C:\Program Files\Debugging Tools for
Windows\windbg.exe
777d0000 4549bdc9 Nov 02 05:43:37 2006 C:\Windows\system32\Ntdll.dll
764c0000 4549bd80 Nov 02 05:42:24 2006 C:\Windows\system32\kernel32.dll
SubSystemData: 00000000
ProcessHeap: 00150000
ProcessParameters: 001512e0
WindowTitle: 'C:\Users\Alex Ionescu\Desktop\WinDbg.1nk'
ImageFile: 'C:\Program Files\Debugging Tools for Windows\windbg.exe'
CommandLine: '"C:\Program Files\Debugging Tools for Windows\windbg.exe" '
DllPath: 'C:\Program Files\Debugging Tools for Windows;C:\Windows\
system32;C:\Windows\system;C:\Windows;. ;C:\Windows\system32;C:\Windows;
C:\Windows\System32\wbem;C:\Program Files\Common Files\Roxio Shared\
DLLShared\;C:\Program Files\Common Files\Roxio Shared\DLLShared\;C:\Program
Files\Common Files\Roxio Shared\9.0\DLLShared\;c:\sysint;C:\Program Files\
QuickTime\QTSystem\'
Environment: 001850a8
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\Alex Ionescu\AppData\Roaming
.
.
.
```

Kernel Variables

A few key kernel global variables that relate to processes are listed in Table 5-2. These variables are referred to later in the chapter, when the steps in creating a process are described.

TABLE 5-2 Process-Related Kernel Variables

Element	Purpose	Additional Reference
<i>PsActiveProcessHead</i>	Doubly linked list	List head of process blocks
<i>PsIdleProcess</i>	Pointer to EPROCESS	Idle process block
<i>PsInitialSystemProcess</i>	Pointer to EPROCESS	Pointer to the process block of the initial system process that contains the system threads
<i>PspCreateProcessNotifyRoutine</i>	Array of executive callback objects	Array of callback objects describing the routines to be called on process creation and deletion (maximum of eight)
<i>PspCreateProcessNotifyRoutineCount</i>	32-bit integer	Count of registered process notification routines
<i>PspCreateProcessNotifyRoutineCountEx</i>	32-bit integer	Count of registered extended process notification routines
<i>PspLoadImageNotifyRoutine</i>	Array of executive callback objects	Array of callback objects describing the routines to be called on image load (maximum of eight)
<i>PspLoadImageNotifyRoutineCount</i>	32-bit integer	Count of registered image-load notification routines
<i>PspNotifyEnableMask</i>	32-bit integer	Mask for quickly checking whether any extended or standard notification routines are enabled
<i>PspCidTable</i>	Pointer to HANDLE_TABLE	Handle table for process and thread client IDs

Performance Counters

Windows maintains a number of counters with which you can track the processes running on your system; you can retrieve these counters programmatically or view them with the Performance tool. Table 5-3 lists the performance counters relevant to processes.

TABLE 5-3 Process-Related Performance Counters

Object: Counter	Function
Process: % Privileged Time	Describes the percentage of time that the threads in the process have run in kernel mode during a specified interval.
Process: % Processor Time	Describes the percentage of CPU time that the threads in the process have used during a specified interval. This count is the sum of % Privileged Time and % User Time.

Object: Counter	Function
Process: % User Time	Describes the percentage of time that the threads in the process have run in user mode during a specified interval.
Process: Elapsed Time	Describes the total elapsed time in seconds since this process was created.
Process: ID Process	Returns the process ID. This ID applies only while the process exists because process IDs are reused.
Process: Creating Process ID	Returns the process ID of the creating process. This value isn't updated if the creating process exits.
Process: Thread Count	Returns the number of threads in the process.
Process: Handle Count	Returns the number of handles open in the process.

Relevant Functions

For reference purposes, some of the Windows functions that apply to processes are described in Table 5-4. For further information, consult the Windows API documentation in the MSDN Library.

TABLE 5-4 Process-Related Functions

Function	Description
<i>CreateProcess</i>	Creates a new process and thread using the caller's security identification
<i>CreateProcessAsUser</i>	Creates a new process and thread with the specified alternate security token
<i>CreateProcessWithLogonW</i>	Creates a new process and thread to run under the credentials of the specified username and password
<i>CreateProcessWithTokenW</i>	Creates a new process and thread with the specified alternate security token, with additional options such as allowing the user profile to be loaded
<i>OpenProcess</i>	Returns a handle to the specified process object
<i>ExitProcess</i>	Ends a process, and notifies all attached DLLs
<i>TerminateProcess</i>	Ends a process without notifying the DLLs
<i>FlushInstructionCache</i>	Empties the specified process's instruction cache
<i>FlushProcessWriteBuffers</i>	Empties the specified process's write queue
<i>GetProcessTimes</i>	Obtains a process's timing information, describing how much time the threads inside the process spent in user and kernel mode
<i>QueryProcessCycleTimeCounter</i>	Obtains a process's CPU timing information, describing how many clock cycles the threads inside the process have spent in total
<i>Query/SetProcessAffinityUpdateMode</i>	Defines whether the process's affinity is automatically updated if new processors are added to the running system

Function	Description
<i>Get/SetProcessDEPPolicy</i>	Returns or sets the DEP (Data Execution Protection) policy for the process
<i>GetExitCodeProcess</i>	Returns the exit code for a process, indicating how and why the process shut down
<i>GetCommandLine</i>	Returns a pointer to the command-line string passed to the current process
<i>QueryFullProcessImageName</i>	Returns the full name of the executable image associated with the process
<i>GetCurrentProcess</i>	Returns a pseudo handle for the current process
<i>GetCurrentProcessId</i>	Returns the ID of the current process
<i>GetProcessVersion</i>	Returns the major and minor versions of the Windows version on which the specified process expects to run
<i>GetStartupInfo</i>	Returns the contents of the STARTUPINFO structure specified during <i>CreateProcess</i>
<i>GetEnvironmentStrings</i>	Returns the address of the environment block
<i>Get/SetEnvironmentVariable</i>	Returns or sets a specific environment variable
<i>Get/SetProcessShutdownParameters</i>	Defines the shutdown priority and number of retries for the current process
<i>SetProcessDPIAware</i>	Specifies whether the process is aware of dots per inch (DPI) settings
<i>GetGuiResources</i>	Returns a count of User and GDI handles



EXPERIMENT: Using the Kernel Debugger *!process* Command

The kernel debugger *!process* command displays a subset of the information in an EPROCESS block. This output is arranged in two parts for each process. First you see the information about the process, as shown here (when you don't specify a process address or ID, *!process* lists information for the active process on the current CPU):

```

1kd> !process
PROCESS 85857160 SessionId: 1 Cid: 0bcc Peb: 7ffd9000 ParentCid: 090c
  DirBase: b45b0820 ObjectTable: b94ffda0 HandleCount: 99.
  Image: windbg.exe
  VadRoot 85a1c8e8 Vads 97 Clone 0 Private 5919. Modified 153. Locked 1.
  DeviceMap 9d32ee50
  Token                                ebaa1938
  ElapsedTime                          00:48:44.125
  UserTime                              00:00:00.000
  KernelTime                            00:00:00.000
  QuotaPoolUsage[PagedPool]            166784
  QuotaPoolUsage[NonPagedPool]         4776
  Working Set Sizes (now,min,max)      (8938, 50, 345) (35752KB, 200KB, 1380KB)
  PeakWorkingSetSize                    8938
  VirtualSize                           106 Mb
  PeakVirtualSize                       108 Mb

```

PageFaultCount	37066
MemoryPriority	BACKGROUND
BasePriority	8
CommitCharge	6242

After the basic process output comes a list of the threads in the process. That output is explained in the “Experiment: Using the Kernel Debugger *!thread* Command” section later in the chapter. Other commands that display process information include *!handle*, which dumps the process handle table (which is described in more detail in the section “Object Handles and the Process Handle Table” in Chapter 3). Process and thread security structures are described in Chapter 6.

Protected Processes

In the Windows security model, any process running with a token containing the debug privilege (such as an administrator’s account) can request any access right that it desires to any other process running on the machine—for example, it can read and write arbitrary process memory, inject code, suspend and resume threads, and query information on other processes. Tools like Process Explorer and Task Manager need and request these access rights to provide their functionality to users.

This logical behavior (which helps ensure that administrators will always have full control of the running code on the system) clashes with the system behavior for digital rights management requirements imposed by the media industry on computer operating systems that need to support playback of advanced, high-quality digital content such as BluRay and HD-DVD media. To support reliable and protected playback of such content, Windows uses *protected processes*. These processes exist alongside normal Windows processes, but they add significant constraints to the access rights that other processes on the system (even when running with administrative privileges) can request.

Protected processes can be created by any application; however, the operating system will only allow a process to be protected if the image file has been digitally signed with a special Windows Media Certificate. The Protected Media Path (PMP) in Windows Vista makes use of protected processes to provide protection for high-value media, and developers of applications such as DVD players can make use of protected processes by using the Media Foundation API.

The Audio Device Graph process (Audiodg.exe) is a protected process, since protected music content may be decoded through it. Similarly, the Windows Error Reporting (WER;

see Chapter 3 for more information) client process (Werfault.exe) can also run protected because it needs to have access to protected processes in case one of them crashes. Finally, the System process itself is protected because some of the decryption information is generated by the Ksecdd.sys driver and stored in its user-mode memory. The System process is also protected to protect the integrity of all kernel handles (since the System process's handle table contains all the kernel handles on the system).

At the kernel level, support for protected processes is twofold: first, the bulk of process creation occurs in kernel mode to avoid injection attacks. (The flow for both protected and standard process creation is described in detail in the next section.) Second, protected processes have a special bit set in their EPROCESS structure that modifies the behavior of security-related routines in the process manager to deny certain access rights that would normally be granted to administrators. Table 5-5 indicates access rights that are limited or denied.

TABLE 5-5 Process Access Rights Denied for Protected Processes

Object: Access Mask	Function
Standard: READ_CONTROL	Prevents the protected process's access control list (ACL) from being read.
Standard: WRITE_DAC, WRITE_OWNER	Prevents access to the protected process's access control list or modifying its owner (which would grant the former).
Process: PROCESS_ALL_ACCESS	Prevents full access to the protected process.
Process: PROCESS_CREATE_PROCESS	Prevents creation of a child process of a protected process.
Process: PROCESS_CREATE_THREAD	Prevents creation of a thread inside a protected process.
Process: PROCESS_DUP_HANDLE	Prevents duplication of a handle owned by the protected process.
Process: PROCESS_QUERY_INFORMATION	Prevents querying all information on a protected process. However, a new access right was added, PROCESS_QUERY_LIMITED_INFORMATION, that grants limited access to information on the process.
Process: PROCESS_SET_QUOTA	Prevents setting memory or processor-usage limits on a protected process.
Process: PROCESS_SET_INFORMATION	Prevents modification of process settings for a protected process.
Process: PROCESS_VM_OPERATION, PROCESS_VM_READ, PROCESS_VM_WRITE	Prevents accessing the memory of a protected process.

Certain access rights are also disabled for threads running inside protected processes; we will look at those access rights later in this chapter in the section "Thread Internals."

Because Process Explorer uses standard user-mode Windows APIs to query information on process internals, it is unable to perform certain operations on such processes. On the other

hand, a tool like WinDbg in kernel debugging mode, which uses kernel-mode infrastructure to obtain this information, will be able to display complete information. See the experiment in the thread internals section on how Process Explorer behaves when confronted with a protected process such as Audiodg.exe.



Note As mentioned in Chapter 1, to perform local kernel debugging you must boot in debugging mode (enabled by using “bcdedit /debug on” or by using the Msconfig advanced boot options). This protects against debugger-based attacks on protected processes and the Protected Media Path (PMP). When booted in debugging mode, high-definition content playback will not work; for example, attempting to play MPEG2 media such as a DVD will result in an access violation inside the media player (this is by design).

Limiting these access rights reliably allows the kernel to sandbox a protected process from user-mode access. On the other hand, because a protected process is indicated by a flag in the EPROCESS block, an administrator can still load a kernel-mode driver that disables this bit. However, this would be a violation of the PMP model and considered malicious, and such a driver would likely eventually be blocked from loading on a 64-bit system because the kernel-mode code-signing policy prohibits the digital signing of malicious code. Even on 32-bit systems, the driver has to be recognized by PMP policy or else the playback will be halted. This policy is implemented by Microsoft and not by any kernel detection. This block would require manual action from Microsoft to identify the signature as malicious and update the kernel.

Flow of *CreateProcess*

So far in this chapter, you’ve seen the structures that are part of a process and the API functions with which you (and the operating system) can manipulate processes. You’ve also found out how you can use tools to view how processes interact with your system. But how did those processes come into being, and how do they exit once they’ve fulfilled their purpose? In the following sections, you’ll discover how a Windows process comes to life.

A Windows subsystem process is created when an application calls one of the process creation functions, such as *CreateProcess*, *CreateProcessAsUser*, *CreateProcessWithTokenW*, or *CreateProcessWithLogonW*. Creating a Windows process consists of several stages carried out in three parts of the operating system: the Windows client-side library Kernel32.dll (in the case of the *CreateProcessAsUser*, *CreateProcessWithTokenW*, and *CreateProcessWithLogonW* routines, part of the work is first done in Advapi32.dll), the Windows executive, and the Windows subsystem process (Csrss).

Because of the multiple environment subsystem architecture of Windows, creating an executive process object (which other subsystems can use) is separated from the work involved in creating a Windows subsystem process. So, although the following description of the flow of the Windows *CreateProcess* function is complicated, keep in mind that part of the work is specific to the semantics added by the Windows subsystem as opposed to the core work needed to create an executive process object.

The following list summarizes the main stages of creating a process with the Windows *CreateProcess* function. The operations performed in each stage are described in detail in the subsequent sections. Some of these operations may be performed by *CreateProcess* itself (or other helper routines in user mode), while others will be performed by *NtCreateUserProcess* or one of its helper routines in kernel mode. In our detailed analysis to follow, we will differentiate between the two at each step required.



Note Many steps of *CreateProcess* are related to the setup of the process virtual address space and therefore refer to many memory management terms and structures that are defined in Chapter 9.

1. Validate parameters; convert Windows subsystem flags and options to their native counterparts; parse, validate, and convert the attribute list to its native counterpart.
2. Open the image file (.exe) to be executed inside the process.
3. Create the Windows executive process object.
4. Create the initial thread (stack, context, and Windows executive thread object).
5. Perform post-creation, Windows-subsystem-specific process initialization.
6. Start execution of the initial thread (unless the `CREATE_ SUSPENDED` flag was specified).
7. In the context of the new process and thread, complete the initialization of the address space (such as load required DLLs) and begin execution of the program.

Figure 5-5 shows an overview of the stages Windows follows to create a process.

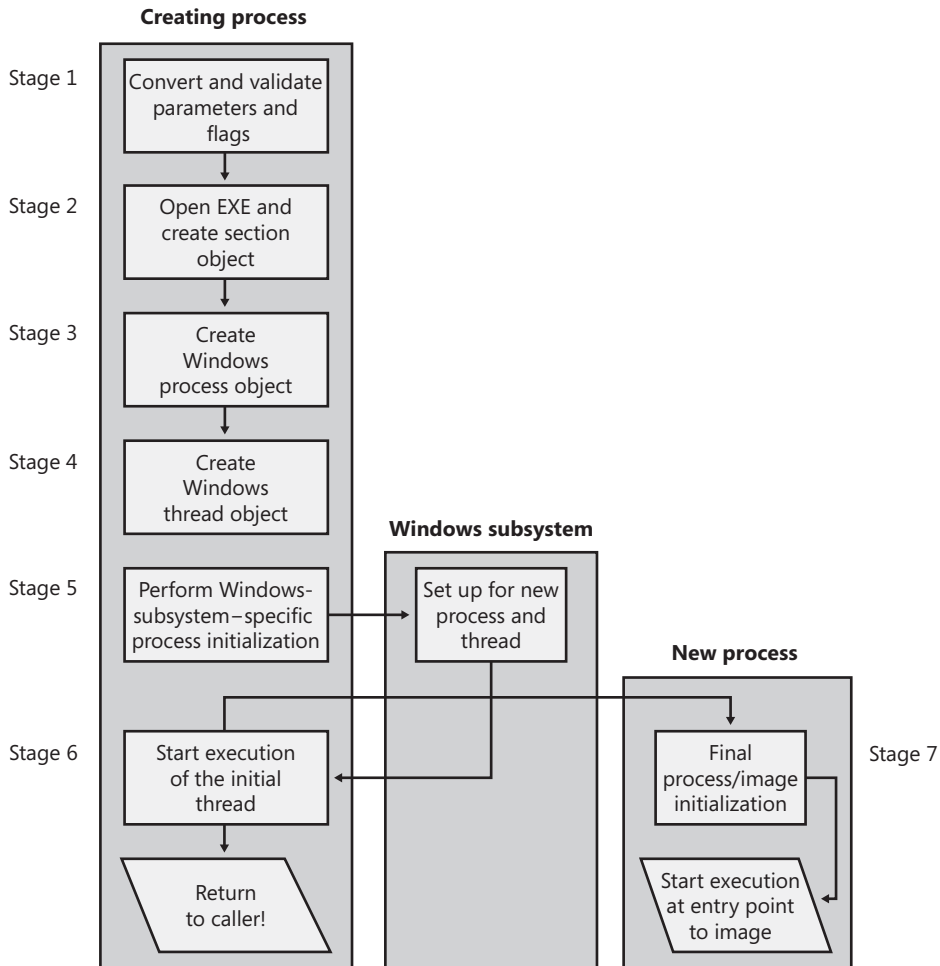


FIGURE 5-5 The main stages of process creation

Stage 1: Converting and Validating Parameters and Flags

Before opening the executable image to run, *CreateProcess* performs the following steps:

- In *CreateProcess*, the priority class for the new process is specified as independent bits in the *CreationFlags* parameter. Thus, you can specify more than one priority class for a single *CreateProcess* call. Windows resolves the question of which priority class to assign to the process by choosing the lowest-priority class set.
- If no priority class is specified for the new process, the priority class defaults to Normal unless the priority class of the process that created it is Idle or Below Normal, in which case the priority class of the new process will have the same priority as the creating class.

- If a Real-time priority class is specified for the new process and the process's caller doesn't have the Increase Scheduling Priority privilege, the High priority class is used instead. In other words, *CreateProcess* doesn't fail just because the caller has insufficient privileges to create the process in the Real-time priority class; the new process just won't have as high a priority as Real-time.
- All windows are associated with desktops, the graphical representation of a workspace. If no desktop is specified in *CreateProcess*, the process is associated with the caller's current desktop.
- If the process is part of a job object, but the creation flags requested a separate virtual DOS machine (VDM), the flag is ignored.
- If the caller is sending a handle to a monitor as an output handle instead of a console handle, standard handle flags are ignored.
- If the creation flags specify that the process will be debugged, Kernel32 initiates a connection to the native debugging code in Ntdll.dll by calling *DbgUiConnectToDbg* and gets a handle to the debug object from the thread environment block (TEB) once the function returns.
- Kernel32.dll sets the default hard error mode if the creation flags specified one.
- The user-specified attribute list is converted from Windows subsystem format to native format, and internal attributes are added to it.



Note The attribute list passed on a *CreateProcess* call permits passing back to the caller information beyond a simple status code, such as the TEB address of the initial thread or information on the image section. This is necessary for protected processes since the parent cannot query this information after the child is created.

Once these steps are completed, *CreateProcess* will perform the initial call to *NtCreateUserProcess* to attempt creation of the process. Because Kernel32.dll has no idea at this point whether the application image name is a real Windows application, or if it might be a POSIX, 16-bit, or DOS application, the call may fail, at which point *CreateProcess* will look at the error reason and attempt to correct the situation.

Stage 2: Opening the Image to Be Executed

As illustrated in Figure 5-6, the first stage in *NtCreateUserProcess* is to find the appropriate Windows image that will run the executable file specified by the caller and to create a section object to later map it into the address space of the new process. If the call failed for any reason, it will return to *CreateProcess* with a failure state (see Table 5-6) that will cause *CreateProcess* to attempt execution again.

If the executable file specified is a Windows .exe, *NtCreateUserProcess* will try to open the file and create a section object for it. The object isn't mapped into memory yet, but it is opened.

Just because a section object has been successfully created doesn't mean that the file is a valid Windows image, however; it could be a DLL or a POSIX executable. If the file is a POSIX executable, the image to be run changes to *Posix.exe*, and *CreateProcess* restarts from the beginning of Stage 1. If the file is a DLL, *CreateProcess* fails.

Now that *NtCreateUserProcess* has found a valid Windows executable image, as part of the process creation code described in Stage 3 it looks in the registry under `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options` to see whether a sub-key with the file name and extension of the executable image (but without the directory and path information—for example, *Image.exe*) exists there. If it does, *PspAllocateProcess* looks for a value named *Debugger* for that key. If this value is present, the image to be run becomes the string in that value and *CreateProcess* restarts at Stage 1.



Tip You can take advantage of this process creation behavior and debug the startup code of Windows services processes before they start rather than attach the debugger after starting a service, which doesn't allow you to debug the startup code.

On the other hand, if the image is not a Windows .exe (for example, if it's an MS-DOS, Win16, or a POSIX application), *CreateProcess* goes through a series of steps to find a Windows *support image* to run it. This process is necessary because non-Windows applications aren't run directly—Windows instead uses one of a few special support images that in turn are responsible for actually running the non-Windows program. For example, if you attempt to run a POSIX application, *CreateProcess* identifies it as such and changes the image to be run to the Windows executable file *Posix.exe*. If you attempt to run an MS-DOS or a Win16 executable, the image to be run becomes the Windows executable *Ntvdm.exe*. In short, you can't directly create a process that is *not* a Windows process. If Windows can't find a way to resolve the activated image as a Windows process (as shown in Table 5-6), *CreateProcess* fails.

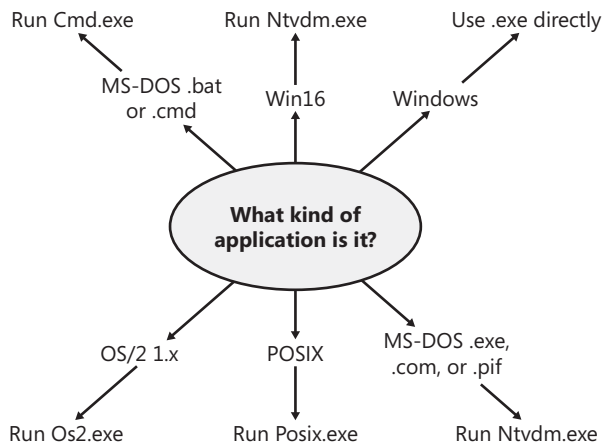


FIGURE 5-6 Choosing a Windows image to activate

TABLE 5-6 Decision Tree for Stage 1 of *CreateProcess*

If the Image . . .	Create State Code	This Image Will Run and This Will Happen
Is a POSIX executable file	<i>PsCreateSuccess</i>	Posix.exe	<i>CreateProcess</i> restarts Stage 1.
Is an MS-DOS application with an .exe, a .com, or a .pif extension	<i>PsCreateFailOnSectionCreate</i>	Ntdvm.exe	<i>CreateProcess</i> restarts Stage 1.
Is a Win16 application	<i>PsCreateFailOnSectionCreate</i>	Ntdvm.exe	<i>CreateProcess</i> restarts Stage 1.
Is a Win64 application on a 32-bit system (or a PPC, MIPS, or Alpha Binary)	<i>PsCreateFailMachineMismatch</i>	N/A	<i>CreateProcess</i> will fail.
Has a Debugger key with another image name	<i>PsCreateFailExeName</i>	Name specified in the Debugger key	<i>CreateProcess</i> restarts Stage 1.
Is an invalid or damaged Windows EXE	<i>PsCreateFailExeFormat</i>	N/A	<i>CreateProcess</i> will fail.
Cannot be opened	<i>PsCreateFailOnFileOpen</i>	N/A	<i>CreateProcess</i> will fail.
Is a command procedure (application with a .bat or a .cmd extension)	<i>PsCreateFailOnSectionCreate</i>	Cmd.exe	<i>CreateProcess</i> restarts Stage 1.

Specifically, the decision tree that *CreateProcess* goes through to run an image is as follows:

- If the image is an MS-DOS application with an .exe, a .com, or a .pif extension, a message is sent to the Windows subsystem to check whether an MS-DOS support process (Ntdvm.exe, specified in the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\cmdline) has already been created for this session. If a support process has been created, it is used to run the MS-DOS application. (The Windows subsystem sends the message to the VDM [Virtual DOS Machine] process to run the new image.) Then *CreateProcess* returns. If a support process hasn't been created, the image to be run changes to Ntdvm.exe and *CreateProcess* restarts at Stage 1.
- If the file to run has a .bat or a .cmd extension, the image to be run becomes Cmd.exe, the Windows command prompt, and *CreateProcess* restarts at Stage 1. (The name of the batch file is passed as the first parameter to Cmd.exe.)
- If the image is a Win16 (Windows 3.1) executable, *CreateProcess* must decide whether a new VDM process must be created to run it or whether it should use the default sessionwide shared VDM process (which might not yet have been created). The *CreateProcess* flags CREATE_SEPARATE_WOW_VDM and CREATE_SHARED_WOW_VDM control this decision. If these flags aren't specified, the registry value HKLM\SYSTEM\CurrentControlSet\Control\WOW\DefaultSeparateVDM dictates the default behavior.

If the application is to be run in a separate VDM, the image to be run changes to the value of `HKLM\SYSTEM\CurrentControlSet\Control\WOW\wowcmdline` and *CreateProcess* restarts at Stage 1. Otherwise, the Windows subsystem sends a message to see whether the shared VDM process exists and can be used. (If the VDM process is running on a different desktop or isn't running under the same security as the caller, it can't be used and a new VDM process must be created.) If a shared VDM process can be used, the Windows subsystem sends a message to it to run the new image and *CreateProcess* returns. If the VDM process hasn't yet been created (or if it exists but can't be used), the image to be run changes to the VDM support image and *CreateProcess* restarts at Stage 1.

Stage 3: Creating the Windows Executive Process Object (*PspAllocateProcess*)

At this point, *NtCreateUserProcess* has opened a valid Windows executable file and created a section object to map it into the new process address space. Next it creates a Windows executive process object to run the image by calling the internal system function *PspAllocateProcess*. Creating the executive process object (which is done by the creating thread) involves the following substages:

- Setting up the EPROCESS block
- Creating the initial process address space
- Initializing the kernel process block (KPROCESS)
- Setting up the PEB
- Concluding the setup of the process address space (which includes initializing the working set list and virtual address space descriptors and mapping the image into address space)



Note The only time there won't be a parent process is during system initialization. After that point, a parent process is always required to provide a security context for the new process.

Stage 3A: Setting Up the EPROCESS Block

This substage involves the following steps:

1. Allocate and initialize the Windows EPROCESS block.
2. Inherit the Windows device namespace (including the definition of drive letters, COM ports, and so on).

3. Inherit the process affinity mask and page priority from the parent process. If there is no parent process, the default page priority (5) is used, and an affinity mask of all processors (*KeActiveProcessors*) is used.
4. Set the new process's quota block to the address of its parent process's quota block, and increment the reference count for the parent's quota block. If the process was created through *CreateProcessAsUser*, this step won't occur.
5. The process minimum and maximum working set size are set to the values of *PspMinimumWorkingSet* and *PspMaximumWorkingSet*, respectively. These values can be overridden if performance options were specified in the *PerfOptions* key part of Image File Execution Options, in which case the maximum working set is taken from there.
6. Store the parent process's process ID in the *InheritedFromUniqueProcessId* field in the new process object.
7. Attach the process to the session of the parent process.
8. Initialize the *KPROCESS* part of the process object. (See Stage 3C.)
9. Create the process's primary access token (a duplicate of its parent's primary token). New processes inherit the security profile of their parents. If the *CreateProcessAsUser* function is being used to specify a different access token for the new process, the token is then changed appropriately.
10. The process handle table is initialized. If the *inherit handles* flag is set for the parent process, any inheritable handles are copied from the parent's object handle table into the new process. (For more information about object handle tables, see Chapter 3.) A process attribute can also be used to specify only a subset of handles, which is useful when you are using *CreateProcessAsUser* to restrict which objects should be inherited by the child process.
11. If performance options were specified through the *PerfOptions* key, these are now applied. The *PerfOptions* key includes overrides for the working set limit, I/O priority, page priority, and CPU priority class of the process.
12. The process priority class and quantum are computed and set.
13. Set the new process's exit status to *STATUS_PENDING*.

Stage 3B: Creating the Initial Process Address Space

The initial process address space consists of the following pages:

- Page directory (and it's possible there'll be more than one for systems with page tables more than two levels, such as x86 systems in PAE mode or 64-bit systems)
- Hyperspace page
- Working set list

To create these three pages, the following steps are taken:

1. Page table entries are created in the appropriate page tables to map the initial pages.
2. The number of pages is deducted from the kernel variable *MmTotalCommittedPages* and added to *MmProcessCommit*.
3. The systemwide default process minimum working set size (*PsMinimumWorkingSet*) is deducted from *MmResidentAvailablePages*.
4. The page table pages for the nonpaged portion of system space and the system cache are mapped into the process.

Stage 3C: Creating the Kernel Process Block

The next stage of *PspAllocateProcess* is the initialization of the KPROCESS block. This work is performed by *KeInitializeProcess*, which contains:

- A pointer to a list of kernel threads. (The kernel has no knowledge of handles, so it bypasses the object table.)
- A pointer to the process's page table directory (which is used to keep track of the process's virtual address space).
- The total time the process's threads have executed.
- The number of clock cycles the process's threads have consumed.
- The process's default base-scheduling priority (which starts as Normal, or 8, unless the parent process was set to Idle or Below Normal, in which case the setting is inherited).
- The default processor affinity for the threads in the process.
- The process swapping state (resident, out-swapped, or in transition).
- The NUMA ideal node (initially set to 0).
- The thread seed, based on the ideal processor that the kernel has chosen for this process (which is based on the previously created process's ideal processor, effectively randomizing this in a round-robin manner). Creating a new process will update the seed in *KeNodeBlock* (the initial NUMA node block) so that the next new process will get a different ideal processor seed.
- The initial value (or reset value) of the process default quantum (which is described in more detail in the "Thread Scheduling" section later in the chapter), which is hardcoded to 6 until it is initialized later (by *PspComputeQuantumAndPriority*).



Note The default initial quantum differs between Windows client and server systems. For more information on thread quanta, turn to their discussion in the section "Thread Scheduling."

Stage 3D: Concluding the Setup of the Process Address Space

Setting up the address space for a new process is somewhat complicated, so let's look at what's involved one step at a time. To get the most out of this section, you should have some familiarity with the internals of the Windows memory manager, which are described in Chapter 9.

- The virtual memory manager sets the value of the process's last trim time to the current time. The working set manager (which runs in the context of the balance set manager system thread) uses this value to determine when to initiate working set trimming.
- The memory manager initializes the process's working set list—page faults can now be taken.
- The section (created when the image file was opened) is now mapped into the new process's address space, and the process section base address is set to the base address of the image.
- Ntdll.dll is mapped into the process.



Note POSIX processes clone the address space of their parents, so they don't have to go through these steps to create a new address space. In the case of POSIX applications, the new process's section base address is set to that of its parent process and the parent's PEB is cloned for the new process.

Stage 3E: Setting Up the PEB

NtCreateUserProcess calls *MmCreatePeb*, which first maps the systemwide national language support (NLS) tables into the process's address space. It next calls *MiCreatePebOrTeb* to allocate a page for the PEB and then initializes a number of fields, which are described in Table 5-7.

TABLE 5-7 Initial Values of the Fields of the PEB

Field	Initial Value
<i>ImageBaseAddress</i>	Base address of section
<i>NumberOfProcessors</i>	<i>KeNumberProcessors</i> kernel variable
<i>NtGlobalFlag</i>	<i>NtGlobalFlag</i> kernel variable
<i>CriticalSectionTimeout</i>	<i>MmCriticalSectionTimeout</i> kernel variable
<i>HeapSegmentReserve</i>	<i>MmHeapSegmentReserve</i> kernel variable
<i>HeapSegmentCommit</i>	<i>MmHeapSegmentCommit</i> kernel variable
<i>HeapDeCommitTotalFreeThreshold</i>	<i>MmHeapDeCommitTotalFreeThreshold</i> kernel variable
<i>HeapDeCommitFreeBlockThreshold</i>	<i>MmHeapDeCommitFreeBlockThreshold</i> kernel variable
<i>NumberOfHeaps</i>	0

Field	Initial Value
<i>MaximumNumberOfHeaps</i>	(Size of a page – size of a PEB) / 4
<i>ProcessHeaps</i>	First byte after PEB
<i>MinimumStackCommit</i>	<i>MmMinimumStackCommitInBytes</i> kernel variable
<i>ImageProcessAffinityMask</i>	<i>KeActiveProcessors</i> or 1 << <i>MmRotatingUniprocessorNumber</i> kernel variable (for uniprocessor-only images)
<i>SessionId</i>	Result of <i>MmGetSessionId</i>
<i>ImageSubsystem</i>	OptionalHeader.Subsystem
<i>ImageSubsystemMajorVersion</i>	OptionalHeader.MajorSubsystemVersion
<i>ImageSubsystemMinorVersion</i>	OptionalHeader.MinorSubsystemVersion
<i>OSMajorVersion</i>	<i>NtMajorVersion</i> kernel variable
<i>OSMinorVersion</i>	<i>NtMinorVersion</i> kernel variable
<i>OSBuildNumber</i>	<i>NtBuildNumber</i> kernel variable & 0x3FFF, combined with <i>CmNtCSDVersion</i> for service packs
<i>OSPlatformId</i>	2

However, if the image file specifies explicit Windows version or affinity values, this information replaces the initial values shown in Table 5-7. The mapping from image information fields to PEB fields is described in Table 5-8.

TABLE 5-8 Windows Replacements for Initial PEB Values

Field Name	Value Taken from Image Header
<i>OSMajorVersion</i>	OptionalHeader.Win32VersionValue & 0xFF
<i>OSMinorVersion</i>	(OptionalHeader.Win32VersionValue >> 8) & 0xFF
<i>OSBuildNumber</i>	(OptionalHeader.Win32VersionValue >> 16) & 0x3FFF, combined with ImageLoadConfigDirectory.CSDVersion
<i>OSPlatformId</i>	(OptionalHeader.Win32VersionValue >> 30) ^ 0x2
<i>ImageProcessAffinityMask</i>	ImageLoadConfigDirectory.ProcessAffinityMask

If the image header characteristics `IMAGE_FILE_UP_SYSTEM_ONLY` flag is set (indicating that the image can run only on a uniprocessor system), a single CPU is chosen for all the threads in this new process to run on. The selection process is performed by simply cycling through the available processors—each time this type of image is run, the next processor is used. In this way, these types of images are spread evenly across the processors.

If the image specifies an explicit processor affinity mask (for example, a field in the configuration header), this value is copied to the PEB and later set as the default process affinity mask.

Stage 3F: Completing the Setup of the Executive Process Object (*PspInsertProcess*)

Before the handle to the new process can be returned, a few final setup steps must be completed, which are performed by *PspInsertProcess* and its helper functions:

1. If systemwide auditing of processes is enabled (either as a result of local policy settings or group policy settings from a domain controller), the process's creation is written to the Security event log.
2. If the parent process was contained in a job, the job is recovered from the job level set of the parent and then bound to the session of the newly created process. Finally, the new process is added to the job.
3. *PspInsertProcess* inserts the new process block at the end of the Windows list of active processes (*PsActiveProcessHead*).
4. The process debug port of the parent process is copied to the new child process, unless the *NoDebugInherit* flag is set (which can be requested when creating the process). If a debug port was specified, it is attached to the new process at this time.
5. Finally, *PspInsertProcess* notifies any registered callback routines, creates a handle for the new process by calling *ObOpenObjectByPointer*, and then returns this handle to the caller.

Stage 4: Creating the Initial Thread and Its Stack and Context

At this point, the Windows executive process object is completely set up. It still has no thread, however, so it can't do anything yet. It's now time to start that work. Normally, the *PspCreateThread* routine is responsible for all aspects of thread creation and is called by *NtCreateThread* when a new thread is being created. However, because the initial thread is created internally by the kernel without user-mode input, the two helper routines that *PspCreateThread* relies on are used instead: *PspAllocateThread* and *PspInsertThread*.

PspAllocateThread handles the actual creation and initialization of the executive thread object itself, while *PspInsertThread* handles the creation of the thread handle and security attributes and the call to *KeStartThread* to turn the executive object into a schedulable thread on the system. However, the thread won't do anything yet—it is created in a suspended state and isn't resumed until the process is completely initialized (as described in Stage 5).



Note The thread parameter (which can't be specified in *CreateProcess* but can be specified in *CreateThread*) is the address of the PEB. This parameter will be used by the initialization code that runs in the context of this new thread (as described in Stage 6).

PspAllocateThread performs the following steps:

1. An executive thread block (ETHREAD) is created and initialized.
2. Before the thread can execute, it needs a stack and a context in which to run, so these are set up. The stack size for the initial thread is taken from the image—there's no way to specify another size.
3. The thread environment block (TEB) is allocated for the new thread.
4. The user-mode thread start address is stored in the ETHREAD. This is the system-supplied thread startup function in Ntdll.dll (*RtlUserThreadStart*). The user's specified Windows start address is stored in the ETHREAD block in a different location so that debugging tools such as Process Explorer can query the information.
5. *KelnitThread* is called to set up the KTHREAD block. The thread's initial and current base priorities are set to the process's base priority, and its affinity and quantum are set to that of the process. This function also sets the initial thread ideal processor. (See the section "Ideal and Last Processor" for a description of how this is chosen.) *KelnitThread* next allocates a kernel stack for the thread and initializes the machine-dependent hardware context for the thread, including the context, trap, and exception frames. The thread's context is set up so that the thread will start in kernel mode in *KiThreadStartup*. Finally, *KelnitThread* sets the thread's state to Initialized and returns to *PspAllocateThread*.

Once that work is finished, *NtCreateUserProcess* will call *PspInsertThread* to perform the following steps:

1. A thread ID is generated for the new thread.
2. The thread count in the process object is incremented, and the thread is added into the process thread list.
3. The thread is put into a suspended state.
4. The object is inserted and any registered thread callbacks are called.
5. The handle is created with *ObOpenObjectByName*.
6. The thread is readied for execution by calling *KeStartThread*.

Stage 5: Performing Windows Subsystem–Specific Post-Initialization

Once *NtCreateUserProcess* returns with a success code, all the necessary executive process and thread objects have been created. Kernel32.dll will now perform various operations related to Windows subsystem–specific operations to finish initializing the process.

First of all, various checks are made for whether Windows should allow the executable to run. These checks includes validating the image version in the header and checking whether Windows application certification has blocked the process (through a group policy). On specialized editions of Windows Server 2008, such as Windows Web Server 2008 and Windows HPC Server 2008, additional checks are made to see if the application imports any disallowed APIs.

If software restriction policies dictate, a restricted token is created for the new process. Afterward, the application compatibility database is queried to see if an entry exists in either the registry or system application database for the process. Compatibility shims will not be applied at this point—the information will be stored in the PEB once the initial thread starts executing (Stage 6).

At this point, Kernel32.dll sends a message to the Windows subsystem so that it can set up SxS information (see the end of this section for more information on side-by-side assemblies) such as manifest files, DLL redirection paths, and out-of-process execution for the new process. It also initializes the Windows subsystem structures for the process and initial thread. The message includes the following information:

- Process and thread handles
- Entries in the creation flags
- ID of the process's creator
- Flag indicating whether the process belongs to a Windows application (so that Csrss can determine whether or not to show the startup cursor)
- UI language Information
- DLL redirection and .local flags
- Manifest file information

The Windows subsystem performs the following steps when it receives this message:

1. *CsrCreateProcess* duplicates a handle for the process and thread. In this step, the usage count of the process and the thread is incremented from 1 (which was set at creation time) to 2.
2. If a process priority class isn't specified, *CsrCreateProcess* sets it according to the algorithm described earlier in this section.
3. The Csrss process block is allocated.
4. The new process's exception port is set to be the general function port for the Windows subsystem so that the Windows subsystem will receive a message when a second chance exception occurs in the process. (For further information on exception handling, see Chapter 3.)

5. The `Csrss` thread block is allocated and initialized.
6. `CsrCreateThread` inserts the thread in the list of threads for the process.
7. The count of processes in this session is incremented.
8. The process shutdown level is set to `0x280` (the default process shutdown level—see *SetProcessShutdownParameters* in the MSDN Library documentation for more information).
9. The new process block is inserted into the list of Windows subsystem-wide processes.
10. The per-process data structure used by the kernel-mode part of the Windows subsystem (`W32PROCESS` structure) is allocated and initialized.
11. The application start cursor is displayed. This cursor is the familiar rolling doughnut shape—the way that Windows says to the user, “I’m starting something, but you can use the cursor in the meantime.” If the process doesn’t make a GUI call after 2 seconds, the cursor reverts to the standard pointer. If the process does make a GUI call in the allotted time, `CsrCreateProcess` waits 5 seconds for the application to show a window. After that time, `CsrCreateProcess` will reset the cursor again.

After `Csrss` has performed these steps, `CreateProcess` checks whether the process was run elevated (which means it was executed through *ShellExecute* and elevated by the `AppInfo` service after the consent dialog box was shown to the user). This includes checking whether the process was a setup program. If it was, the process’s token is opened, and the virtualization flag is turned on so that the application is virtualized. (See the information on UAC and virtualization in Chapter 6.) If the application contained elevation shims or had a requested elevation level in its manifest, the process is destroyed and an elevation request is sent to the `AppInfo` service. (See Chapter 6 for more information on elevation.)

Note that most of these checks are not performed for protected processes; because these processes must have been designed for Windows Vista or later, there’s no reason why they should require elevation, virtualization, or application compatibility checks and processing. Additionally, allowing mechanisms such as the shim engine to use its usual hooking and memory patching techniques on a protected process would result in a security hole if someone could figure how to insert arbitrary shims that modify the behavior of the protected process.

Stage 6: Starting Execution of the Initial Thread

At this point, the process environment has been determined, resources for its threads to use have been allocated, the process has a thread, and the Windows subsystem knows about the new process. Unless the caller specified the `CREATE_SUSPENDED` flag, the initial thread is now resumed so that it can start running and perform the remainder of the process initialization work that occurs in the context of the new process (Stage 7).

Stage 7: Performing Process Initialization in the Context of the New Process

The new thread begins life running the kernel-mode thread startup routine *KiThreadStartup*. *KiThreadStartup* lowers the thread's IRQL level from DPC/dispatch level to APC level and then calls the system initial thread routine, *PspUserThreadStartup*. The user-specified thread start address is passed as a parameter to this routine.

First, this function sets the Locale ID and the ideal processor in the TEB, based on the information present in kernel-mode data structures, and then it checks if thread creation actually failed. Next it calls *DbgkCreateThread*, which checks if image notifications were sent for the new process. If they weren't, and notifications are enabled, an image notification is sent first for the process and then for the image load of Ntdll.dll. Note that this is done in this stage rather than when the images were first mapped, because the process ID (which is required for the callouts) is not yet allocated at that time.

Once those checks are completed, another check is performed to see whether the process is a debuggee. If it is, then *PspUserThreadStartup* checks if the debugger notifications have already been sent for this process. If not, then a create process message is sent through the debug object (if one is present) so that the process startup debug event (CREATE_PROCESS_DEBUG_INFO) can be sent to the appropriate debugger process. This is followed by a similar thread startup debug event and by another debug event for the image load of Ntdll.dll. *DbgkCreateThread* then waits for the Windows subsystem to get the reply from the debugger (via the *ContinueDebugEvent* function).

Now that the debugger has been notified, *PspUserThreadStartup* looks at the result of the initial check on the thread's life. If it was killed on startup, the thread is terminated. This check is done after the debugger and image notifications to be sure that the kernel-mode and user-mode debuggers don't miss information on the thread, even if the thread never got a chance to run.

Otherwise, the routine checks whether application prefetching is enabled on the system and, if so, calls the prefetcher (and Superfetch) to process the prefetch instruction file (if it exists) and prefetch pages referenced during the first 10 seconds the last time the process ran. (For details on the prefetcher and Superfetch, see Chapter 9.)

PspUserThreadStartup then checks if the systemwide cookie in the SharedUserData structure has been set up yet. If it hasn't, it generates it based on a hash of system information such as the number of interrupts processed, DPC deliveries, and page faults. This systemwide cookie is used in the internal decoding and encoding of pointers, such as in the heap manager (for more information on heap manager security, see Chapter 9), to protect against certain classes of exploitation.

Finally, *PspUserThreadStartup* sets up the initial thunk context to run the image loader initialization routine (*LdrInitializeThunk* in *Ntdll.dll*), as well as the systemwide thread startup stub (*RtlUserThreadStart* in *Ntdll.dll*). These steps are done by editing the context of the thread in place and then issuing an *exit from system service* operation, which will load the specially crafted user context. The *LdrInitializeThunk* routine initializes the loader, heap manager, NLS tables, thread-local storage (TLS) and fiber-local storage (FLS) array, and critical section structures. It then loads any required DLLs and calls the DLL entry points with the `DLL_PROCESS_ATTACH` function code. (See the sidebar “Side-by-Side Assemblies” for a description of a mechanism Windows uses to address DLL versioning problems.)

Once the function returns, *NtContinue* will restore the new user context and return back to user mode—thread execution now truly starts.

RtlUserThreadStart will use the address of the actual image entry point and the start parameter and call the application. These two parameters have also already been pushed onto the stack by the kernel. This complicated series of events has two purposes. First of all, it allows the image loader inside *Ntdll.dll* to set up the process internally and behind the scenes so that other user-mode code can run properly (otherwise, it would have no heap, no thread local storage, and so on).

Second, having all threads begin in a common routine allows them to be wrapped in exception handling, so that when they crash, *Ntdll.dll* is aware of that and can call the unhandled exception filter inside *Kernel32.dll*. It is also able to coordinate thread exit on return from the thread’s start routine and to perform various cleanup work. Application developers can also call *SetUnhandledExceptionFilter* to add their own unhandled exception handling code.



Side-by-Side Assemblies

In order to isolate DLLs distributed with applications from DLLs that ship with the operating system, Windows allows applications to use private copies of these core DLLs. To use a private copy of a DLL instead of the one in the system directory, an application’s installation must include a file named *Application.exe.local* (where *Application* is the name of the application’s executable), which directs the loader to first look for DLLs in that directory. Note that any DLLs that are loaded from the list of *KnownDLLs* (DLLs that are permanently mapped into memory) or that are loaded by those DLLs cannot be redirected using this mechanism.

To further address application and DLL compatibility while allowing sharing, Windows implements the concept of *shared assemblies*. An *assembly* consists of a group of resources, including DLLs, and an XML manifest file that describes the assembly and its contents. An application references an assembly through the existence of its own XML manifest. The manifest can be a file in the application’s installation directory that has

the same name as the application with “.manifest” appended (for example, application.exe.manifest), or it can be linked into the application as a resource. The manifest describes the application and its dependence on assemblies.

There are two types of assemblies: private and shared. The difference between the two is that shared assemblies are digitally signed so that corruption or modification of their contents can be detected. In addition, shared assemblies are stored under the \Windows\Winsxs directory, whereas private assemblies are stored in an application’s installation directory. Thus, shared assemblies also have an associated catalog file (.cat) that contains its digital signature information. Shared assemblies can be “side-by-side” assemblies because multiple versions of a DLL can reside on a system simultaneously, with applications dependent on a particular version of a DLL always using that particular version.

An assembly’s manifest file typically has a name that includes the name of the assembly, version information, some text that represents a unique signature, and the extension “.manifest”. The manifests are stored in \Windows\Winsxs\Manifests, and the rest of the assembly’s resources are stored in subdirectories of \Windows\Winsxs that have the same name as the corresponding manifest files, with the exception of the trailing .manifest extension.

An example of a shared assembly is version 6 of the Windows common controls DLL, comctl32.dll. Its manifest file is named \Windows\Winsxs\Manifests\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.0.0_x-ww_1382d70a.manifest. It has an associated catalog file (which is the same name with the .cat extension) and a subdirectory of Winsxs that includes comctl32.dll.

Version 6 of Comctl32.dll added integration with Windows themes, and because applications not written with theme support in mind might not appear correctly with the new DLL, it’s available only to applications that explicitly reference the shared assembly containing it—the version of Comctl32.dll installed in \Windows\System32 is an instance of version 5.x, which is not theme aware. When an application loads, the loader looks for the application’s manifest, and if one exists, loads the DLLs from the assemblies specified. DLLs not included in assemblies referenced in the manifest are loaded in the traditional way. Legacy applications, therefore, link against the version in \Windows\System32, whereas theme-aware applications can specify the new version in their manifest.

A final advantage that shared assemblies have is that a publisher can issue a publisher configuration, which can redirect all applications that use a particular assembly to use an updated version. Publishers would do this if they were preserving backward compatibility while addressing bugs. Ultimately, however, because of the flexibility inherent in the assembly model, an application could decide to override the new setting and continue to use an older version.



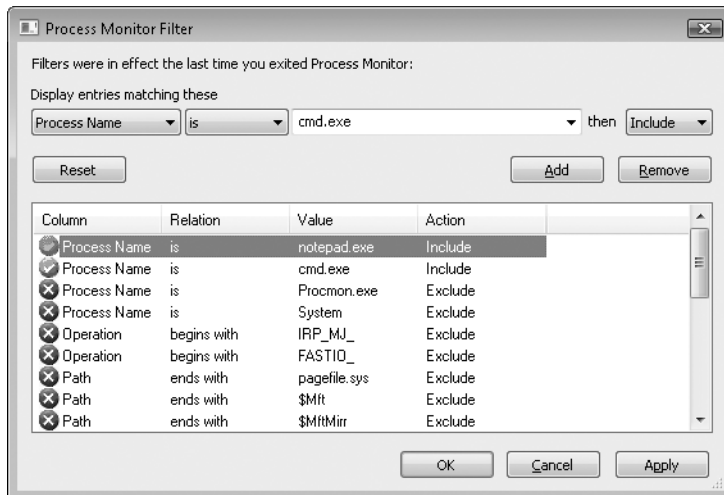
EXPERIMENT: Tracing Process Startup

Now that we've looked in detail at how a process starts up and the different operations required to begin executing an application, we're going to use Process Monitor to take a look at some of the file I/O and registry keys that are accessed during this process.

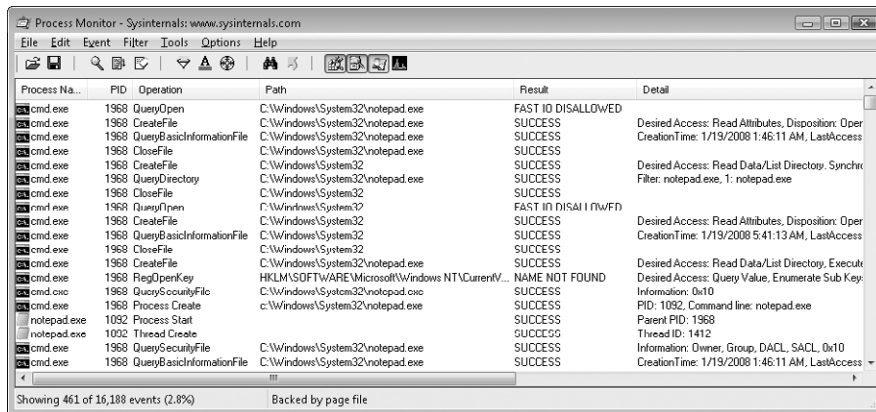
Although this experiment will not provide a complete picture of all the internal steps we've described, you'll be able to see several parts of the system in action, notably Prefetch and Superfetch, image file execution options and other compatibility checks, and the image loader's DLL mapping.

We're going to be looking at a very simple executable—Notepad.exe—and we will be launching it from a Command Prompt window (Cmd.exe). It's important that we look both at the operations inside Cmd.exe and those inside Notepad.exe. Recall that a lot of the user-mode work is performed by *CreateProcess*, which is called by the parent process before the kernel has created a new process object.

To set things up correctly, add two filters to Process Monitor: one for Cmd.exe, and one for Notepad.exe—these are the only two processes we want to include. It will be helpful to be sure that you don't have any currently running instances of these two processes so that you know you're looking at the right events. The filter window should look like this:

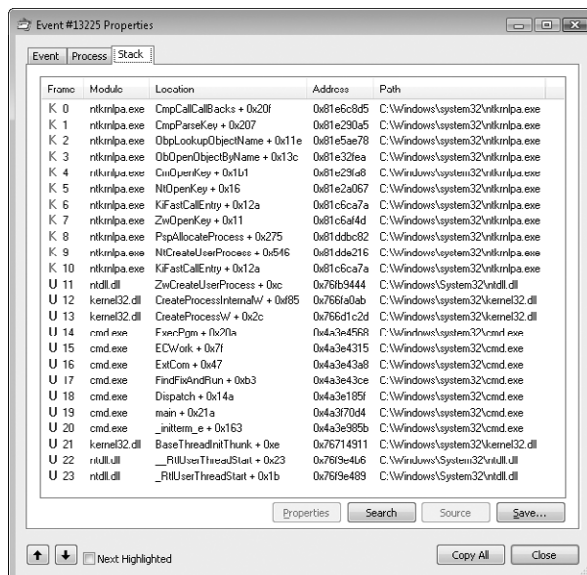


Next, make sure that event logging is currently disabled (clear File, Capture Events), and then start up the command prompt. Enable event logging (using the File menu again, or simply press CTRL+E or click the magnifying glass icon on the toolbar) and then enter **Notepad.exe** and press Enter. On a typical Windows Vista system, you should see anywhere between 500 and 1500 events appear. Go ahead and hide the Sequence and Time Of Day columns so that we can focus our attention on the columns of interest. Your window should look similar to the one shown next.



Just as described in Stage 1 of the *CreateProcess* flow, one of the first things to notice is that just before the process is started and the first thread is created, *Cmd.exe* does a registry read at *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options*. Because there were no image execution options associated with *Notepad.exe*, the process was created as is.

As with this and any other event in Process Monitor's log, you have the ability to see whether each part of the process creation flow was performed in user mode or kernel mode, and by which routines, by looking at the stack of the event. To do this, double-click on the *RegOpenKey* event mentioned and switch to the *Stack* tab. The following screen shows the standard stack on a 32-bit Windows Vista machine.



This stack shows that we have already reached the part of process creation performed in kernel mode (through *NtCreateUserProcess*) and that the helper routine *PspAllocateProcess* is responsible for this check.

Going down the list of events after the thread and process have been created, you will notice three groups of events. The first is a simple check for application compatibility flags, which will let the user-mode process creation code know if checks inside the application compatibility database are required through the shim engine.

This check is followed by multiple reads to Side-By-Side, Manifest, and MUI/Language keys, which are part of the assembly framework mentioned earlier. Finally, you may see file I/O to one or more .sdb files, which are the application compatibility databases on the system. This I/O is where additional checks are done to see if the shim engine needs to be invoked for this application. Since Notepad is a well behaved Microsoft program, it doesn't require any shims.

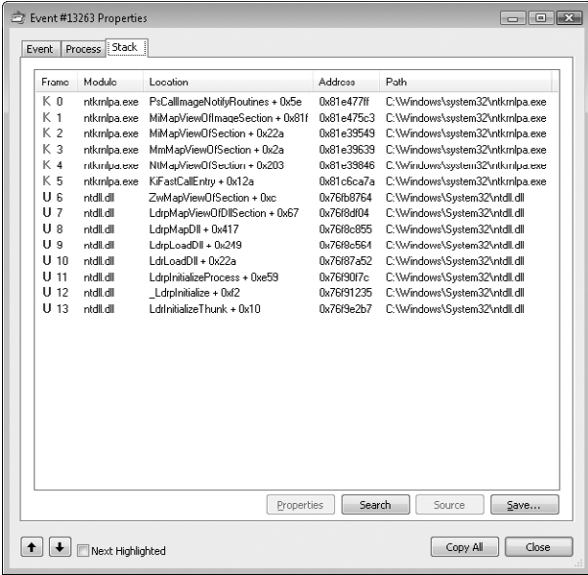
The following screen shows the next series of events, which happen inside the Notepad process itself. These are actions initiated by the user-mode thread startup wrapper in kernel mode, which performs the actions described earlier. The first two are the Notepad.exe and Ntdll.dll image load debug notification messages, which can only be generated now that code is running inside Notepad's process context and not the context for the command prompt.

Process Name	PID	Operation	Path	Result	Detail
notepad.exe	1092	QueryNameInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	Name: \Windows\System32\notepad.exe
notepad.exe	1092	Load Image	C:\Windows\System32\notepad.exe	SUCCESS	Image Base: 0x70000, Image Size: 0x26...
notepad.exe	1092	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7680000, Image Size: 0...
notepad.exe	1092	CreateFile	C:\Windows\System32\...	SUCCESS	Desired Access: Execute/Traverse, Syn...
notepad.exe	1092	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x768d000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x7680000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\vpct4.dll	SUCCESS	Image Base: 0x770a000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x76de000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x76ad000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\msvcrt.dll	SUCCESS	Image Base: 0x76b7000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\comdlg32.dll	SUCCESS	Image Base: 0x76eb000, Image Size: 0...
notepad.exe	1092	Load Image	C:\Windows\System32\shlwapi.dll	SUCCESS	Image Base: 0x7686000, Image Size: 0...
notepad.exe	1092	RegOpenKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Si...	REPARSE	Desired Access: Enumerate Sub Keys
notepad.exe	1092	RegOpenKey	HKLM\COMPONENTS\AssemblyStorageRoots	NAME NOT FOUND	Desired Access: Enumerate Sub Keys
notepad.exe	1092	QueryOpen	C:\Windows\System32\notepad.exe.Local	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\System32\notepad.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Dispos...
notepad.exe	1092	QueryOpen	C:\Windows\winsxs\x86_microsoft.windows.common-c...	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Attributes, Dispos...
notepad.exe	1092	QueryBasicInformationFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	CreationTime: 1/19/2008 5:43:08 AM, L...
notepad.exe	1092	CloseFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Execute/Traverse, Syn...
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Data/List Directo...
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\N...	REPARSE	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\N...	NAME NOT FOUND	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Safer\C...	SUCCESS	Desired Access: Query Value
notepad.exe	1092	RegOpenValue	HKLM\Software\Policies\Microsoft\Windows\Safer\C...	NAME NOT FOUND	Desired Access: Query Value

Next, the prefetcher kicks in, looking for a prefetch database file that has already been generated for Notepad. (For more information on the prefetcher, see Chapter 9). On a system where Notepad has already been run at least once, this database will exist, and the prefetcher will begin executing the commands specified inside it. If this is the case, scrolling down you will see multiple DLLs being read and queried. Unlike typical DLL loading, which is done by the user-mode image loader by looking at the import tables or when an application manually loads a DLL, these events are being generated by the prefetcher, which is already aware of the libraries that Notepad will require. Typical image loading of the DLLs required happens next, and you will see events similar to the ones shown here.

Process Name	PID	Operation	Path	Result	Detail
notepad.exe	1092	QueryNameInformationFile	C:\Windows\System32\notepad.exe	SUCCESS	Name: \Windows\System32\notepad.exe
notepad.exe	1092	Load Image	C:\Windows\System32\notepad.exe	SUCCESS	Image Base: 0x70000, Image Size: 0x28
notepad.exe	1092	Load Image	C:\Windows\System32\nltdll.dll	SUCCESS	Image Base: 0x76f60000, Image Size: 0
notepad.exe	1092	CreateFile	C:\Windows\System32\kernel32.dll	SUCCESS	Desired Access: Execute/Traverse, Sym
notepad.exe	1092	Load Image	C:\Windows\System32\advapi32.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x76de0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\user32.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\comctl32.dll	SUCCESS	Image Base: 0x770a0000, Image Size: 0
notepad.exe	1092	Load Image	C:\Windows\System32\shlwapi.dll	SUCCESS	Image Base: 0x776b0000, Image Size: 0
notepad.exe	1092	RegOpenKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Si...	REPARSE	Desired Access: Enumerate Sub Keys
notepad.exe	1092	RegOpenKey	HKLM\COMPONENTS\AssemblyStorageRoots	NAME NOT FOUND	Desired Access: Enumerate Sub Keys
notepad.exe	1092	QueryOpen	C:\Windows\System32\notepad.exe.Roots	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\System32\notepad.exe.Local	NAME NOT FOUND	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryOpen	C:\Windows\winsxs\x86_microsoft.windows.common-c...	FAST IO DISALLOWED	
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Attributes, Dispos
notepad.exe	1092	QueryBasicInformationFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	CreationTime: 7/19/2008 5:43:08 AM, L
notepad.exe	1092	CloseFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Execute/Traverse, Sym
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Read Data/List Directo
notepad.exe	1092	CreateFile	C:\Windows\winsxs\x86_microsoft.windows.common-c...	SUCCESS	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	REPARSE	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\System\CurrentControlSet\Control\SafeBoot\O...	NAME NOT FOUND	Desired Access: Query Value, Set Value
notepad.exe	1092	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\Safer\C...	SUCCESS	Desired Access: Query Value
notepad.exe	1092	RegOpenValue	HKLM\SOFTWARE\Policies\Microsoft\Windows\Safer...	NAME NOT FOUND	Desired Access: Query Value

These events are now being generated from code running inside user mode, which was called once the kernel-mode wrapper function finished its work. Therefore, these are the first events coming from *LdrpInitializeProcess*, which we mentioned is the internal system wrapper function for any new process, before the start address wrapper is called. You can confirm this on your own by looking at the stack of these events; for example, the kernel32.dll image load event, which is shown in the next screen.



Event #13263 Properties

Event Process Stack

Frame	Module	Location	Address	Path
K 0	ntkernel.exe	PtCallImageNotifyRoutines + 0x5e	0x81e477ff	C:\Windows\system32\ntkernel.exe
K 1	ntkernel.exe	MmMapViewOfFileSection + 0x81f	0x81e475c3	C:\Windows\system32\ntkernel.exe
K 2	ntkernel.exe	MmMapViewOfFileSection + 0x22a	0x81e39549	C:\Windows\system32\ntkernel.exe
K 3	ntkernel.exe	MmMapViewOfFileSection + 0x22a	0x81e39639	C:\Windows\system32\ntkernel.exe
K 4	ntkernel.exe	NMMapViewOfFileSection + 0x203	0x81e39846	C:\Windows\system32\ntkernel.exe
K 5	ntkernel.exe	KiFastCallEntry + 0x12a	0x81e5ca7a	C:\Windows\system32\ntkernel.exe
U 6	ntddi.dll	ZwMapViewOfFileSection + 0xc	0x76b8764	C:\Windows\System32\ntddi.dll
U 7	ntddi.dll	LdtpMapViewOfFileSection + 0x67	0x76b8804	C:\Windows\System32\ntddi.dll
U 8	ntddi.dll	LdtpMapDll + 0x17	0x76b8c855	C:\Windows\System32\ntddi.dll
U 9	ntddi.dll	LdtpLoadDll + 0x219	0x76b8d564	C:\Windows\System32\ntddi.dll
U 10	ntddi.dll	LdtpLoadDll + 0x22a	0x76b87a52	C:\Windows\System32\ntddi.dll
U 11	ntddi.dll	LdtpInitializeProcess + 0xe59	0x76b907c	C:\Windows\System32\ntddi.dll
U 12	ntddi.dll	LdtpInitialize + 0x2	0x76b91235	C:\Windows\System32\ntddi.dll
U 13	ntddi.dll	LdtpInitializeTrunk + 0x10	0x76b9e2b7	C:\Windows\System32\ntddi.dll

Properties Search Source Save...

Next Highlighted Copy All Close

Further events are generated by this routine and its associated helper functions until you finally reach events generated by the *WinMain* function inside Notepad, which is where code under the developer's control is now being executed. Describing in detail all the events and user-mode components that come into play during process execution would fill up this entire chapter, so exploration of any further events is left as an exercise for the reader.

Thread Internals

Now that we've dissected processes, let's turn our attention to the structure of a thread. Unless explicitly stated otherwise, you can assume that anything in this section applies to both user-mode threads and kernel-mode system threads (which are described in Chapter 2).

Data Structures

At the operating-system level, a Windows thread is represented by an executive thread (ETHREAD) block, which is illustrated in Figure 5-7. The ETHREAD block and the structures it points to exist in the system address space, with the exception of the thread environment block (TEB), which exists in the process address space (again, because user-mode components need to have access to it).

In addition, the Windows subsystem process (Csrss) also maintains a parallel structure for each thread created in a Windows subsystem application. Also, for threads that have called a Windows subsystem USER or GDI function, the kernel-mode portion of the Windows subsystem (Win32k.sys) maintains a per-thread data structure (called the W32THREAD structure) that the ETHREAD block points to.

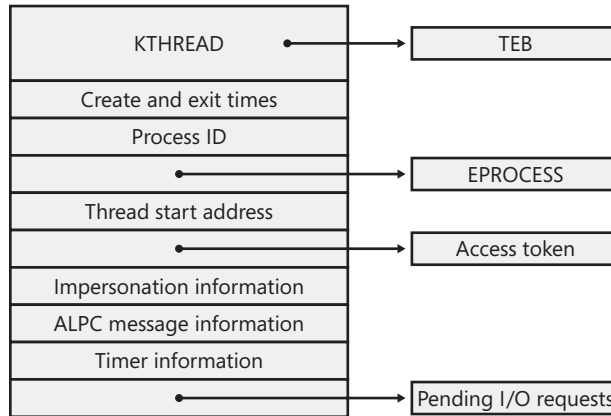


FIGURE 5-7 Structure of the executive thread block

Most of the fields illustrated in Figure 5-7 are self-explanatory. The first field is the kernel thread (KTHREAD) block. Following that are the thread identification information, the process identification information (including a pointer to the owning process so that its environment information can be accessed), security information in the form of a pointer to the access token and impersonation information, and finally, fields relating to ALPC messages and pending I/O requests. As you can see in Table 5-9, some of these key fields are covered in more detail elsewhere in this book. For more details on the internal structure of an ETHREAD block, you can use the kernel debugger *dt* command to display the format of the structure.

TABLE 5-9 Key Contents of the Executive Thread Block

Field Name	Value Taken from Image Header	Additional Information
KTHREAD	See Table 5-10.	
Thread time	Thread create and exit time information.	
Process identification	Process ID and pointer to EPROCESS block of the process that the thread belongs to.	
Start address	Address of thread start routine.	
Impersonation information	Access token and impersonation level (if the thread is impersonating a client).	Chapter 6
ALPC information	Message ID that the thread is waiting for and address of message.	Advanced local procedure calls (ALPC) (Chapter 3)
I/O information	List of pending I/O request packets (IRPs).	I/O system (Chapter 7)

Let’s take a closer look at two of the key thread data structures referred to in the preceding text: the KTHREAD block and the TEB. The KTHREAD block (also called the TCB, or thread control block) contains the information that the Windows kernel needs to access to perform thread scheduling and synchronization on behalf of running threads. Its layout is illustrated in Figure 5-8.

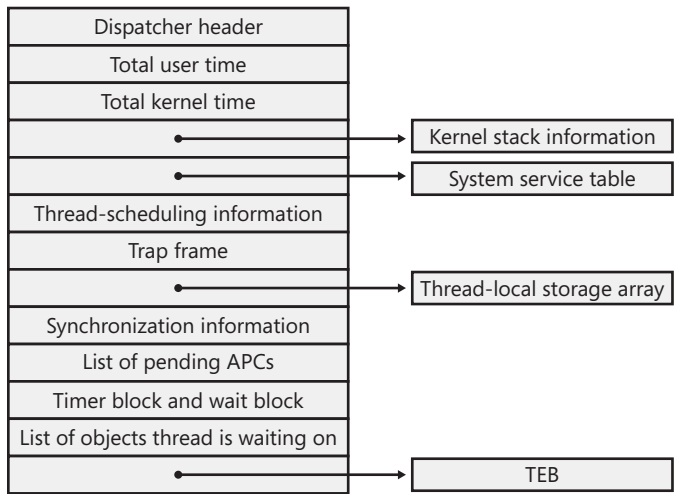


FIGURE 5-8 Structure of the kernel thread block

The key fields of the KTHREAD block are described briefly in Table 5-10.

TABLE 5-10 Key Contents of the KTHREAD Block

Element	Description	Additional Reference
Dispatcher header	Because the thread is an object that can be waited on, it starts with a standard kernel dispatcher object header.	Kernel dispatcher objects (Chapter 3)
Execution time	Total user and kernel CPU time.	
Cycle time	Total CPU cycle time.	Thread scheduling
Pointer to kernel stack information	Base and upper address of the kernel stack.	Memory management (Chapter 9)
Pointer to system service table	Each thread starts out with this field service table pointing to the main system service table (<i>KeServiceDescriptorTable</i>). When a thread first calls a Windows GUI service, its system service table is changed to one that includes the GDI and USER services in Win32k.sys.	System service dispatching (Chapter 3)

Element	Description	Additional Reference
Scheduling information	Base and current priority, quantum target, quantum reset, affinity mask, ideal processor, deferred processor, next processor, scheduling state, freeze count, suspend count, adjust increment and adjust reason.	Thread scheduling
Wait blocks	The thread block contains four built-in wait blocks so that wait blocks don't have to be allocated and initialized each time the thread waits for something. (One wait block is dedicated to timers.)	Synchronization (Chapter 3)
Wait information	List of objects the thread is waiting for, wait reason, IRQL at the time of wait, result of the wait, and time at which the thread entered the wait state.	Synchronization (Chapter 3)
Mutant list	List of mutant objects the thread owns.	Synchronization (Chapter 3)
APC queues	List of pending user-mode and kernel-mode APCs, alerted flag, and flags to disable APCs.	Asynchronous procedure call (APC) interrupts (Chapter 3)
Timer block	Built-in timer block (also a corresponding wait block).	
Suspend APC and semaphore	Built-in APC and semaphore used when suspending and resuming a thread.	Synchronization (Chapter 3)
Queue	Pointer to queue object that the thread is associated with.	Synchronization (Chapter 3)
Gate	Pointer to gate object that the thread is waiting on.	Synchronization (Chapter 3)
Pointer to TEB	Thread ID, TLS and FLS information, PEB pointer, and Winsock, RPC, GDI, OpenGL, and other user-mode information.	



EXPERIMENT: Displaying ETHREAD and KTHREAD Structures

The ETHREAD and KTHREAD structures can be displayed with the *dt* command in the kernel debugger. The following output shows the format of an ETHREAD on a 32-bit system:

```

1kd> dt nt!_ethread
nt!_ETHREAD
+0x000 Tcb                : _KTHREAD
+0x1e0 CreateTime         : _LARGE_INTEGER
+0x1e8 ExitTime           : _LARGE_INTEGER
+0x1e8 KeyedWaitChain     : _LIST_ENTRY
+0x1f0 ExitStatus         : Int4B
+0x1f0 OfsChain           : Ptr32 Void
+0x1f4 PostBlockList     : _LIST_ENTRY
+0x1f4 ForwardLinkShadow : Ptr32 Void
+0x1f8 StartAddress       : Ptr32 Void
+0x1fc TerminationPort   : Ptr32 _TERMINATION_PORT
+0x1fc ReaperLink        : Ptr32 _ETHREAD
+0x1fc KeyedWaitValue     : Ptr32 Void
+0x1fc Win32StartParameter : Ptr32 Void
+0x200 ActiveTimerListLock : Uint4B
+0x204 ActiveTimerListHead : _LIST_ENTRY
+0x20c Cid                 : _CLIENT_ID
+0x214 KeyedWaitSemaphore : _KSEMAPHORE
+0x214 AlpcWaitSemaphore  : _KSEMAPHORE
+0x228 ClientSecurity     : _PS_CLIENT_SECURITY_CONTEXT
+0x22c IrpList            : _LIST_ENTRY
+0x234 TopLevelIrp        : Uint4B
+0x238 DeviceToVerify     : Ptr32 _DEVICE_OBJECT
+0x23c RateControlApc     : Ptr32 _PSP_RATE_APC
+0x240 Win32StartAddress  : Ptr32 Void
+0x244 SparePtr0         : Ptr32 Void
+0x248 ThreadListEntry    : _LIST_ENTRY
+0x250 RundownProtect     : _EX_RUNDOWN_REF
+0x254 ThreadLock        : _EX_PUSH_LOCK
+0x258 ReadClusterSize   : Uint4B
+0x25c MmLockOrdering    : Int4B
+0x260 CrossThreadFlags  : Uint4B
+0x260 Terminated       : Pos 0, 1 Bit
+0x260 ThreadInserted    : Pos 1, 1 Bit
+0x260 HideFromDebugger  : Pos 2, 1 Bit
+0x260 ActiveImpersonationInfo : Pos 3, 1 Bit
+0x260 SystemThread      : Pos 4, 1 Bit
+0x260 HardErrorsAreDisabled : Pos 5, 1 Bit
+0x260 BreakOnTermination : Pos 6, 1 Bit
+0x260 SkipCreationMsg   : Pos 7, 1 Bit
+0x260 SkipTerminationMsg : Pos 8, 1 Bit
+0x260 CopyTokenOnOpen   : Pos 9, 1 Bit
+0x260 ThreadIoPriority   : Pos 10, 3 Bits
+0x260 ThreadPagePriority : Pos 13, 3 Bits
+0x260 RundownFail       : Pos 16, 1 Bit
+0x264 SameThreadPassiveFlags : Uint4B
+0x264 ActiveExWorker    : Pos 0, 1 Bit
+0x264 ExWorkerCanWaitUser : Pos 1, 1 Bit

```

```

+0x264 MemoryMaker      : Pos 2, 1 Bit
+0x264 ClonedThread    : Pos 3, 1 Bit
+0x264 KeyedEventInUse : Pos 4, 1 Bit
+0x264 RateApcState    : Pos 5, 2 Bits
+0x264 SelfTerminate   : Pos 7, 1 Bit
+0x268 SameThreadApcFlags : Uint4B
+0x268 Spare           : Pos 0, 1 Bit
+0x268 StartAddressInvalid : Pos 1, 1 Bit
+0x268 EtwPageFaultCalloutActive : Pos 2, 1 Bit
+0x268 OwnsProcessWorkingSetExclusive : Pos 3, 1 Bit
+0x268 OwnsProcessWorkingSetShared : Pos 4, 1 Bit
+0x268 OwnsSystemWorkingSetExclusive : Pos 5, 1 Bit
+0x268 OwnsSystemWorkingSetShared : Pos 6, 1 Bit
+0x268 OwnsSessionWorkingSetExclusive : Pos 7, 1 Bit
+0x269 OwnsSessionWorkingSetShared : Pos 0, 1 Bit
+0x269 OwnsProcessAddressSpaceExclusive : Pos 1, 1 Bit
+0x269 OwnsProcessAddressSpaceShared : Pos 2, 1 Bit
+0x269 SuppressSymbolLoad : Pos 3, 1 Bit
+0x269 Prefetching     : Pos 4, 1 Bit
+0x269 OwnsDynamicMemoryShared : Pos 5, 1 Bit
+0x269 OwnsChangeControlAreaExclusive : Pos 6, 1 Bit
+0x269 OwnsChangeControlAreaShared : Pos 7, 1 Bit
+0x26a PriorityRegionActive : Pos 0, 4 Bits
+0x26c CacheManagerActive : UChar
+0x26d DisablePageFaultClustering : UChar
+0x26e ActiveFaultCount : UChar
+0x270 AlpcMessageId   : Uint4B
+0x274 AlpcMessage     : Ptr32 Void
+0x274 AlpcReceiveAttributeSet : Uint4B
+0x278 AlpcWaitListEntry : _LIST_ENTRY
+0x280 CacheManagerCount : Uint4B

```

The KTHREAD can be displayed with a similar command:

```

!kd> dt nt!_kthread
nt!_KTHREAD
+0x000 Header          : _DISPATCHER_HEADER
+0x010 CycleTime      : Uint8B
+0x018 HighCycleTime  : Uint4B
+0x020 QuantumTarget  : Uint8B
+0x028 InitialStack   : Ptr32 Void
+0x02c StackLimit     : Ptr32 Void
+0x030 KernelStack    : Ptr32 Void
+0x034 ThreadLock     : Uint4B
+0x038 ApcState       : _KAPC_STATE
+0x038 ApcStateFill   : [23] UChar
+0x04f Priority        : Char
+0x050 NextProcessor  : Uint2B
+0x052 DeferredProcessor : Uint2B
+0x054 ApcQueueLock   : Uint4B
+0x058 ContextSwitches : Uint4B
+0x05c State          : UChar
+0x05d NpxState       : UChar
+0x05e WaitIrql       : UChar
+0x05f WaitMode       : Char
+0x060 WaitStatus     : Int4B

```



EXPERIMENT: Using the Kernel Debugger *!thread* Command

The kernel debugger *!thread* command dumps a subset of the information in the thread data structures. Some key elements of the information the kernel debugger displays can't be displayed by any utility: internal structure addresses; priority details; stack information; the pending I/O request list; and, for threads in a wait state, the list of objects the thread is waiting for.

To display thread information, use either the *!process* command (which displays all the thread blocks after displaying the process block) or the *!thread* command to dump a specific thread. The output of the thread information, along with some annotations of key fields, is shown here:

Address of ETHREAD	Thread ID	Address of thread environment block	
THREAD 83160f0	Cid: 9f.3d7eb:	7ffdc000	win32Thread: e153d2c8
WAIT: (WrUserRequest)	UserMode Non-Alertable		Thread state
808e9d60	SynchronizationEvent		Objects being waited on
Not impersonating			
Owning Process 81b44880			Address of EPROCESS for owning process
Wait Time (seconds)	953945		
Context Switch Count	2697		LargeStack
UserTime	0:00:00.0289		Actual thread start address
KernelTime	0:00:04.0644		
Start Address kernal32!BaseProcessStart (0x77e8f268)			
win32 Start Address 0x020d9d98			Address of user thread function
Stack Init f7818000 Current f7817bb0 Base f7818000 Limit f7812000 Call 0			
Priority 14 BasePriority 9 PriorityDecrement 6 DecrementCount 13			Priority information
kernal stack not resident.			
ChildEBP RetAddr	Args to Child		
F7817bb0 8008f430 00000001 00000000 00000000	ntoskrnl!KiSwapThreadExit		
F7817c50 de0119ec 00000001 00000000 00000000	ntoskrnl!KeWaitForSingleObject+0x2a0		
F7817cc0 de0123f4 00000001 00000000 00000000	win32k!xxxSleepThread+0x23c		
F7817d10 de01f2f0 00000001 00000000 00000000	win32k!xxxInternalGetMessage+0x504		
F7817d80 800bab58 00000001 00000000 00000000	win32k!NtUserGetMessage+0x58		
F7817df0 77d887d0 00000001 00000000 00000000	ntoskrnl!KiSystemServiceEndAddress+0x4		
0012fef0 00000000 00000001 00000000 00000000	user32!GetMessageW+0x30		

Stack dump



EXPERIMENT: Viewing Thread Information

The following output is the detailed display of a process produced by using the Tlist utility in the Debugging Tools for Windows. Notice that the thread list shows the “Win32StartAddr.” This is the address passed to the *CreateThread* function by the application. All the other utilities, except Process Explorer, that show the thread start address show the actual start address (a function in Ntdll.dll), not the application-specified start address.

```
C:\> tlist winword
2400 WINWORD.EXE          WinInt5E_Chapter06.doc [Compatibility Mode] - Microsoft Word
  CWD:          C:\Users\Alex Ionescu\Documents\
  CmdLine:      "C:\Program Files\Microsoft Office\Office12\WINWORD.EXE" /n /dde
  VirtualSize:  310656 KB   PeakVirtualSize:  343552 KB
  WorkingSetSize: 91548 KB   PeakWorkingSetSize:100788 KB
  NumberOfThreads: 6
  2456 Win32StartAddr:0x2f7f10cc LastErr:0x00000000 State:Waiting
  1452 Win32StartAddr:0x6882f519 LastErr:0x00000000 State:Waiting
  2464 Win32StartAddr:0x6b603850 LastErr:0x00000000 State:Waiting
  3036 Win32StartAddr:0x690dc17f LastErr:0x00000002 State:Waiting
  3932 Win32StartAddr:0x775cac65 LastErr:0x00000102 State:Waiting
  3140 Win32StartAddr:0x687d6ffd LastErr:0x000003f0 State:Waiting
12.0.4518.1014 shp  0x2F7F0000  C:\Program Files\Microsoft Office\Office12\
  WINWORD.EXE
  6.0.6000.16386 shp  0x777D0000  C:\Windows\system32\Ntdll.dll
  6.0.6000.16386 shp  0x764C0000  C:\Windows\system32\kernel32.dll
  §                list of DLLs loaded in process
```

The TEB, illustrated in Figure 5-9, is the only data structure explained in this section that exists in the process address space (as opposed to the system space).

The TEB stores context information for the image loader and various Windows DLLs. Because these components run in user mode, they need a data structure writable from user mode. That’s why this structure exists in the process address space instead of in the system space, where it would be writable only from kernel mode. You can find the address of the TEB with the kernel debugger *!thread* command.

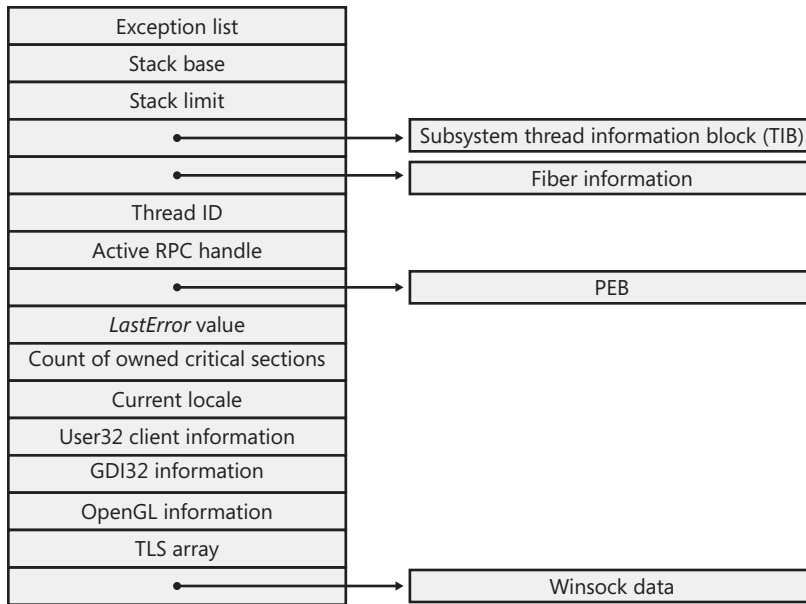


FIGURE 5-9 Fields of the thread environment block



EXPERIMENT: Examining the TEB

You can dump the TEB structure with the `!teb` command in the kernel debugger. The output looks like this:

```
kd> !teb
TEB at 7ffde000
  ExceptionList:      019e8e44
  StackBase:         019f0000
  StackLimit:        019db000
  SubSystemTib:      00000000
  FiberData:         00001e00
  ArbitraryUserPointer: 00000000
  Self:              7ffde000
  EnvironmentPointer: 00000000
  ClientId:          00000bcc . 00000864
  RpcHandle:         00000000
  Tls Storage:       7ffde02c
  PEB Address:       7ffd9000
  LastErrorValue:    0
  LastStatusValue:   c0000139
  Count Owned Locks: 0
  HardErrorMode:     0
```

Kernel Variables

As with processes, a number of Windows kernel variables control how threads run. Table 5-11 shows the kernel-mode kernel variables that relate to threads.

TABLE 5-11 Thread-Related Kernel Variables

Variable	Type	Description
<i>PspCreateThreadNotifyRoutine</i>	Array of executive callback objects	Array of callback objects describing the routines to be called on thread creation and deletion (maximum of 64)
<i>PspCreateThreadNotifyRoutineCount</i>	32-bit integer	Count of registered thread-notification routines

Performance Counters

Most of the key information in the thread data structures is exported as performance counters, which are listed in Table 5-12. You can extract much information about the internals of a thread just by using the Reliability and Performance Monitor in Windows.

TABLE 5-12 Thread-Related Performance Counters

Object: Counter	Function
Process: Priority Base	Returns the current base priority of the process. This is the starting priority for threads created within this process.
Thread: % Privileged Time	Describes the percentage of time that the thread has run in kernel mode during a specified interval.
Thread: % Processor Time	Describes the percentage of CPU time that the thread has used during a specified interval. This count is the sum of % Privileged Time and % User Time.
Thread: % User Time	Describes the percentage of time that the thread has run in user mode during a specified interval.
Thread: Context Switches/Sec	Returns the number of context switches per second that the system is executing.
Thread: Elapsed Time	Returns the amount of CPU time (in seconds) that the thread has consumed.
Thread: ID Process	Returns the process ID of the thread's process.
Thread: ID Thread	Returns the thread's thread ID. This ID is valid only during the thread's lifetime because thread IDs are reused.
Thread: Priority Base	Returns the thread's current base priority. This number might be different from the thread's starting base priority.
Thread: Priority Current	Returns the thread's current dynamic priority.
Thread: Start Address	Returns the thread's starting virtual address (<i>Note:</i> This address will be the same for most threads.)

Object: Counter	Function
Thread: Thread State	Returns a value from 0 through 7 relating to the current state of the thread.
Thread: Thread Wait Reason	Returns a value from 0 through 19 relating to the reason why the thread is in a wait state.

Relevant Functions

Table 5-13 shows the Windows functions for creating and manipulating threads. This table doesn't include functions that have to do with thread scheduling and priorities—those are included in the section “Thread Scheduling” later in this chapter.

TABLE 5-13 Windows Thread Functions

Function	Description
<i>CreateThread</i>	Creates a new thread
<i>CreateRemoteThread</i>	Creates a thread in another process
<i>OpenThread</i>	Opens an existing thread
<i>ExitThread</i>	Ends execution of a thread normally
<i>TerminateThread</i>	Terminates a thread
<i>IsThreadAFiber</i>	Returns whether the current thread is a fiber
<i>GetExitCodeThread</i>	Gets another thread's exit code
<i>GetThreadTimes</i>	Returns timing information for a thread
<i>QueryThreadCycleTime</i>	Returns CPU clock cycle information for a thread
<i>GetCurrentThread</i>	Returns a pseudo handle for the current thread
<i>GetCurrentProcessId</i>	Returns the thread ID of the current thread
<i>GetThreadId</i>	Returns the thread ID of the specified thread
<i>Get/SetThreadContext</i>	Returns or changes a thread's CPU registers
<i>GetThreadSelectorEntry</i>	Returns another thread's descriptor table entry (applies only to x86 systems)

Birth of a Thread

A thread's life cycle starts when a program creates a new thread. The request filters down to the Windows executive, where the process manager allocates space for a thread object and calls the kernel to initialize the kernel thread block. The steps in the following list are taken inside the Windows *CreateThread* function in Kernel32.dll to create a Windows thread.

1. *CreateThread* converts the Windows API parameters to native flags and builds a native structure describing object parameters (OBJECT_ATTRIBUTES). See Chapter 3 for more information.

2. *CreateThread* builds an attribute list with two entries: client ID and TEB address. This allows *CreateThread* to receive those values once the thread has been created. (For more information on attribute lists, see the section “Flow of *CreateProcess*” earlier in this chapter.)
3. *NtCreateThreadEx* is called to create the user-mode context and probe and capture the attribute list. It then calls *PspCreateThread* to create a suspended executive thread object. For a description of the steps performed by this function, see the descriptions of Stage 3 and Stage 5 in the section “Flow of *CreateProcess*.”
4. *CreateThread* allocates an activation stack for the thread used by side-by-side assembly support. It then queries the activation stack to see if it requires activation, and does so if needed. The activation stack pointer is saved in the new thread’s TEB.
5. *CreateThread* notifies the Windows subsystem about the new thread, and the subsystem does some setup work for the new thread.
6. The thread handle and the thread ID (generated during step 3) are returned to the caller.
7. Unless the caller created the thread with the `CREATE_SUSPENDED` flag set, the thread is now resumed so that it can be scheduled for execution. When the thread starts running, it executes the steps described in the earlier section “Stage 7: Performing Process Initialization in the Context of the New Process” before calling the actual user’s specified start address.

Examining Thread Activity

Examining thread activity is especially important if you are trying to determine why a process that is hosting multiple services is running (such as `Svchost.exe`, `Dllhost.exe`, or `Lsass.exe`) or why a process is hung.

There are several tools that expose various elements of the state of Windows threads: WinDbg (in user-process attach and kernel debugging mode), the Reliability and Performance Monitor, and Process Explorer. (The tools that show thread-scheduling information are listed in the section “Thread Scheduling.”)

To view the threads in a process with Process Explorer, select a process and open the process properties (double-click on the process or click on the Process, Properties menu item). Then click on the Threads tab. This tab shows a list of the threads in the process and three columns of information. For each thread it shows the percentage of CPU consumed (based on the refresh interval configured), the number of context switches to the thread, and the thread start address. You can sort by any of these three columns.

New threads that are created are highlighted in green, and threads that exit are highlighted in red. (The highlight duration can be configured with the Options, Configure Highlighting menu item.) This might be helpful to discover unnecessary thread creation occurring in a process. (In general, threads should be created at process startup, not every time a request is processed inside a process.)

As you select each thread in the list, Process Explorer displays the thread ID, start time, state, CPU time counters, number of context switches, and the base and current priority. There is a Kill button, which will terminate an individual thread, but this should be used with extreme care.

The best way to measure actual CPU activity with Process Explorer is to add the clock cycle delta column, which uses the clock cycle counter designed for thread run-time accounting (as described later in this chapter). Because many threads run for such a short amount of time that they are seldom (if ever) the currently running thread when the clock interval timer interrupt occurs, they are not charged for much of their CPU time. The total number of clock cycles represents the actual number of processor cycles that each thread in the process accrued. It is independent of the clock interval timer's resolution because the count is maintained internally by the processor at each cycle and updated by Windows at each interrupt entry (a final accumulation is done before a context switch).

The thread start address is displayed in the form "*module!function*", where *module* is the name of the .exe or .dll. The function name relies on access to symbol files for the module. (See "Experiment: Viewing Process Details with Process Explorer" in Chapter 1.) If you are unsure what the module is, click the Module button. This opens an Explorer file properties window for the module containing the thread's start address (for example, the .exe or .dll).



Note For threads created by the Windows *CreateThread* function, Process Explorer displays the function passed to *CreateThread*, not the actual thread start function. That is because all Windows threads start at a common thread startup wrapper function (*RtlUserThreadStart* in *Ntdll.dll*). If Process Explorer showed the actual start address, most threads in processes would appear to have started at the same address, which would not be helpful in trying to understand what code the thread was executing. However, if Process Explorer can't query the user-defined startup address (such as in the case of a protected process), it will show the wrapper function, so you will see all threads starting at *RtlUserThreadStart*.

However, the thread start address displayed might not be enough information to pinpoint what the thread is doing and which component within the process is responsible for the CPU consumed by the thread. This is especially true if the thread start address is a generic startup function (for example, if the function name does not indicate what the thread is actually doing). In this case, examining the thread stack might answer the question. To view the stack for a thread, double-click on the thread of interest (or select it and click the Stack button).

Process Explorer displays the thread's stack (both user and kernel, if the thread was in kernel mode).



Note While the user-mode debuggers (WinDbg, Ntsd, and Cdb) permit you to attach to a process and display the user stack for a thread, Process Explorer shows both the user and kernel stack in one easy click of a button. You can also examine user and kernel thread stacks using WinDbg in local kernel debugging mode.

Viewing the thread stack can also help you determine why a process is hung. As an example, on one system, Microsoft Office PowerPoint was hanging for one minute on startup. To determine why it was hung, after starting PowerPoint, Process Explorer was used to examine the thread stack of the one thread in the process. The result is shown in Figure 5-10.

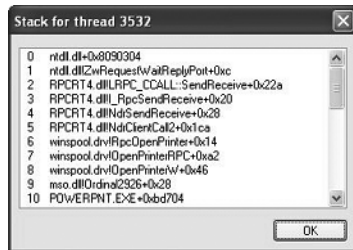


FIGURE 5-10 Hung thread stack in PowerPoint

This thread stack shows that PowerPoint (line 10) called a function in *Mso.dll* (the central Microsoft Office DLL), which called the *OpenPrinterW* function in *Winspool.drv* (a DLL used to connect to printers). *Winspool.drv* then dispatched to a function *OpenPrinterRPC*, which then called a function in the RPC runtime DLL, indicating it was sending the request to a remote printer. So, without having to understand the internals of PowerPoint, the module and function names displayed on the thread stack indicate that the thread was waiting to connect to a network printer. On this particular system, there was a network printer that was not responding, which explained the delay starting PowerPoint. (Microsoft Office applications connect to all configured printers at process startup.) The connection to that printer was deleted from the user's system, and the problem went away.

Finally, when looking at 32-bit applications running on 64-bit systems as a Wow64 process (see Chapter 3 for more information on Wow64), Process Explorer shows both the 32-bit and 64-bit stack for threads. Because at the time of the system call proper, the thread has been switched to a 64-bit stack and context, simply looking at the thread's 64-bit stack would reveal only half the story—the 64-bit part of the thread, with Wow64's thunking code. So, when examining Wow64 processes, be sure to take into account both the 32-bit and 64-bit stacks. An example of a Wow64 thread inside Microsoft Office Word 2007 is shown in Figure 5-11. The stack frames highlighted in the box are the 32-bit stack frames from the 32-bit stack.

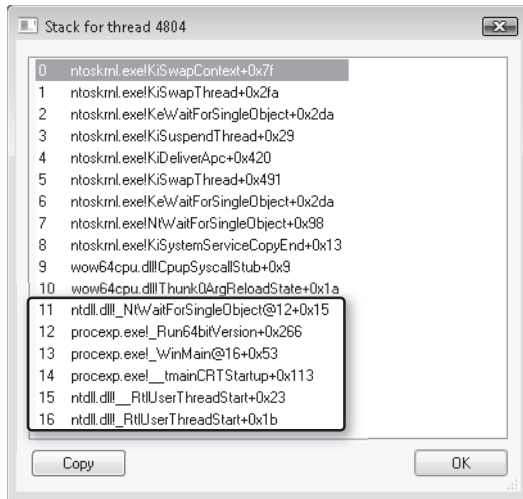


FIGURE 5-11 Example Wow64 stack

Limitations on Protected Process Threads

As we discussed in the process internals section, protected processes have several limitations in terms of which access rights will be granted, even to the users with the highest privileges on the system. These limitations also apply to threads inside such a process. This ensures that the actual code running inside the protected process cannot be hijacked or otherwise affected through standard Windows functions, which require the access rights in Table 5-14.

TABLE 5-14 Thread Access Rights Denied for Threads Inside a Protected Process

Object: Access Mask	Function
Thread: THREAD_ALL_ACCESS	Disables full access to a thread inside a protected process.
Thread: THREAD_DIRECT_IMPERSONATION	Disables impersonating a thread inside a protected process.
Thread: THREAD_GET_CONTEXT, THREAD_SET_CONTEXT	Disables accessing and/or modifying the CPU context (registers and stack) of a thread inside a protected process.
Thread: THREAD_QUERY_INFORMATION	Disables querying all information on a thread inside a protected process. However, a new access right was added, THREAD_QUERY_LIMITED_INFORMATION, that grants limited access to information on the thread.

Object: Access Mask	Function
Thread: THREAD_SET_INFORMATION	Disables setting all information on a thread inside a protected process. However, a new access right was added, THREAD_SET_LIMITED_INFORMATION, that grants limited access to modifying information on the thread.
Thread: THREAD_SET_THREAD_TOKEN	Disables setting an impersonation token for a thread inside a protected process.
Thread: THREAD_TERMINATE	Disables terminating a thread inside a protected process. Note that terminating all threads atomically through process termination is allowed.

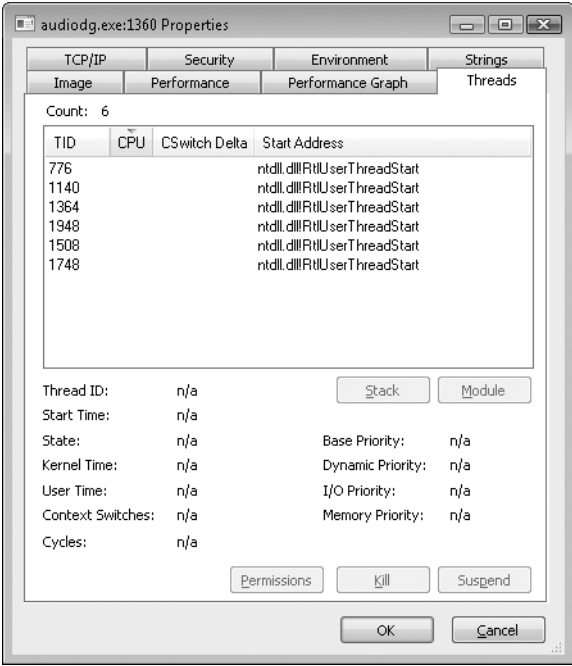


EXPERIMENT: Viewing Protected Process Thread Information

In the previous section, we took a look at how Process Explorer can be helpful in examining thread activity to determine the cause of potential system or application issues. This time, we'll use Process Explorer to look at a protected process and see how the different access rights being denied affect its ability and usefulness on such a process.

Find the Audiodg.exe service inside the process list. This is a process responsible for much of the core work behind the user-mode audio stack in Windows Vista, and it requires protection to ensure that high-definition decrypted audio content does not leak out to untrusted sources. Bring up the process properties view and take a look at the Image tab. Notice how the numbers for WS Private, WS Shareable, and WS Shared are 0, although the total Working Set is still displayed. This is an example of the THREAD_QUERY_INFORMATION versus THREAD_QUERY_LIMITED_INFORMATION rights.

More importantly, take a look at the Threads tab. As you can see here, Process Explorer is unable to show the Win32 thread start address and instead displays the standard thread start wrapper inside Ntdll.dll. If you try clicking on the Stack button, you'll get an error, because Process Explorer needs to read the virtual memory inside the protected process, which it can't do.



Finally, note that although the Base and Dynamic priorities are shown, the I/O and Memory priorities are not, another example of the limited versus full query information access right. As you try to kill a thread inside `Audiodg.exe`, notice yet another access denied error: recall the lack of `THREAD_TERMINATE` access shown earlier in Table 5-14.

Worker Factories (Thread Pools)

Worker factories refer to the internal mechanism used to implement user-mode thread pools. Prior to Windows Vista, the thread pool routines were completely implemented in user mode inside the `Ntdll.dll` library, and the Windows API provided various routines to call into the relevant routines, which provided waitable timers, wait callbacks, and automatic thread creation and deletion depending on the amount of work being done.



Note Information on the new thread pool API is available on MSDN at <http://msdn2.microsoft.com/en-us/library/ms686760.aspx>. It includes information on the APIs introduced and the APIs retired, as well as important differences in certain details of the way the two APIs are implemented.

In Windows Vista, the thread pool implementation in user mode was completely re-architected, and part of the management functionality has been moved to kernel mode in order to improve efficiency and performance and minimize complexity. The original thread pool implementation required the user-mode code inside Ntdll.dll to remain aware of how many threads were currently active as worker threads, and to enlarge this number in periods of high demand.

Because querying the information necessary to make this decision, as well as the work to create the threads, took place in user mode, several system calls were required that could have been avoided if these operations were performed in kernel mode. Moving this code into kernel mode means fewer transitions between user and kernel mode, and it allows Ntdll.dll to manage the thread pool itself and not the system mechanisms behind it. It also provides other benefits, such as the ability to remotely create a thread pool in a process other than the calling process (although possible in user mode, it would be very complex given the necessity of using APIs to access the remote process's address space).

The functionality in Windows Vista is introduced by a new object manager type called *TpWorkerFactory*, as well as four new native system calls for managing the factory and its workers—*NtCreateWorkerFactory*, *NtWorkerFactoryWorkerReady*, *NtReleaseWorkerFactoryWorker*, *NtShutdownWorkerFactory*—two new query/set native calls (*NtQueryInformationWorkerFactory* and *NtSetInformationWorkerFactory*), and a new wait call, *NtWaitForWorkViaWorkerFactory*.

Just like other native system calls, these calls provide user mode with a handle to the *TpWorkerFactory* object, which contains information such as the name and object attributes, the desired access mask, and a security descriptor. Unlike other system calls wrapped by the Windows API, however, thread pool management is handled by Ntdll.dll's native code, which means that developers work with an opaque descriptor (a TP_WORK pointer) owned by Ntdll.dll, in which the actual handle is stored.

As its name suggests, the worker factory implementation is responsible for allocating worker threads (and calling the given user-mode worker thread entry point), maintaining a minimum and maximum thread count (allowing for either permanent worker pools or totally dynamic pools), as well as other accounting information. This enables operations such as shutting down the thread pool to be performed with a single call to the kernel, because the kernel has been the only component responsible for thread creation and termination.

Because the kernel dynamically creates new threads as requested, this also increases the scalability of applications using the new thread pool implementation. Developers have always been able to take advantage of as many threads as possible (based on the number of processors on the system) through the old implementation, but through support for dynamic processors in Windows Vista (see the section on this topic later in this chapter), it's now possible for applications using thread pools to automatically take advantage of new processors added at run time.

It's important to note that the new worker factory support is merely a wrapper to manage mundane tasks that would otherwise have to be performed in user mode (at a loss of performance). Many of the improvements in the new thread pool code are the result of changes in the Ntdll.dll side of this architecture. Also, it is not the worker factory code that provides the scalability, wait internals, and efficiency of work processing. Instead, it is a much older component of Windows that we have already discussed—I/O completion ports, or more correctly, kernel queues (KQUEUE; see Chapter 7 for more information).

In fact, when creating a worker factory, an I/O completion port must have already been created by user mode, and the handle needs to be passed on. It is through this I/O completion port that the user-mode implementation will queue work and also wait for work—but by calling the worker factory system calls instead of the I/O completion port APIs. Internally, however, the “release” worker factory call (which queues work) is a wrapper around *IoSetIoCompletion*, which increases pending work, while the “wait” call is a wrapper around *IoRemoveIoCompletion*. Both these routines call into the kernel queue implementation.

Therefore, the job of the worker factory code is to manage either a persistent, static, or dynamic thread pool; wrap the I/O completion port model into interfaces that try to prevent stalled worker queues by automatically creating dynamic threads; and to simplify global cleanup and termination operations during a factory shutdown request (as well as to easily block new requests against the factory in such a scenario).

Unfortunately, the data structures used by the worker factory implementation are not in the public symbols, but it is still possible to look at some worker pools, as we'll show in the next experiment.

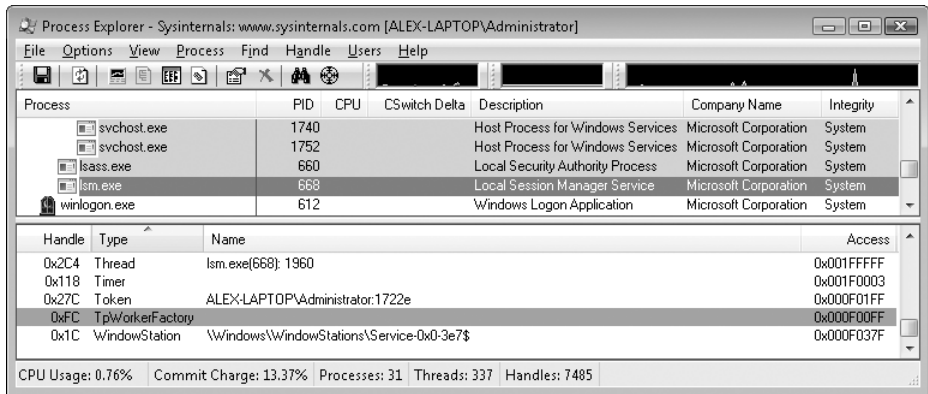


EXPERIMENT: Looking at Thread Pools

Because of the more efficient and simpler thread pool implementation in Windows Vista, many core system components and applications were updated to make use of it. One of the ways to identify which processes are using a worker factory is to look at the handle list in Process Explorer. Follow these steps to look at some details behind them:

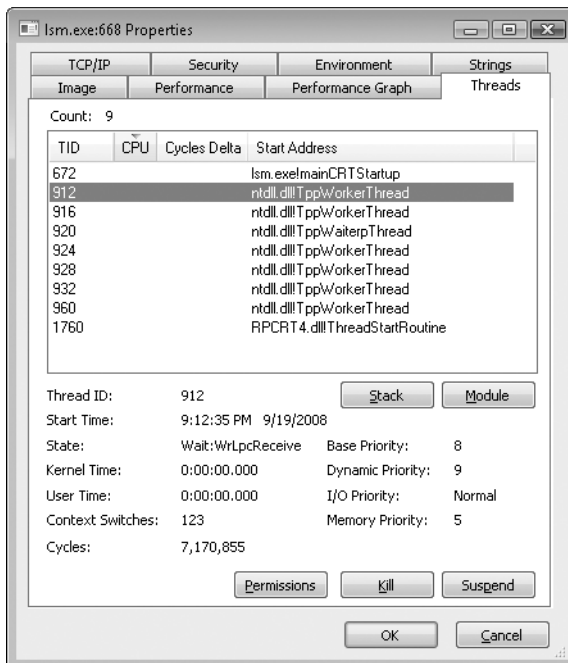
1. Run Process Explorer and select Show Unnamed Handles And Mappings from the View menu. Unfortunately, worker factories aren't named by Ntdll.dll, so you need to take this step in order to see the handles.
2. Select Lsm.exe from the list of processes, and look at the handle table. Make sure that the lower pane is shown (View, Show Lower Pane) and is displaying handle table mode (View, Lower Pane View, Handles).
3. Right-click on the lower pane columns, and then click on Select Columns. Make sure that the Type column is selected to be shown.

- Now scroll down the handles, looking at the Type column, until you find a handle of type *TpWorkerFactory*. You should see something like this:



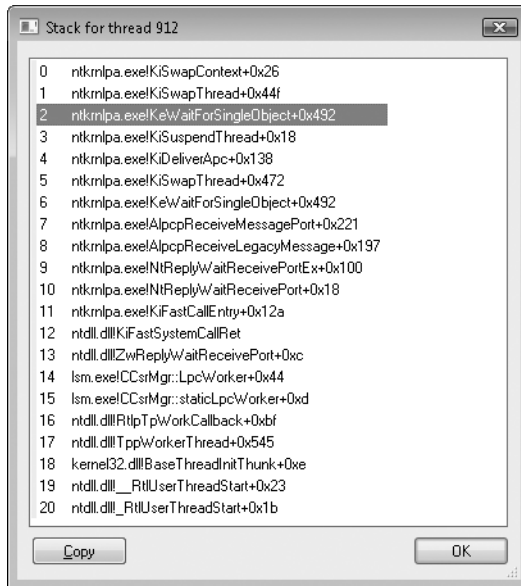
Notice how the *TpWorkerFactory* handle is immediately preceded by an *IoCompletion* handle. As was described previously, this occurs because before creating a worker factory, a handle to an I/O completion port on which work will be sent must be created.

- Now double-click `lsmsvc.exe` in the list of processes, and go to the Threads tab. You should see something similar to the image here:



On this system (with two processors), the worker factory has created six worker threads at the request of *lsm.exe* (processes can define a minimum and maximum number of threads) and based on its usage and the count of processors on the machine. These threads are identified as *TppWorkerThread*, which is *Ntdll.dll*'s worker entry point when calling the worker factory system calls.

6. *Ntdll.dll* is responsible for its own internal accounting inside the worker thread wrapper (*TppWorkerThread*) before calling the worker callback that the application has registered. By looking at the Wait reason in the State information for each thread, you can get a rough idea of what each worker thread may be doing. Double-click on one of the threads inside an LPC wait to look at its stack. Here's an example:



This specific worker thread is being used by *lsm.exe* for LPC communication. Because the local session manager needs to communicate with other components such as *Smss* and *Csrss* through LPC, it makes sense that it would want a number of its threads to be busy replying and waiting for LPC messages (the more threads doing this, the less stalling on the LPC pipeline).

If you look at other worker threads, you'll see some are waiting for objects such as events. A process can have multiple thread pools, and each thread pool can have a variety of threads doing completely unrelated tasks. It's up to the developer to assign work and to call the thread pool APIs to register this work through *Ntdll.dll*.

Thread Scheduling

This section describes the Windows scheduling policies and algorithms. The first subsection provides a condensed description of how scheduling works on Windows and a definition of key terms. Then Windows priority levels are described from both the Windows API and the Windows kernel points of view. After a review of the relevant Windows functions and Windows utilities and tools that relate to scheduling, the detailed data structures and algorithms that make up the Windows scheduling system are presented, with uniprocessor systems examined first and then multiprocessor systems.

Overview of Windows Scheduling

Windows implements a priority-driven, preemptive scheduling system—the highest-priority runnable (*ready*) thread always runs, with the caveat that the thread chosen to run might be limited by the processors on which the thread is allowed to run, a phenomenon called *processor affinity*. By default, threads can run on any available processor, but you can alter processor affinity by using one of the Windows scheduling functions listed in Table 5-15 (shown later in the chapter) or by setting an affinity mask in the image header.



EXPERIMENT: Viewing Ready Threads

You can view the list of ready threads with the kernel debugger *!ready* command. This command displays the thread or list of threads that are ready to run at each priority level. In the following example, generated on a 32-bit machine with a dual-core processor, five threads are ready to run at priority 8 on the first processor, and three threads at priority 10, two threads at priority 9, and six threads at priority 8 are ready to run on the second processor. Determining which of these threads get to run on their respective processor is a complex result at the end of several algorithms that the scheduler uses. We will cover this topic later in this section.

```
kd> !ready
Processor 0: Ready Threads at priority 8
  THREAD 857d9030 Cid 0ec8.0e30 Teb: 7ffdd000 Win32Thread: 00000000 READY
  THREAD 855c8300 Cid 0ec8.0eb0 Teb: 7ff9c000 Win32Thread: 00000000 READY
  THREAD 8576c030 Cid 0ec8.0c9c Teb: 7ffa8000 Win32Thread: 00000000 READY
  THREAD 85a8a7f0 Cid 0ec8.0d3c Teb: 7ff97000 Win32Thread: 00000000 READY
  THREAD 87d34488 Cid 0c48.04a0 Teb: 7ffde000 Win32Thread: 00000000 READY
Processor 1: Ready Threads at priority 10
  THREAD 857c0030 Cid 04c8.0378 Teb: 7ffdf000 Win32Thread: fef7f8c0 READY
  THREAD 856cc8e8 Cid 0e84.0a70 Teb: 7ffdb000 Win32Thread: f98fb4c0 READY
  THREAD 85c41c68 Cid 0e84.00ac Teb: 7ffde000 Win32Thread: ff460668 READY
Processor 1: Ready Threads at priority 9
  THREAD 87fc86f0 Cid 0ec8.04c0 Teb: 7ffd3000 Win32Thread: 00000000 READY
  THREAD 88696700 Cid 0ec8.0ce8 Teb: 7ffa0000 Win32Thread: 00000000 READY
```

```
Processor 1: Ready Threads at priority 8
THREAD 856e5520 Cid 0ec8.0228 Teb: 7ff98000 Win32Thread: 00000000 READY
THREAD 85609d78 Cid 0ec8.09b0 Teb: 7ffd9000 Win32Thread: 00000000 READY
THREAD 85fdeb78 Cid 0ec8.0218 Teb: 7ff72000 Win32Thread: 00000000 READY
THREAD 86086278 Cid 0ec8.0cc8 Teb: 7ff8d000 Win32Thread: 00000000 READY
THREAD 8816f7f0 Cid 0ec8.0b60 Teb: 7ffd5000 Win32Thread: 00000000 READY
THREAD 87710d78 Cid 0004.01b4 Teb: 00000000 Win32Thread: 00000000 READY
```

When a thread is selected to run, it runs for an amount of time called a *quantum*. A quantum is the length of time a thread is allowed to run before another thread at the same priority level (or higher, which can occur on a multiprocessor system) is given a turn to run. Quantum values can vary from system to system and process to process for any of three reasons: system configuration settings (long or short quanta), foreground/background status of the process, or use of the job object to alter the quantum. (Quantums are described in more detail in the “Quantum” section later in the chapter.) A thread might not get to complete its quantum, however. Because Windows implements a preemptive scheduler, if another thread with a higher priority becomes ready to run, the currently running thread might be preempted before finishing its time slice. In fact, a thread can be selected to run next and be preempted before even beginning its quantum!

The Windows scheduling code is implemented in the kernel. There’s no single “scheduler” module or routine, however—the code is spread throughout the kernel in which scheduling-related events occur. The routines that perform these duties are collectively called the kernel’s *dispatcher*. The following events might require thread dispatching:

- A thread becomes ready to execute—for example, a thread has been newly created or has just been released from the wait state.
- A thread leaves the running state because its time quantum ends, it terminates, it yields execution, or it enters a wait state.
- A thread’s priority changes, either because of a system service call or because Windows itself changes the priority value.
- A thread’s processor affinity changes so that it will no longer run on the processor on which it was running.

At each of these junctions, Windows must determine which thread should run next. When Windows selects a new thread to run, it performs a *context switch* to it. A context switch is the procedure of saving the volatile machine state associated with a running thread, loading another thread’s volatile state, and starting the new thread’s execution.

As already noted, Windows schedules at the thread granularity. This approach makes sense when you consider that processes don’t run but only provide resources and a context in

which their threads run. Because scheduling decisions are made strictly on a thread basis, no consideration is given to what process the thread belongs to. For example, if process *A* has 10 runnable threads, process *B* has 2 runnable threads, and all 12 threads are at the same priority, each thread would theoretically receive one-twelfth of the CPU time—Windows wouldn't give 50 percent of the CPU to process *A* and 50 percent to process *B*.

Priority Levels

To understand the thread-scheduling algorithms, you must first understand the priority levels that Windows uses. As illustrated in Figure 5-12, internally Windows uses 32 priority levels, ranging from 0 through 31. These values divide up as follows:

- Sixteen real-time levels (16 through 31)
- Fifteen variable levels (1 through 15)
- One system level (0), reserved for the zero page thread

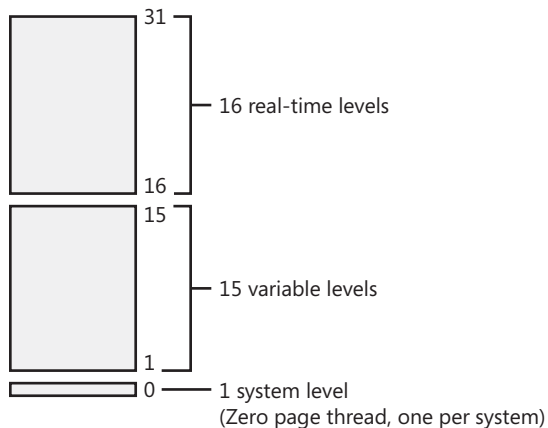


FIGURE 5-12 Thread priority levels

Thread priority levels are assigned from two different perspectives: those of the Windows API and those of the Windows kernel. The Windows API first organizes processes by the priority class to which they are assigned at creation (Real-time, High, Above Normal, Normal, Below Normal, and Idle) and then by the relative priority of the individual threads within those processes (Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle).

In the Windows API, each thread has a base priority that is a function of its process priority class and its relative thread priority. The mapping from Windows priority to internal Windows numeric priority is shown in Figure 5-13.

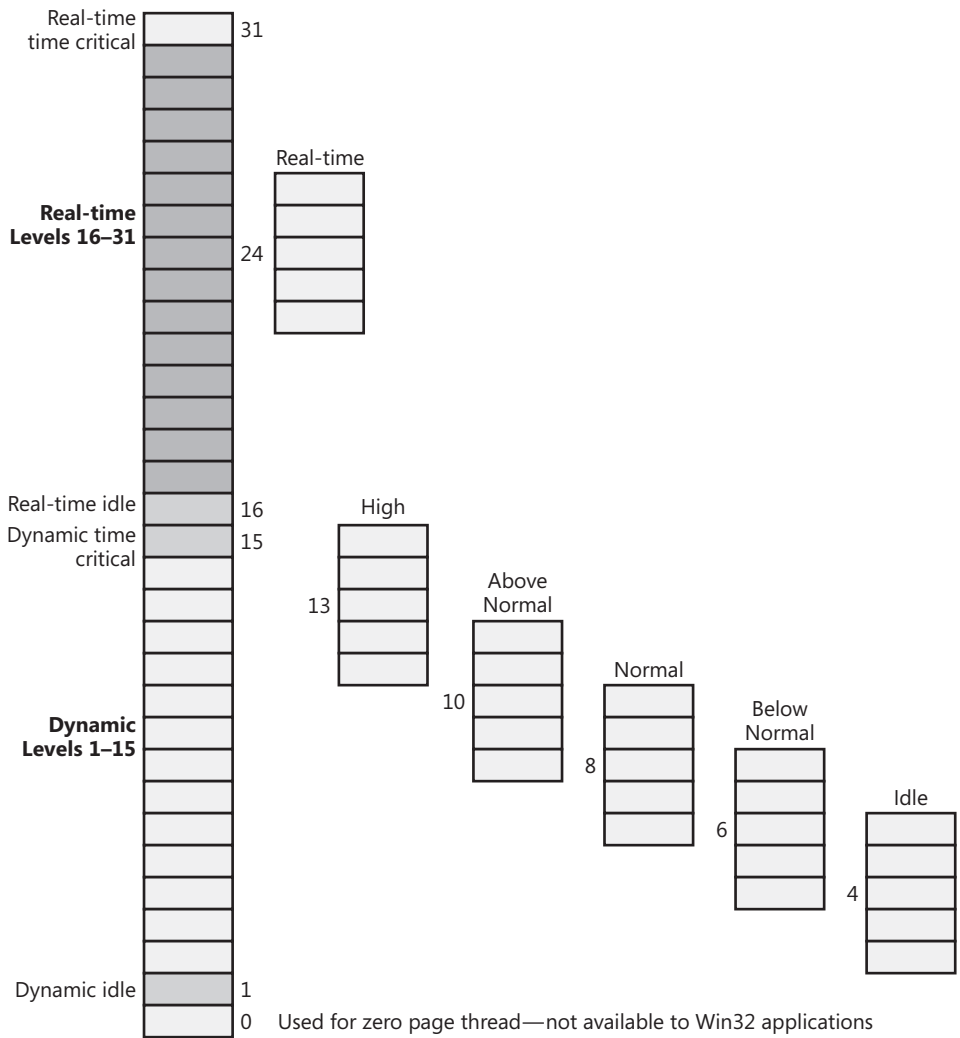


FIGURE 5-13 Mapping of Windows kernel priorities to the Windows API

Whereas a process has only a single base priority value, each thread has two priority values: current and base. Scheduling decisions are made based on the current priority. As explained in the following section on priority boosting, the system under certain circumstances increases the priority of threads in the dynamic range (1 through 15) for brief periods. Windows never adjusts the priority of threads in the real-time range (16 through 31), so they always have the same base and current priority.

A thread's initial base priority is inherited from the process base priority. A process, by default, inherits its base priority from the process that created it. This behavior can be overridden on the *CreateProcess* function or by using the command-line *start* command. A process priority can also be changed after being created by using the *SetPriorityClass* function

or various tools that expose that function, such as Task Manager and Process Explorer (by right-clicking on the process and choosing a new priority class). For example, you can lower the priority of a CPU-intensive process so that it does not interfere with normal system activities. Changing the priority of a process changes the thread priorities up or down, but their relative settings remain the same. It usually doesn't make sense, however, to change individual thread priorities within a process, because unless you wrote the program or have the source code, you don't really know what the individual threads are doing, and changing their relative importance might cause the program not to behave in the intended fashion.

Normally, the process base priority (and therefore the starting thread base priority) will default to the value at the middle of each process priority range (24, 13, 10, 8, 6, or 4). However, some Windows system processes (such as the Session Manager, service controller, and local security authentication server) have a base process priority slightly higher than the default for the Normal class (8). This higher default value ensures that the threads in these processes will all start at a higher priority than the default value of 8. These system processes use an internal system call (*NtSetInformationProcess*) to set their process base priority to a numeric value other than the normal default starting base priority.

Windows Scheduling APIs

The Windows API functions that relate to thread scheduling are listed in Table 5-15. (For more information, see the Windows API reference documentation.)

TABLE 5-15 Scheduling-Related APIs and Their Functions

API	Function
<i>Suspend/ResumeThread</i>	Suspends or resumes a paused thread from execution.
<i>Get/SetPriorityClass</i>	Returns or sets a process's priority class (base priority).
<i>Get/SetThreadPriority</i>	Returns or sets a thread's priority (relative to its process base priority).
<i>Get/SetProcessAffinityMask</i>	Returns or sets a process's affinity mask.
<i>SetThreadAffinityMask</i>	Sets a thread's affinity mask (must be a subset of the process's affinity mask) for a particular set of processors, restricting it to running on those processors.
<i>SetInformationJobObject</i>	Sets attributes for a job; some of the attributes affect scheduling, such as affinity and priority. (See the "Job Objects" section later in the chapter for a description of the job object.)
<i>GetLogicalProcessorInformation</i>	Returns details about processor hardware configuration (for hyperthreaded and NUMA systems).
<i>Get/SetThreadPriorityBoost</i>	Returns or sets the ability for Windows to boost the priority of a thread temporarily. (This ability applies only to threads in the dynamic range.)
<i>SetThreadIdealProcessor</i>	Establishes a preferred processor for a particular thread, but doesn't restrict the thread to that processor.

API	Function
<i>Get/SetProcessPriorityBoost</i>	Returns or sets the default priority boost control state of the current process. (This function is used to set the thread priority boost control state when a thread is created.)
<i>WaitForSingle/MultipleObject(s)</i>	Puts the current thread into a wait state until the specified object(s) is/are satisfied, or until the specified time interval (figured in milliseconds [msec]) expires, if given.
<i>SwitchToThread</i>	Yields execution to another thread (at priority 1 or higher) that is ready to run on the current processor.
<i>Sleep</i>	Puts the current thread into a wait state for a specified time interval (figured in milliseconds [msec]). A zero value relinquishes the rest of the thread's quantum.
<i>SleepEx</i>	Causes the current thread to go into a wait state until either an I/O completion callback is completed, an APC is queued to the thread, or the specified time interval ends.

Relevant Tools

You can change (and view) the base process priority with Task Manager and Process Explorer. You can kill individual threads in a process with Process Explorer (which should be done, of course, with extreme care).

You can view individual thread priorities with the Reliability and Performance Monitor, Process Explorer, or WinDbg. While it might be useful to increase or lower the priority of a process, it typically does not make sense to adjust individual thread priorities within a process because only a person who thoroughly understands the program (in other words, typically only the developer himself) would understand the relative importance of the threads within the process.

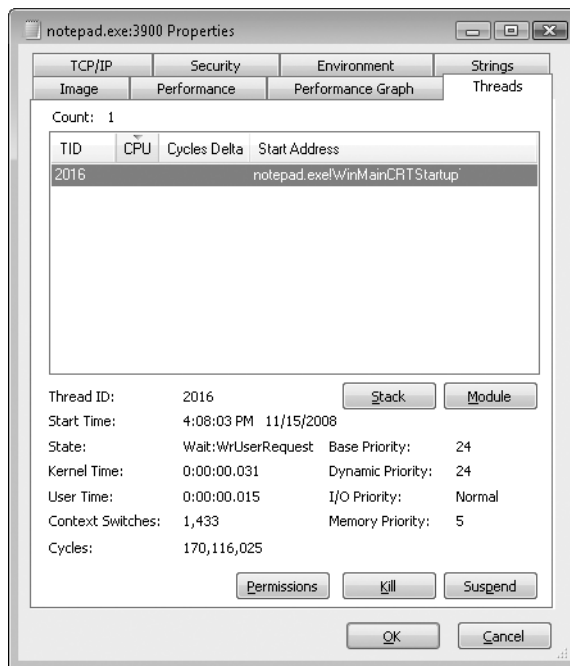
The only way to specify a starting priority class for a process is with the *start* command in the Windows command prompt. If you want to have a program start every time with a specific priority, you can define a shortcut to use the *start* command by beginning the command with **cmd /c**. This runs the command prompt, executes the command on the command line, and terminates the command prompt. For example, to run Notepad in the low-process priority, the shortcut would be **cmd /c start /low Notepad.exe**.



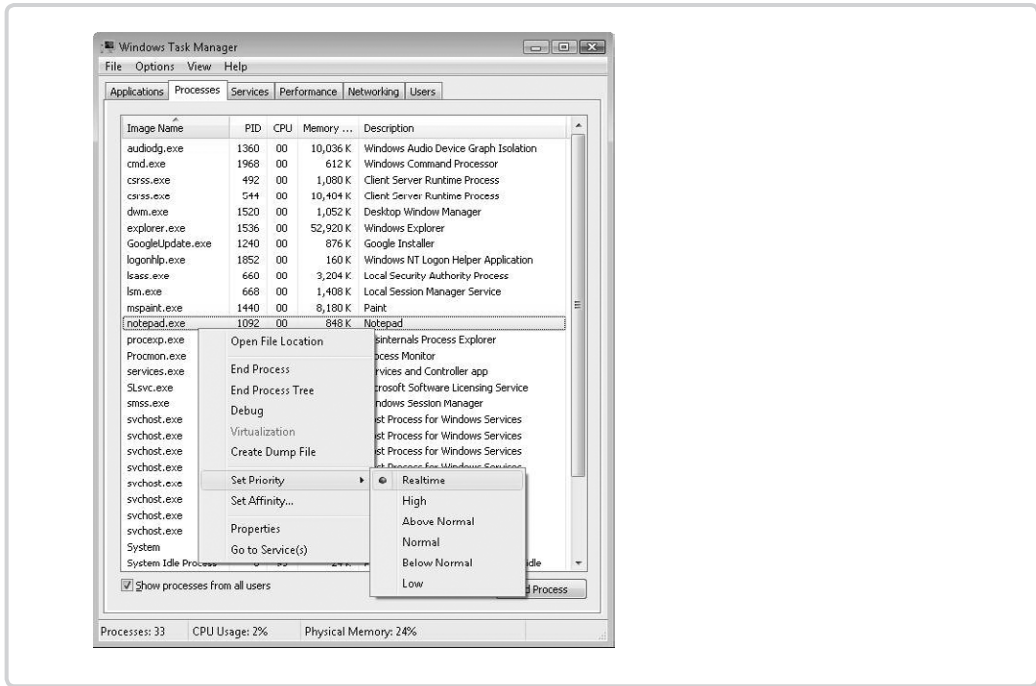
EXPERIMENT: Examining and Specifying Process and Thread Priorities

Try the following experiment:

1. From an elevated command prompt, type **start /realtime notepad**. Notepad should open.
2. Run Process Explorer and select Notepad.exe from the list of processes. Double-click on Notepad.exe to show the process properties window, and then click on the Threads tab, as shown here. Notice that the dynamic priority of the thread in Notepad is 24. This matches the real-time value shown in this image:



3. Task Manager can show you similar information. Press Ctrl+Shift+Esc to start Task Manager, and go to the Processes tab. Right-click on the Notepad.exe process, and select the Set Priority option. You can see that Notepad's process priority class is Realtime, as shown in the following dialog box.



Windows System Resource Manager

Windows Server 2008 Enterprise Edition and Windows Server 2008 Datacenter Edition include an optionally installable component called Windows System Resource Manager (WSRM). It permits the administrator to configure policies that specify CPU utilization, affinity settings, and memory limits (both physical and virtual) for processes. In addition, WSRM can generate resource utilization reports that can be used for accounting and verification of service-level agreements with users.

Policies can be applied for specific applications (by matching the name of the image with or without specific command-line arguments), users, or groups. The policies can be scheduled to take effect at certain periods or can be enabled all the time.

After you have set a resource-allocation policy to manage specific processes, the WSRM service monitors CPU consumption of managed processes and adjusts process base priorities when those processes do not meet their target CPU allocations.

The physical memory limitation uses the function `SetProcessWorkingSetSizeEx` to set a hard-working set maximum. The virtual memory limit is implemented by the service checking the private virtual memory consumed by the processes. (See Chapter 9 for an explanation of these memory limits.) If this limit is exceeded, WSRM can be configured to either kill the processes or write an entry to the Event Log. This behavior could be used to detect a process with a memory leak before it consumes all the available committed virtual memory on the system. Note that WSRM memory limits do not apply to Address Windowing Extensions (AWE) memory, large page memory, or kernel memory (nonpaged or paged pool).

Real-Time Priorities

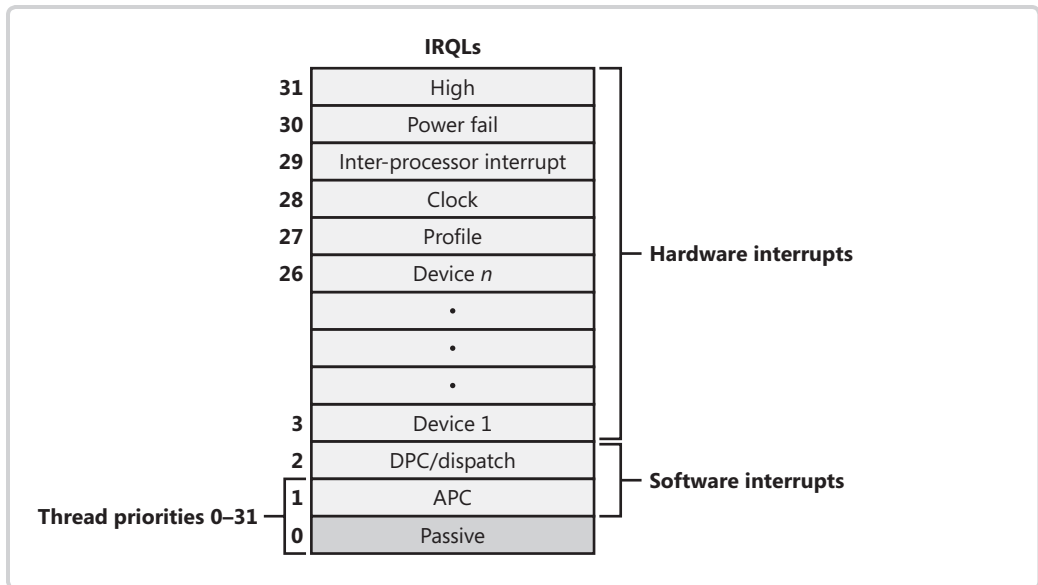
You can raise or lower thread priorities within the dynamic range in any application; however, you must have the *increase scheduling priority* privilege to enter the real-time range. Be aware that many important Windows kernel-mode system threads run in the real-time priority range, so if threads spend excessive time running in this range, they might block critical system functions (such as in the memory manager, cache manager, or other device drivers).



Note As illustrated in the following figure showing the x86 interrupt request levels (IRQLs), although Windows has a set of priorities called *real-time*, they are not real-time in the common definition of the term. This is because Windows doesn't provide true real-time operating system facilities, such as guaranteed interrupt latency or a way for threads to obtain a guaranteed execution time. For more information, see the sidebar "Windows and Real-Time Processing" in Chapter 3 as well as the MSDN Library article "Real-Time Systems and Microsoft Windows NT."

Interrupt Levels vs. Priority Levels

As illustrated in the following figure of the interrupt request levels (IRQLs) for a 32-bit system, threads normally run at IRQL 0 or 1. (For a description of how Windows uses interrupt levels, see Chapter 3.) User-mode code always runs at IRQL 0. Because of this, no user-mode thread, regardless of its priority, blocks hardware interrupts (although high-priority real-time threads can block the execution of important system threads). Only kernel-mode APCs execute at IRQL 1 because they interrupt the execution of a thread. (For more information on APCs, see Chapter 3.) Threads running in kernel mode can raise IRQL to higher levels, though—for example, while executing a system call that involves thread dispatching.



Thread States

Before you can comprehend the thread-scheduling algorithms, you need to understand the various execution states that a thread can be in. Figure 5-14 illustrates the state transitions for threads. (The numeric values shown represent the value of the thread state performance counter.) More details on what happens at each transition are included later in this section.

The thread states are as follows:

- **Ready** A thread in the ready state is waiting to execute. When looking for a thread to execute, the dispatcher considers only the pool of threads in the ready state.
- **Deferred ready** This state is used for threads that have been selected to run on a specific processor but have not yet been scheduled. This state exists so that the kernel can minimize the amount of time the systemwide lock on the scheduling database is held.
- **Standby** A thread in the standby state has been selected to run next on a particular processor. When the correct conditions exist, the dispatcher performs a context switch to this thread. Only one thread can be in the standby state for each processor on the system. Note that a thread can be preempted out of the standby state before it ever executes (if, for example, a higher priority thread becomes runnable before the standby thread begins execution).
- **Running** Once the dispatcher performs a context switch to a thread, the thread enters the running state and executes. The thread's execution continues until its quantum ends (and another thread at the same priority is ready to run), it is preempted by a higher priority thread, it terminates, it yields execution, or it voluntarily enters the wait state.

- **Waiting** A thread can enter the wait state in several ways: a thread can voluntarily wait for an object to synchronize its execution, the operating system can wait on the thread's behalf (such as to resolve a paging I/O), or an environment subsystem can direct the thread to suspend itself. When the thread's wait ends, depending on the priority, the thread either begins running immediately or is moved back to the ready state.
- **Gate Waiting** When a thread does a wait on a gate dispatcher object (see Chapter 3 for more information on gates), it enters the gate waiting state instead of the waiting state. This difference is important when breaking a thread's wait as the result of an APC. Because gates don't use the dispatcher lock, but a per-object lock, the kernel needs to perform some unique locking operations when breaking the wait of a thread waiting on a gate and a way to differentiate this from a normal wait.
- **Transition** A thread enters the transition state if it is ready for execution but its kernel stack is paged out of memory. Once its kernel stack is brought back into memory, the thread enters the ready state.
- **Terminated** When a thread finishes executing, it enters the terminated state. Once the thread is terminated, the executive thread block (the data structure in nonpaged pool that describes the thread) might or might not be deallocated. (The object manager sets policy regarding when to delete the object.)
- **Initialized** This state is used internally while a thread is being created.

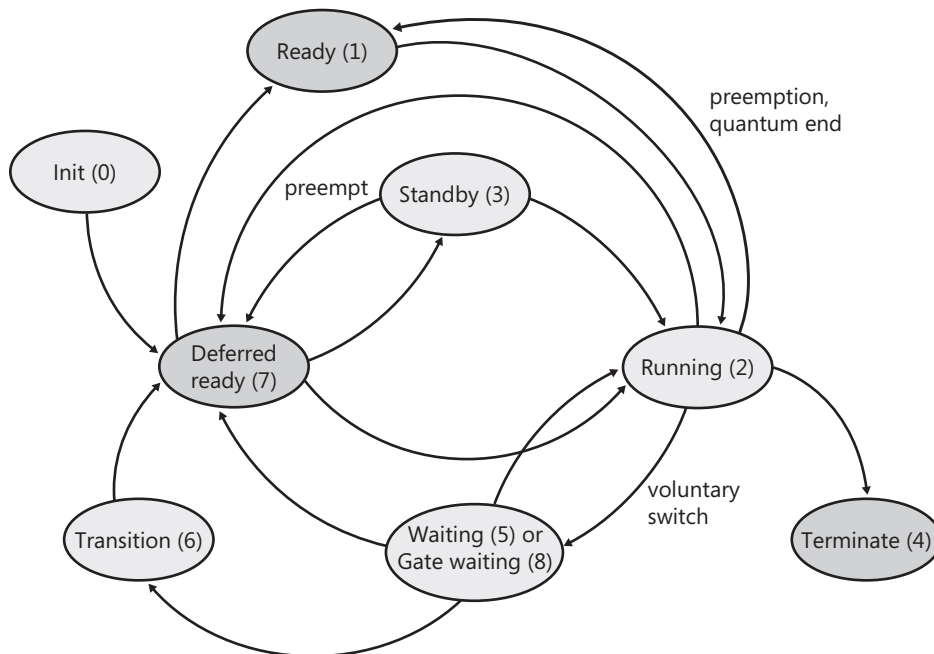


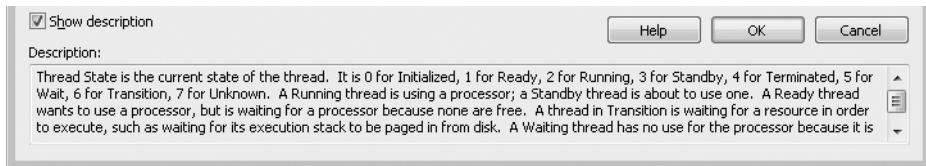
FIGURE 5-14 Thread states and transitions



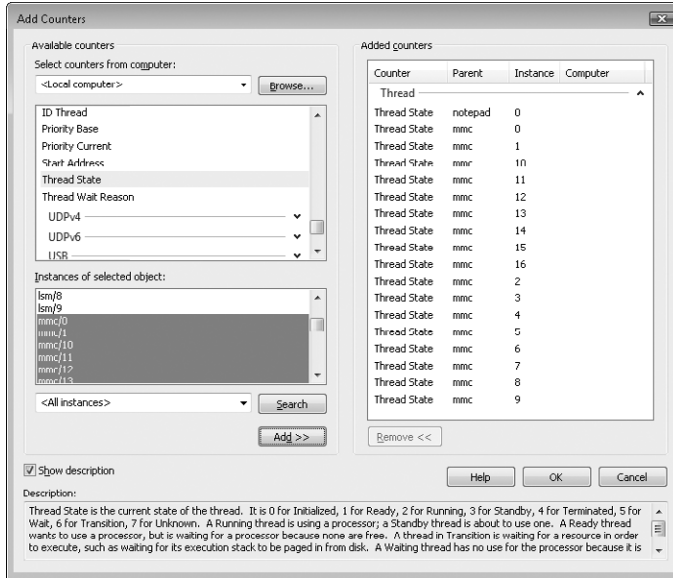
EXPERIMENT: Thread-Scheduling State Changes

You can watch thread-scheduling state changes with the Performance tool in Windows. This utility can be useful when you're debugging a multithreaded application and you're unsure about the state of the threads running in the process. To watch thread-scheduling state changes by using the Performance tool, follow these steps:

1. Run Notepad (Notepad.exe).
2. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability and Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
3. Select chart view if you're in some other view.
4. Right-click on the graph, and choose Properties.
5. Click the Graph tab, and change the chart vertical scale maximum to 7. (As you'll see from the explanation text for the performance counter, thread states are numbered from 0 through 7.) Click OK.
6. Click the Add button on the toolbar to bring up the Add Counters dialog box.
7. Select the Thread performance object, and then select the Thread State counter. Select the Show Description check box to see the definition of the values:

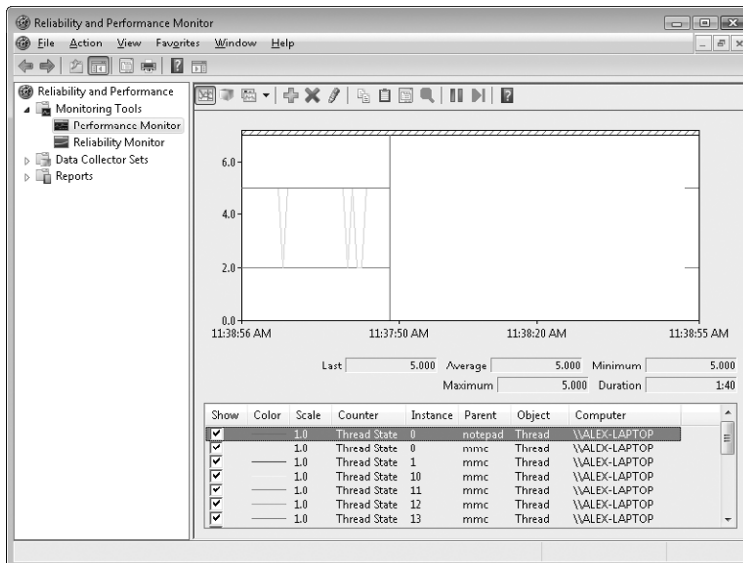


8. In the Instances box, select <All instances> and click Search. Scroll down until you see the Notepad process (notepad/0); select it, and click the Add button.
9. Scroll back up in the Instances box to the Mmc process (the Microsoft Management Console process running the System Monitor), select all the threads (mmc/0, mmc/1, and so on), and add them to the chart by clicking the Add button. Before you click Add, you should see something like the following dialog box.



10. Now close the Add Counters dialog box by clicking OK

11. You should see the state of the Notepad thread (the very top line in the following figure) as a 5, which, as shown in the explanation text you saw under step 7, represents the waiting state (because the thread is waiting for GUI input):



12. Notice that one thread in the Mmc process (running the Performance tool snap-in) is in the running state (number 2). This is the thread that's querying the thread states, so it's always displayed in the running state.
13. You'll never see Notepad in the running state (unless you're on a multiprocessor system) because Mmc is always in the running state when it gathers the state of the threads you're monitoring.

Dispatcher Database

To make thread-scheduling decisions, the kernel maintains a set of data structures known collectively as the *dispatcher database*, illustrated in Figure 5-15. The dispatcher database keeps track of which threads are waiting to execute and which processors are executing which threads.

To improve scalability, including thread-dispatching concurrency, Windows multiprocessor systems have per-processor dispatcher ready queues, as illustrated in Figure 5-15. In this way each CPU can check its own ready queues for the next thread to run without having to lock the systemwide ready queues. (Versions of Windows before Windows Server 2003 used a global database).

The per-processor ready queues, as well as the per-processor ready summary, are part of the processor control block (PRCB) structure. (To see the fields in the PRCB, type **dt nt!_prcb** in the kernel debugger.) The names of each component that we will talk about (in italics) are field members of the PRCB structure.

The dispatcher *ready queues* (*DispatcherReadyListHead*) contain the threads that are in the ready state, waiting to be scheduled for execution. There is one queue for each of the 32 priority levels. To speed up the selection of which thread to run or preempt, Windows maintains a 32-bit bit mask called the *ready summary* (*ReadySummary*). Each bit set indicates one or more threads in the ready queue for that priority level. (Bit 0 represents priority 0, and so on.)

Instead of scanning each ready list to see whether it is empty or not (which would make scheduling decisions dependent on the number of different priority threads), a single bit scan is performed as a native processor command to find the highest bit set. Regardless of the number of threads in the ready queue, this operation takes a constant amount of time, which is why you may sometimes see the Windows scheduling algorithm referred to as an O(1), or constant time, algorithm.

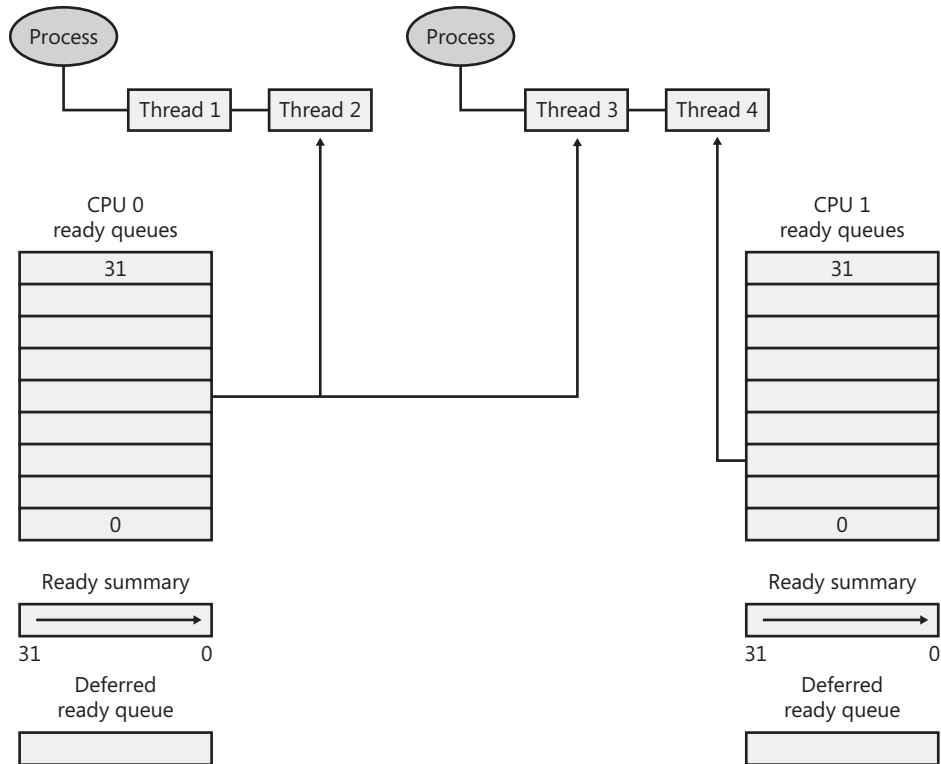


FIGURE 5-15 Windows multiprocessor dispatcher database

Table 5-16 lists the KPRCB fields involved in thread scheduling.

TABLE 5-16 Thread-Scheduling KPRCB Fields

KPRCB Field	Type	Description
<i>ReadySummary</i>	Bitmask (32 bits)	Bitmask of priority levels that have one or more ready threads
<i>DeferredReadyListHead</i>	Singly linked list	Single list head for the deferred ready queue
<i>DispatcherReadyListHead</i>	Array of 32 list entries	List heads for the 32 ready queues

The dispatcher database is synchronized by raising IRQL to SYNCH_LEVEL (which is defined as level 2). (For an explanation of interrupt priority levels, see the “Trap Dispatching” section in Chapter 3.) Raising IRQL in this way prevents other threads from interrupting thread dispatching on the processor because threads normally run at IRQL 0 or 1. However, on a multiprocessor system, more is required than just raising IRQL because other processors can simultaneously raise to the same IRQL and attempt to operate on the dispatcher database. How Windows synchronizes access to the dispatcher database is explained in the “Multiprocessor Systems” section later in the chapter.

Quantum

As mentioned earlier in the chapter, a quantum is the amount of time a thread gets to run before Windows checks to see whether another thread at the same priority is waiting to run. If a thread completes its quantum and there are no other threads at its priority, Windows permits the thread to run for another quantum.

On Windows Vista, threads run by default for 2 clock intervals; on Windows Server systems, by default, a thread runs for 12 clock intervals. (We'll explain how you can change these values later.) The rationale for the longer default value on server systems is to minimize context switching. By having a longer quantum, server applications that wake up as the result of a client request have a better chance of completing the request and going back into a wait state before their quantum ends.

The length of the clock interval varies according to the hardware platform. The frequency of the clock interrupts is up to the HAL, not the kernel. For example, the clock interval for most x86 uniprocessors is about 10 milliseconds, and for most x86 and x64 multiprocessors it is about 15 milliseconds. This clock interval is stored in the kernel variable *KeMaximumIncrement* as hundreds of nanoseconds.

Because of changes in thread run-time accounting in Windows Vista (briefly mentioned earlier in the thread activity experiment), although threads still run in units of clock intervals, the system does not use the count of clock ticks as the deciding factor for how long a thread has run and whether its quantum has expired. Instead, when the system starts up, a calculation is made whose result is the number of clock cycles that each quantum is equivalent to (this value is stored in the kernel variable *KiCyclesPerClockQuantum*). This calculation is made by multiplying the processor speed in Hz (CPU clock cycles per second) with the number of seconds it takes for one clock tick to fire (based on the *KeMaximumIncrement* value described above).

The end result of this new accounting method is that, as of Windows Vista, threads do not actually run for a quantum number based on clock ticks; they instead run for a *quantum target*, which represents an estimate of what the number of CPU clock cycles the thread has consumed should be when its turn would be given up. This target should be equal to an equivalent number of clock interval timer ticks because, as we've just seen, the calculation of clock cycles per quantum is based on the clock interval timer frequency, which you can check using the following experiment. On the other hand, because interrupt cycles are not charged to the thread, the actual clock time may be longer.



EXPERIMENT: Determining the Clock Interval Frequency

The Windows *GetSystemTimeAdjustment* function returns the clock interval. To determine the clock interval, download and run the Clockres program from Windows Sysinternals (www.microsoft.com/technet/sysinternals). Here's the output from a dual-core 32-bit Windows Vista system:

```
C:\>clockres

ClockRes - View the system clock resolution
By Mark Russinovich
SysInternals - www.sysinternals.com

The system clock interval is 15.600100 ms
```

Quantum Accounting

Each process has a quantum reset value in the kernel process block. This value is used when creating new threads inside the process and is duplicated in the kernel thread block, which is then used when giving a thread a new quantum target. The quantum reset value is stored in terms of actual quantum units (we'll discuss what these mean soon), which are then multiplied by the number of clock cycles per quantum, resulting in the quantum target.

As a thread runs, CPU clock cycles are charged at different events (context switches, interrupts, and certain scheduling decisions). If at a clock interval timer interrupt, the number of CPU clock cycles charged has reached (or passed) the quantum target, then quantum end processing is triggered. If there is another thread at the same priority waiting to run, a context switch occurs to the next thread in the ready queue.

Internally, a quantum unit is represented as one third of a clock tick (so one clock tick equals three quanta). This means that on Windows Vista systems, threads, by default, have a quantum reset value of 6 ($2 * 3$), and that Windows Server 2008 systems have a quantum reset value of 36 ($12 * 3$). For this reason, the *KiCyclesPerClockQuantum* value is divided by three at the end of the calculation previously described, since the original value would describe only CPU clock cycles per clock interval timer tick.

The reason a quantum was stored internally as a fraction of a clock tick rather than as an entire tick was to allow for partial quantum decay on wait completion on versions of Windows prior to Windows Vista. Prior versions used the clock interval timer for quantum expiration. If this adjustment were not made, it would have been possible for threads never to have their quanta reduced. For example, if a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired, it would never have its quantum charged for the time it was running. Because threads now have CPU clock cycles charged instead of quanta, and because this no longer depends on the clock interval timer, these adjustments are not required.



EXPERIMENT: Determining the Clock Cycles per Quantum

Windows doesn't expose the number of clock cycles per quantum through any function, but with the calculation and description we've given, you should be able to determine this on your own using the following steps and a kernel debugger such as WinDbg in local debugging mode.

1. Obtain your processor frequency as Windows has detected it. You can use the value stored in the PRCB's MHz field, which can be displayed with the `!cpuinfo` command. Here is a sample output of a dual-core Intel system running at 2829 MHz.

```

1kd> !cpuinfo
CP F/M/S Manufacturer MHz PRCB Signature MSR 8B Signature Features
 0 6,15,6 GenuineIntel 2829 000000c700000000 >000000c700000000<a00f3fff
 1 6,15,6 GenuineIntel 2829 000000c700000000 a00f3fff
          Cached Update Signature 000000c700000000
          Initial Update Signature 000000c700000000

```

2. Convert the number to Hertz (Hz). This is the number of CPU clock cycles that occur each second on your system. In this case, 2,829,000,000 cycles per second.
3. Obtain the clock interval on your system by using `clockres`. This measures how long it takes before the clock fires. On the sample system used here, this interval was 15.600100 ms.
4. Convert this number to the number of times the clock interval timer fires each second. One second is 1000 ms, so divide the number derived in step 3 by 1000. In this case, the timer fires every 0.0156001 second.
5. Multiply this count by the number of cycles each second that you obtained in step 2. In our case, 44,132,682.9 cycles have elapsed after each clock interval.
6. Remember that each quantum unit is one-third of a clock interval, so divide the number of cycles by three. In our example, this gives us 14,710,894, or 0xE0786E in hexadecimal. This is the number of clock cycles each quantum unit should take on a system running at 2829 MHz with a clock interval of around 15 ms.
7. To verify your calculation, dump the value of `KiCyclesPerClockQuantum` on your system—it should match.

```

1kd> dd nt!KiCyclesPerClockQuantum 11
81d31ae8 00e0786e

```

Controlling the Quantum

You can change the thread quantum for all processes, but you can choose only one of two settings: short (2 clock ticks, the default for client machines) or long (12 clock ticks, the default for server systems).



Note By using the job object on a system running with long quantum values, you can select other quantum values for the processes in the job. For more information on the job object, see the “Job Objects” section later in the chapter.

To change this setting, right-click on your computer name’s icon on the desktop, choose Properties, click the Advanced System Settings label, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. The dialog box displayed is shown in Figure 5-16.

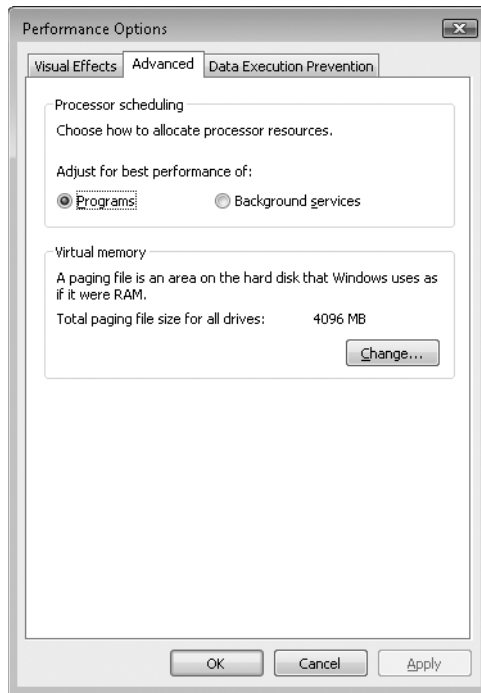


FIGURE 5-16 Quantum configuration in the Performance Options dialog box

The Programs setting designates the use of short, variable quantum values—the default for Windows Vista. If you install Terminal Services on Windows Server 2008 systems and configure the server as an application server, this setting is selected so that the users on the terminal server will have the same quantum settings that would normally be set on a desktop or client system. You might also select this manually if you were running Windows Server as your desktop operating system.

The Background Services option designates the use of long, fixed quantum values—the default for Windows Server 2008 systems. The only reason you might select this option on a workstation system is if you were using the workstation as a server system.

One additional difference between the Programs and Background Services settings is the effect they have on the quantum of the threads in the foreground process. This is explained in the next section.

Quantum Boosting

When a window is brought into the foreground on a client system, all the threads in the process containing the thread that owns the foreground window have their quanta tripled. Thus, threads in the foreground process run with a quantum of 6 clock ticks, whereas threads in other processes have the default client quantum of 2 clock ticks. In this way, when you switch away from a CPU-intensive process, the new foreground process will get proportionally more of the CPU, because when its threads run they will have a longer turn than background threads (again, assuming the thread priorities are the same in both the foreground and background processes).

Note that this adjustment of quanta applies only to processes with a priority higher than Idle on systems configured to Programs in the Performance Options settings described in the previous section. Thread quanta are not changed for the foreground process on systems configured to Background Services (the default on Windows Server 2008 systems).

Quantum Settings Registry Value

The user interface to control quantum settings described earlier modifies the registry value HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation. In addition to specifying the relative length of thread quanta (short or long), this registry value also defines whether or not threads in the foreground process should have their quanta boosted (and if so, the amount of the boost). This value consists of 6 bits divided into the three 2-bit fields shown in Figure 5-17.

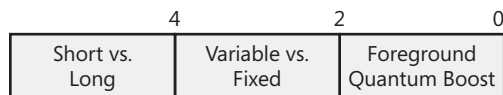


FIGURE 5-17 Fields of the Win32PrioritySeparation registry value

The fields shown in Figure 5-17 can be defined as follows:

- **Short vs. Long** A setting of 1 specifies long, and 2 specifies short. A setting of 0 or 3 indicates that the default will be used (short for Windows Vista, long for Windows Server 2008 systems).
- **Variable vs. Fixed** A setting of 1 means to vary the quantum for the foreground process, and 2 means that quantum values don't change for foreground processes. A setting of 0 or 3 means that the default (which is variable for Windows Vista and fixed for Windows Server 2008 systems) will be used.

- **Foreground Quantum Boost** This field (stored in the kernel variable *PsPrioritySeperation*) must have a value of 0, 1, or 2. (A setting of 3 is invalid and treated as 2.) It is used as an index into a three-element byte array named *PspForegroundQuantum* to obtain the quantum for the threads in the foreground process. The quantum for threads in background processes is taken from the first entry in this quantum table. Table 5-17 shows the possible settings for *PspForegroundQuantum*.

TABLE 5-17 Quantum Values

		Short			Long		
Variable	6	12	18	12	24	36	
Fixed	18	18	18	36	36	36	

Note that when you're using the Performance Options dialog box described earlier, you can choose from only two combinations: short quanta with foreground quanta tripled, or long quanta with no quantum changes for foreground threads. However, you can select other combinations by modifying the *Win32PrioritySeparation* registry value directly.



EXPERIMENT: Effects of Changing the Quantum Configuration

Using a local debugger (Kd or WinDbg), you can see how the two quantum configuration settings, *Programs and Background Services*, affect the *PsPrioritySeperation* and *PspForegroundQuantum* tables, as well as modify the *QuantumReset* value of threads on the system. Take the following steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. Select the Programs option and click Apply. Keep this window open for the duration of the experiment.
2. Dump the values of *PsPrioritySeperation* (this is a deliberate misspelling inside the Windows kernel, not an error in this book) and *PspForegroundQuantum*, as shown here. The values shown are what you should see on a Windows Vista system after making the change in step 1. Notice how the variable, short quantum table is being used, and that a priority boost of 2 will apply to foreground applications.

```

1kd> dd PsPrioritySeperation 11
81d3101c 00000002
1kd> db PspForegroundQuantum 13
81d0946c 06 0c 12
...
```


- Now take a look at the *QuantumReset* value of any process on the system. As described earlier, this is the default, full quantum of each thread on the system when it is replenished. This value is cached into each thread of the process, but the *KPROCESS* structure is easier to look at. Notice in this case it is 6, since WinDbg, like most other applications, gets the quantum set in the first entry of the *PspForegroundQuantum* table.

```

1kd> .process
Implicit process is now 85b32d90
1kd> dt _KPROCESS 85b32d90
nt!_KPROCESS
+0x000 Header           : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY [ 0x85b32da0 - 0x85b32da0 ]
+0x018 DirectoryTableBase : 0xb45b0880
+0x01c Unused0         : 0
+0x020 LdtDescriptor   : _KGDTENTRY
+0x028 Int21Descriptor : _KIDTENTRY
+0x030 IopmOffset      : 0x20ac
+0x032 Iop1            : 0 ''
+0x033 Unused          : 0 ''
+0x034 ActiveProcessors : 1
+0x038 Kerne1Time      : 0
+0x03c UserTime        : 0
+0x040 ReadyListHead  : _LIST_ENTRY [ 0x85b32dd0 - 0x85b32dd0 ]
+0x048 SwapListEntry  : _SINGLE_LIST_ENTRY
+0x04c VdmTrapHandler : (null)
+0x050 ThreadListHead : _LIST_ENTRY [ 0x861e7e0c - 0x8620637c ]
+0x058 ProcessLock    : 0
+0x05c Affinity        : 3
+0x060 AutoAlignment  : 0y0
+0x060 DisableBoost   : 0y0
+0x060 DisableQuantum : 0y0
+0x060 ReservedFlags  : 0y00000000000000000000000000000000 (0)
+0x060 ProcessFlags   : 0
+0x064 BasePriority    : 8 ''
+0x065 QuantumReset   : 6 ''

```

- Now change the Performance option to Background Services in the dialog box you opened in step 1.
- Repeat the commands shown in steps 2 and 3. You should see the values change in a manner consistent with our discussion in this section:

```

1kd> dd PsPrioritySeperation L1
81d3101c 00000000
1kd> db PspForegroundQuantum 1 3
81d0946c 24 24 24 $$$
1kd> dt _KPROCESS 85b32d90
nt!_KPROCESS
+0x000 Header           : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY [ 0x85b32da0 - 0x85b32da0 ]
+0x018 DirectoryTableBase : 0xb45b0880
+0x01c Unused0         : 0

```

```

+0x020 LdtDescriptor      : _KGDTENTRY
+0x028 Int21Descriptor    : _KIDTENTRY
+0x030 IopmOffset        : 0x20ac
+0x032 Iop1               : 0 ''
+0x033 Unused             : 0 ''
+0x034 ActiveProcessors  : 1
+0x038 KernelTime        : 0
+0x03c UserTime           : 0
+0x040 ReadyListHead     : _LIST_ENTRY [ 0x85b32dd0 - 0x85b32dd0 ]
+0x048 SwapListEntry     : _SINGLE_LIST_ENTRY
+0x04c VdmTrapHandler    : (null)
+0x050 ThreadListHead    : _LIST_ENTRY [ 0x861e7e0c - 0x860c14f4 ]
+0x058 ProcessLock       : 0
+0x05c Affinity           : 3
+0x060 AutoAlignment     : 0y0
+0x060 DisableBoost      : 0y0
+0x060 DisableQuantum    : 0y0
+0x060 ReservedFlags     : 0y00000000000000000000000000000000 (0)
+0x060 ProcessFlags      : 0
+0x064 BasePriority       : 8 ''
+0x065 QuantumReset      : 36 '$'

```

Scheduling Scenarios

Windows bases the question of “Who gets the CPU?” on thread priority; but how does this approach work in practice? The following sections illustrate just how priority-driven preemptive multitasking works on the thread level.

Voluntary Switch

First a thread might voluntarily relinquish use of the processor by entering a wait state on some object (such as an event, a mutex, a semaphore, an I/O completion port, a process, a thread, a window message, and so on) by calling one of the Windows wait functions (such as *WaitForSingleObject* or *WaitForMultipleObjects*). Waiting for objects is described in more detail in Chapter 3.

Figure 5-18 illustrates a thread entering a wait state and Windows selecting a new thread to run.

In Figure 5-18, the top block (thread) is voluntarily relinquishing the processor so that the next thread in the ready queue can run (as represented by the halo it has when in the Running column). Although it might appear from this figure that the relinquishing thread’s priority is being reduced, it’s not—it’s just being moved to the wait queue of the objects the thread is waiting for.

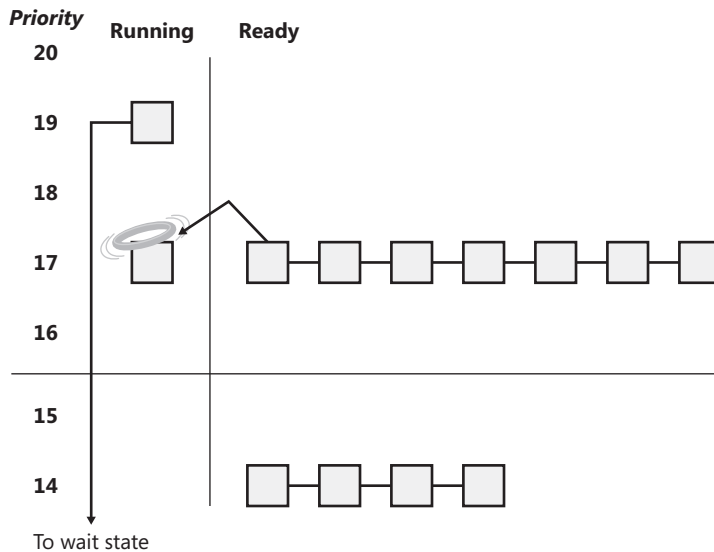


FIGURE 5-18 Voluntary switching

Preemption

In this scheduling scenario, a lower-priority thread is preempted when a higher-priority thread becomes ready to run. This situation might occur for a couple of reasons:

- A higher-priority thread's wait completes. (The event that the other thread was waiting for has occurred.)
- A thread priority is increased or decreased.

In either of these cases, Windows must determine whether the currently running thread should still continue to run or whether it should be preempted to allow a higher-priority thread to run.



Note Threads running in user mode can preempt threads running in kernel mode—the mode in which the thread is running doesn't matter. The thread priority is the determining factor.

When a thread is preempted, it is put at the head of the ready queue for the priority it was running at. Figure 5-19 illustrates this situation.

In Figure 5-19, a thread with priority 18 emerges from a wait state and repossesses the CPU, causing the thread that had been running (at priority 16) to be bumped to the head of the ready queue. Notice that the bumped thread isn't going to the end of the queue but to the beginning; when the preempting thread has finished running, the bumped thread can complete its quantum.

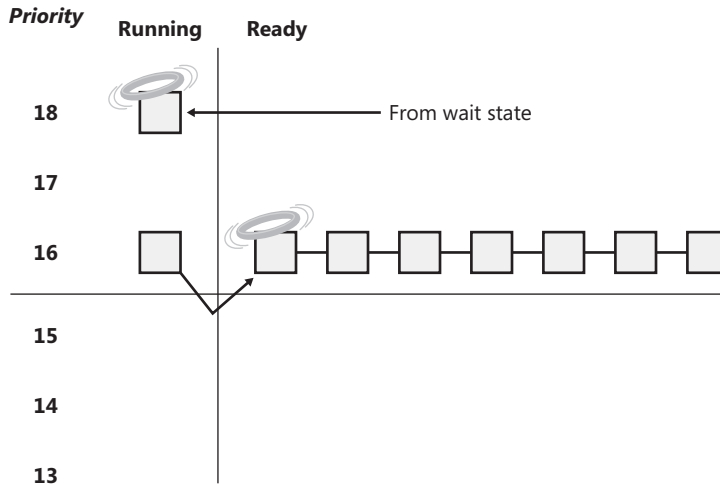


FIGURE 5-19 Preemptive thread scheduling

Quantum End

When the running thread exhausts its CPU quantum, Windows must determine whether the thread's priority should be decremented and then whether another thread should be scheduled on the processor.

If the thread priority is reduced, Windows looks for a more appropriate thread to schedule. (For example, a more appropriate thread would be a thread in a ready queue with a higher priority than the new priority for the currently running thread.) If the thread priority isn't reduced and there are other threads in the ready queue at the same priority level, Windows selects the next thread in the ready queue at that same priority level and moves the previously running thread to the tail of that queue (giving it a new quantum value and changing its state from running to ready). This case is illustrated in Figure 5-20. If no other thread of the same priority is ready to run, the thread gets to run for another quantum.

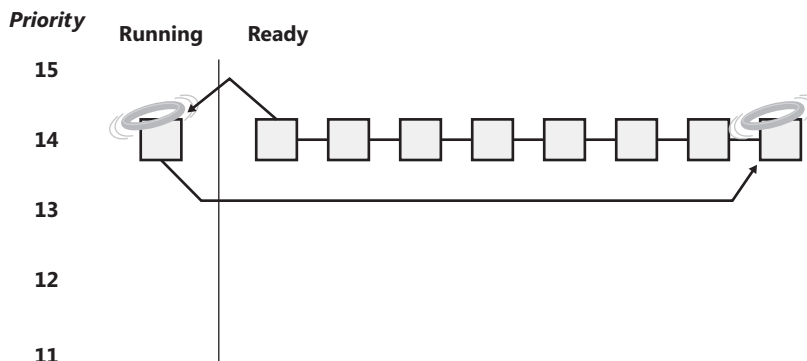


FIGURE 5-20 Quantum end thread scheduling

As we've seen, instead of simply relying on a clock interval timer–based quantum to schedule threads, Windows uses an accurate CPU clock cycle count to maintain quantum targets. One factor we haven't yet mentioned is that Windows also uses this count to determine whether quantum end is currently appropriate for the thread—something that may have happened previously and is important to discuss.

Under the scheduling model prior to Windows Vista, which relied only on the clock interval timer, the following situation could occur:

- Threads *A* and *B* become ready to run during the middle of an interval (scheduling code runs not just at each clock interval, so this is often the case).
- Thread *A* starts running but is interrupted for a while. The time spent handling the interrupt is charged to the thread.
- Interrupt processing finishes, thread *A* starts running again, but it quickly hits the next clock interval. The scheduler can only assume that thread *A* had been running all this time and now switches to thread *B*.
- Thread *B* starts running and has a chance to run for a full clock interval (barring pre-emption or interrupt handling).

In this scenario, thread *A* was unfairly penalized in two different ways. First of all, the time that it had to spend handling a device interrupt was accounted to its own CPU time, even though the thread had probably nothing to do with the interrupt. (Recall that interrupts are handled in the context of whichever thread had been running at the time.) It was also unfairly penalized for the time the system was idling inside that clock interval before it was scheduled.

Figure 5-21 represents this scenario.

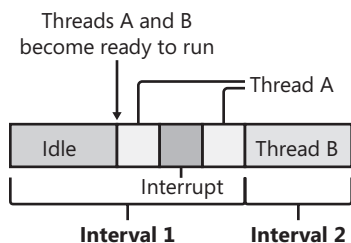


FIGURE 5-21 Unfair time slicing in previous versions of Windows

Because Windows keeps an accurate count of the exact number of CPU clock cycles spent doing work that the thread was scheduled to do (which means excluding interrupts), and because it keeps a quantum target of clock cycles that should have been spent by the thread at the end of its quantum, both of the unfair decisions that would have been made against thread *A* will not happen in Windows.

Instead, the following situation will occur:

- Threads *A* and *B* become ready to run during the middle of an interval.
- Thread *A* starts running but is interrupted for a while. The CPU clock cycles spent handling the interrupt are not charged to the thread.
- Interrupt processing finishes, thread *A* starts running again, but it quickly hits the next clock interval. The scheduler looks at the number of CPU clock cycles that have been charged to the thread and compares them to the expected CPU clock cycles that should have been charged at quantum end.
- Because the former number is much smaller than it should be, the scheduler assumes that thread *A* started running in the middle of a clock interval and may have additionally been interrupted.
- Thread *A* gets its quantum increased by another clock interval, and the quantum target is recalculated. Thread *A* now has its chance to run for a full clock interval.
- At the next clock interval, thread *A* has finished its quantum, and thread *B* now gets a chance to run.

Figure 5-22 represents this scenario.

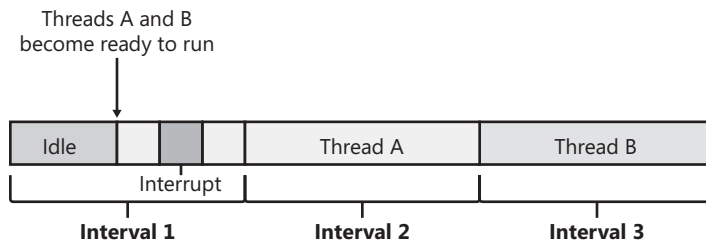


FIGURE 5-22 Fair time slicing in current versions of Windows

Termination

When a thread finishes running (either because it returned from its main routine, called *ExitThread*, or was killed with *TerminateThread*), it moves from the running state to the terminated state. If there are no handles open on the thread object, the thread is removed from the process thread list and the associated data structures are deallocated and released.

Context Switching

A thread's context and the procedure for context switching vary depending on the processor's architecture. A typical context switch requires saving and reloading the following data:

- Instruction pointer
- Kernel stack pointer
- A pointer to the address space in which the thread runs (the process's page table directory)

The kernel saves this information from the old thread by pushing it onto the current (old thread's) kernel-mode stack, updating the stack pointer, and saving the stack pointer in the old thread's KTHREAD block. The kernel stack pointer is then set to the new thread's kernel stack, and the new thread's context is loaded. If the new thread is in a different process, it loads the address of its page table directory into a special processor register so that its address space is available. (See the description of address translation in Chapter 9.) If a kernel APC that needs to be delivered is pending, an interrupt at IRQL 1 is requested. Otherwise, control passes to the new thread's restored instruction pointer and the new thread resumes execution.

Idle Thread

When no runnable thread exists on a CPU, Windows dispatches the per-CPU idle thread. Each CPU is allotted one idle thread because on a multiprocessor system one CPU can be executing a thread while other CPUs might have no threads to execute.

Various Windows process viewer utilities report the idle process using different names. Task Manager and Process Explorer call it "System Idle Process," while Tlist calls it "System Process." If you look at the EPROCESS structure's *ImageFileName* member, you'll see the internal name for the process is "Idle." Windows reports the priority of the idle thread as 0 (15 on x64 systems). In reality, however, the idle threads don't have a priority level because they run only when there are no real threads to run—they are not scheduled and never part of any ready queues. (Remember, only one thread per Windows system is actually running at priority 0—the zero page thread, explained in Chapter 9.)

Apart from priority, there are many other fields in the idle process or its threads that may be reported as 0. This occurs because the idle process is not an actual full-blown object manager process object, and neither are its idle threads. Instead, the initial idle thread and idle process objects are statically allocated and used to bootstrap the system before the process manager initializes. Subsequent idle thread structures are allocated dynamically as additional processors are brought online. Once process management initializes, it uses the special variable *PsystemProcess* to refer to the idle process.

Apart from some critical fields provided so that these threads and their process can have a PID and name, everything else is ignored, which means that query APIs may simply return zeroed data.

The idle loop runs at DPC/dispatch level, polling for work to do, such as delivering deferred procedure calls (DPCs) or looking for threads to dispatch to. Although some details of the flow vary between architectures, the basic flow of control of the idle thread is as follows:

1. Enables and disables interrupts (allowing any pending interrupts to be delivered).
2. Checks whether any DPCs (described in Chapter 3) are pending on the processor. If DPCs are pending, clears the pending software interrupt and delivers them. (This will also perform timer expiration, as well as deferred ready processing. The latter is explained in the upcoming multiprocessor scheduling section.)
3. Checks whether a thread has been selected to run next on the processor, and if so, dispatches that thread.
4. Calls the registered power management processor idle routine (in case any power management functions need to be performed), which is either in the processor power driver (such as `intelppm.sys`) or in the HAL if such a driver is unavailable.
5. On debug systems, checks if there is a kernel debugger trying to break into the system and gives it access.
6. If requested, checks for threads waiting to run on other processors and schedules them locally. (This operation is also explained in the upcoming multiprocessor scheduling section.)

Priority Boosts

In six cases, the Windows scheduler can boost (increase) the current priority value of threads:

- On completion of I/O operations
- After waiting for executive events or semaphores
- When a thread has been waiting on an executive resource for too long
- After threads in the foreground process complete a wait operation
- When GUI threads wake up because of windowing activity
- When a thread that's ready to run hasn't been running for some time (CPU starvation)

The intent of these adjustments is to improve overall system throughput and responsiveness as well as resolve potentially unfair scheduling scenarios. Like any scheduling algorithms, however, these adjustments aren't perfect, and they might not benefit all applications.



Note Windows never boosts the priority of threads in the real-time range (16 through 31). Therefore, scheduling is always predictable with respect to other threads in the real-time range. Windows assumes that if you're using the real-time thread priorities, you know what you're doing.

Windows Vista adds one more scenario in which a priority boost can occur, multimedia playback. Unlike the other priority boosts, which are applied directly by kernel code, multimedia playback boosts are managed by a user-mode service called the MultiMedia Class Scheduler Service (MMCSS). (Although the boosts are still done *in* kernel mode, the request to boost the threads is managed by this user-mode service.) We'll first cover the typical kernel-managed priority boosts and then talk about MMCSS and the kind of boosting it performs.

Priority Boosting after I/O Completion

Windows gives temporary priority boosts upon completion of certain I/O operations so that threads that were waiting for an I/O will have more of a chance to run right away and process whatever was being waited for. Recall that 1 quantum unit is deducted from the thread's remaining quantum when it wakes up so that I/O bound threads aren't unfairly favored. Although you'll find recommended boost values in the Windows Driver Kit (WDK) header files (by searching for "#define IO" in Wdm.h or Ntddk.h), the actual value for the boost is up to the device driver. (These values are listed in Table 5-18.) It is the device driver that specifies the boost when it completes an I/O request on its call to the kernel function *IoCompleteRequest*. In Table 5-18, notice that I/O requests to devices that warrant better responsiveness have higher boost values.

TABLE 5-18 Recommended Boost Values

Device	Boost
Disk, CD-ROM, parallel, video	1
Network, mailslot, named pipe, serial	2
Keyboard, mouse	6
Sound	8

The boost is always applied to a thread's current priority, not its base priority. As illustrated in Figure 5-23, after the boost is applied, the thread gets to run for one quantum at the elevated priority level. After the thread has completed its quantum, it decays one priority level and then runs another quantum. This cycle continues until the thread's priority level has decayed back to its base priority. A thread with a higher priority can still preempt the boosted thread, but the interrupted thread gets to finish its time slice at the boosted priority level before it decays to the next lower priority.

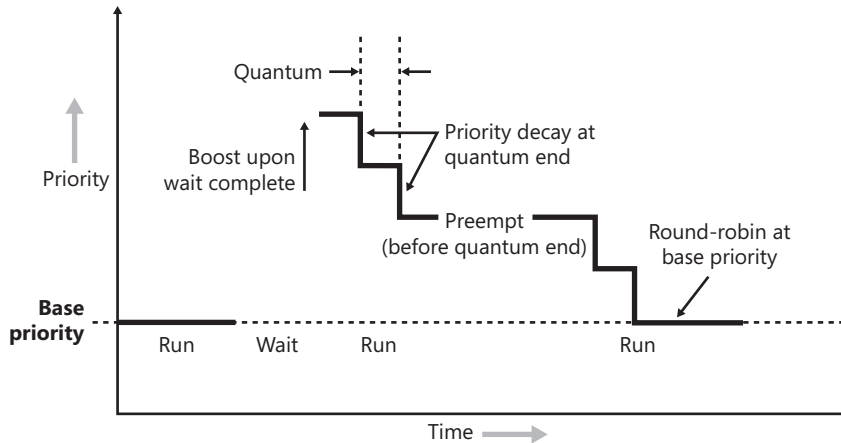


FIGURE 5-23 Priority boosting and decay

As noted earlier, these boosts apply only to threads in the dynamic priority range (0 through 15). No matter how large the boost is, the thread will never be boosted beyond level 15 into the real-time priority range. In other words, a priority 14 thread that receives a boost of 5 will go up to priority 15. A priority 15 thread that receives a boost will remain at priority 15.

Boosts After Waiting for Events and Semaphores

When a thread that was waiting for an executive event or a semaphore object has its wait satisfied (because of a call to the function *SetEvent*, *PulseEvent*, or *ReleaseSemaphore*), it receives a boost of 1. (See the value for `EVENT_INCREMENT` and `SEMAPHORE_INCREMENT` in the WDK header files.) Threads that wait for events and semaphores warrant a boost for the same reason that threads that wait for I/O operations do—threads that block on events are requesting CPU cycles less frequently than CPU-bound threads. This adjustment helps balance the scales.

This boost operates the same as the boost that occurs after I/O completion, as described in the previous section:

- The boost is always applied to the base priority (not the current priority).
- The priority will never be boosted above 15.
- The thread gets to run at the elevated priority for its remaining quantum (as described earlier, quanta are reduced by 1 when threads exit a wait) before decaying one priority level at a time until it reaches its original base priority.

A special boost is applied to threads that are awoken as a result of setting an event with the special functions *NtSetEventBoostPriority* (used in `Ntdll.dll` for critical sections) and *KeSetEventBoostPriority* (used for executive resources) or if a signaling gate is used (such as with pushlocks). If a thread waiting for an event is woken up as a result of the special event

boost function and its priority is 13 or below, it will have its priority boosted to be the setting thread's priority plus one. If its quantum is less than 4 quantum units, it is set to 4 quantum units. This boost is removed at quantum end.

Boosts During Waiting on Executive Resources

When a thread attempts to acquire an executive resource (ERESOURCE; see Chapter 3 for more information on kernel synchronization objects) that is already owned exclusively by another thread, it must enter a wait state until the other thread has released the resource. To avoid deadlocks, the executive performs this wait in intervals of five seconds instead of doing an infinite wait on the resource.

At the end of these five seconds, if the resource is still owned, the executive will attempt to prevent CPU starvation by acquiring the dispatcher lock, boosting the owning thread or threads, and performing another wait. Because the dispatcher lock is held and the thread's *WaitNext* flag is set to TRUE, this ensures a consistent state during the boosting process until the next wait is done.

This boost operates in the following manner:

- The boost is always applied to the base priority (not the current priority) of the owner thread.
- The boost raises priority to 14.
- The boost is only applied if the owner thread has a lower priority than the waiting thread, and only if the owner thread's priority isn't already 14.
- The quantum of the thread is reset so that the thread gets to run at the elevated priority for a full quantum, instead of only the quantum it had left. Just like other boosts, at each quantum end, the priority boost will slowly decrease by one level.

Because executive resources can be either shared or exclusive, the kernel will first boost the exclusive owner and then check for shared owners and boost all of them. When the waiting thread enters the wait state again, the hope is that the scheduler will schedule one of the owner threads, which will have enough time to complete its work and release the resource. It's important to note that this boosting mechanism is used only if the resource doesn't have the Disable Boost flag set, which developers can choose to set if the priority inversion mechanism described here works well with their usage of the resource.

Additionally, this mechanism isn't perfect. For example, if the resource has multiple shared owners, the executive will boost all those threads to priority 14, resulting in a sudden surge of high-priority threads on the system, all with full quanta. Although the exclusive thread will run first (since it was the first to be boosted and therefore first on the ready list), the other shared owners will run next, since the waiting thread's priority was not boosted. Only until after all the shared owners have gotten a chance to run and their priority decreased

below the waiting thread will the waiting thread finally get its chance to acquire the resource. Because shared owners can promote or convert their ownership from shared to exclusive as soon as the exclusive owner releases the resource, it's possible for this mechanism not to work as intended.

Priority Boosts for Foreground Threads After Waits

Whenever a thread in the foreground process completes a wait operation on a kernel object, the kernel function *KiUnwaitThread* boosts its current (not base) priority by the current value of *PsPrioritySeparation*. (The windowing system is responsible for determining which process is considered to be in the foreground.) As described in the section on quantum controls, *PsPrioritySeparation* reflects the quantum-table index used to select quanta for the threads of foreground applications. However, in this case, it is being used as a priority boost value.

The reason for this boost is to improve the responsiveness of interactive applications—by giving the foreground application a small boost when it completes a wait, it has a better chance of running right away, especially when other processes at the same base priority might be running in the background.

Unlike other types of boosting, this boost applies to all Windows systems, and you *can't* disable this boost, even if you've disabled priority boosting using the Windows *SetThreadPriorityBoost* function.

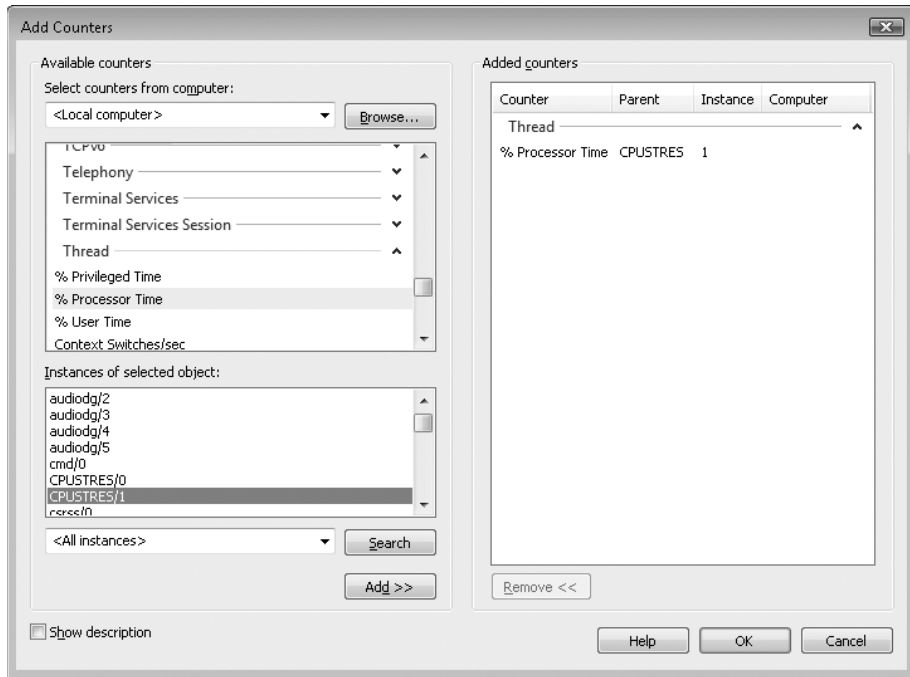


EXPERIMENT: Watching Foreground Priority Boosts and Decays

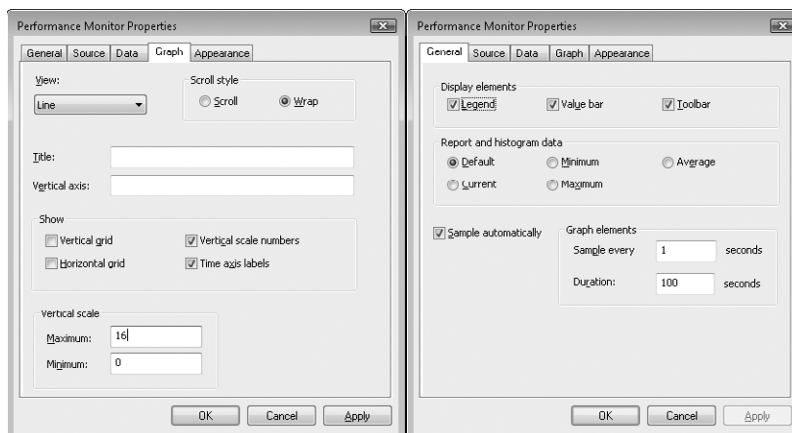
Using the CPU Stress tool, you can watch priority boosts in action. Take the following steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. Select the Programs option. This causes *PsPrioritySeparation* to get a value of 2.
2. Run *Cpustres.exe*, and change the activity of thread 1 from Low to Busy.
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the % Processor Time counter.

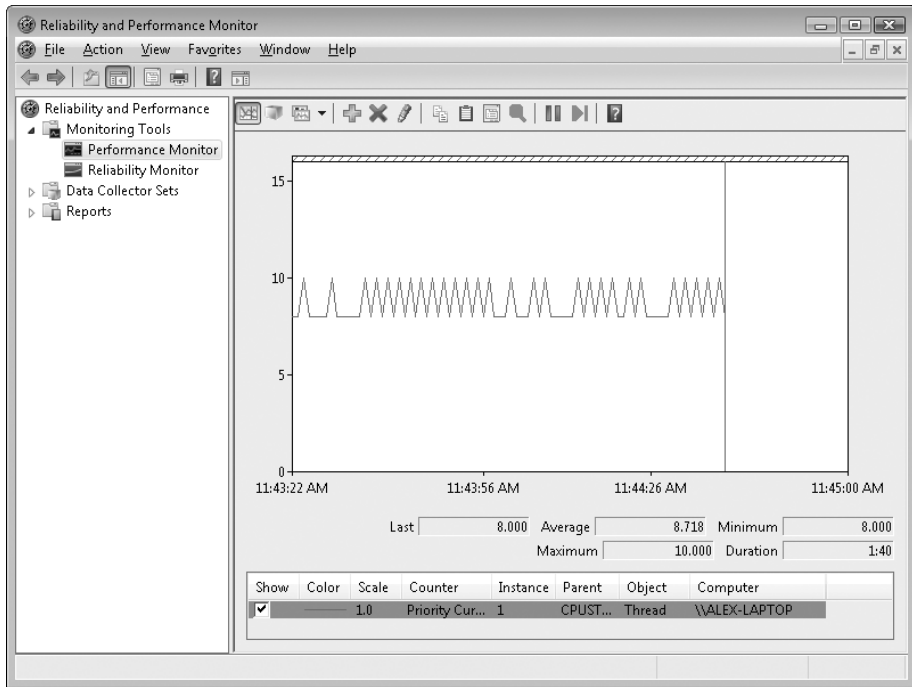
6. In the Instances box, select <All instances> and click Search. Scroll down until you see the CPUSTRES process. Select the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



7. Click the Add button, and then click OK.
8. Select Properties from the Action menu. Change the Vertical Scale Maximum to 16 and set the interval to Sample Every N Seconds in the Graph Elements area.



9. Now bring the CPUSTRES process to the foreground. You should see the priority of the CPUSTRES thread being boosted by 2 and then decaying back to the base priority as follows:



10. The reason CPUSTRES receives a boost of 2 periodically is because the thread you're monitoring is sleeping about 25 percent of the time and then waking up (this is the Busy Activity level). The boost is applied when the thread wakes up. If you set the Activity level to Maximum, you won't see any boosts because Maximum in CPUSTRES puts the thread into an infinite loop. Therefore, the thread doesn't invoke any wait functions and as a result doesn't receive any boosts.
11. When you've finished, exit Reliability and Performance Monitor and CPU Stress.

Priority Boosts After GUI Threads Wake Up

Threads that own windows receive an additional boost of 2 when they wake up because of windowing activity such as the arrival of window messages. The windowing system (Win32k.sys) applies this boost when it calls *KeSetEvent* to set an event used to wake up a GUI thread. The reason for this boost is similar to the previous one—to favor interactive applications.



EXPERIMENT: Watching Priority Boosts on GUI Threads

You can also see the windowing system apply its boost of 2 for GUI threads that wake up to process window messages by monitoring the current priority of a GUI application and moving the mouse across the window. Just follow these steps:

1. Open the System utility in Control Panel (or right-click on your computer name's icon on the desktop, and choose Properties). Click the Advanced System Settings label, select the Advanced tab, click the Settings button in the Performance section, and finally click the Advanced tab. Be sure that the Programs option is selected. This causes *PsPrioritySeperation* to get a value of 2.
2. Run Notepad from the Start menu by selecting Programs/Accessories/Notepad.
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the % Processor Time counter.
6. In the Instances box, select <All instances>, and then click Search. Scroll down until you see Notepad thread 0. Click it, click the Add button, and then click OK.
7. As in the previous experiment, select Properties from the Action menu. Change the Vertical Scale Maximum to 16, set the interval to Sample Every *N* Seconds in the Graph Elements area, and click OK.
8. You should see the priority of thread 0 in Notepad at 8, 9, or 10. Because Notepad entered a wait state shortly after it received the boost of 2 that threads in the foreground process receive, it might not yet have decayed from 10 to 9 and then to 8.
9. With Reliability and Performance Monitor in the foreground, move the mouse across the Notepad window. (Make both windows visible on the desktop.) You'll see that the priority sometimes remains at 10 and sometimes at 9, for the reasons just explained. (The reason you won't likely catch Notepad at 8 is that it runs so little after receiving the GUI thread boost of 2 that it never experiences more than one priority level of decay before waking up again because of additional windowing activity and receiving the boost of 2 again.)
10. Now bring Notepad to the foreground. You should see the priority rise to 12 and remain there (or drop to 11, because it might experience the normal priority decay that occurs for boosted threads on the quantum end) because the thread is receiving two boosts: the boost of 2 applied to GUI threads when they wake up

to process windowing input and an additional boost of 2 because Notepad is in the foreground.

11. If you then move the mouse over Notepad (while it's still in the foreground), you might see the priority drop to 11 (or maybe even 10) as it experiences the priority decay that normally occurs on boosted threads as they complete their turn. However, the boost of 2 that is applied because it's the foreground process remains as long as Notepad remains in the foreground.
12. When you've finished, exit Reliability and Performance Monitor and Notepad.

Priority Boosts for CPU Starvation

Imagine the following situation: you have a priority 7 thread that's running, preventing a priority 4 thread from ever receiving CPU time; however, a priority 11 thread is waiting for some resource that the priority 4 thread has locked. But because the priority 7 thread in the middle is eating up all the CPU time, the priority 4 thread will never run long enough to finish whatever it's doing and release the resource blocking the priority 11 thread. What does Windows do to address this situation?

We have previously seen how the executive code responsible for executive resources manages this scenario by boosting the owner threads so that they can have a chance to run and release the resource. However, executive resources are only one of the many synchronization constructs available to developers, and the boosting technique will not apply to any other primitive. Therefore, Windows also includes a generic CPU starvation relief mechanism as part of a thread called the *balance set manager* (a system thread that exists primarily to perform memory management functions and is described in more detail in Chapter 9).

Once per second, this thread scans the ready queues for any threads that have been in the ready state (that is, haven't run) for approximately 4 seconds. If it finds such a thread, the balance set manager boosts the thread's priority to 15 and sets the quantum target to an equivalent CPU clock cycle count of 4 quantum units. Once the quantum is expired, the thread's priority decays immediately to its original base priority. If the thread wasn't finished and a higher priority thread is ready to run, the decayed thread will return to the ready queue, where it again becomes eligible for another boost if it remains there for another 4 seconds.

The balance set manager doesn't actually scan all ready threads every time it runs. To minimize the CPU time it uses, it scans only 16 ready threads; if there are more threads at that priority level, it remembers where it left off and picks up again on the next pass. Also, it will boost only 10 threads per pass—if it finds 10 threads meriting this particular boost (which would indicate an unusually busy system), it stops the scan at that point and picks up again on the next pass.



Note We mentioned earlier that scheduling decisions in Windows are not affected by the number of threads, and that they are made in *constant time*, or $O(1)$. Because the balance set manager does need to scan ready queues manually, this operation does depend on the number of threads on the system, and more threads will require more scanning time. However, the balance set manager is not considered part of the scheduler or its algorithms and is simply an extended mechanism to increase reliability. Additionally, because of the cap on threads and queues to scan, the performance impact is minimized and predictable in a worst-case scenario.

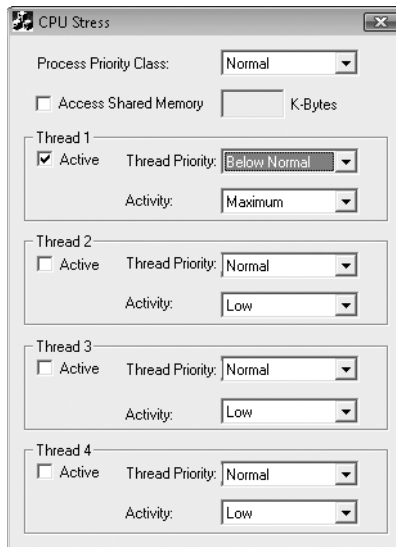
Will this algorithm always solve the priority inversion issue? No—it's not perfect by any means. But over time, CPU-starved threads should get enough CPU time to finish whatever processing they were doing and reenter a wait state.



EXPERIMENT: Watching Priority Boosts for CPU Starvation

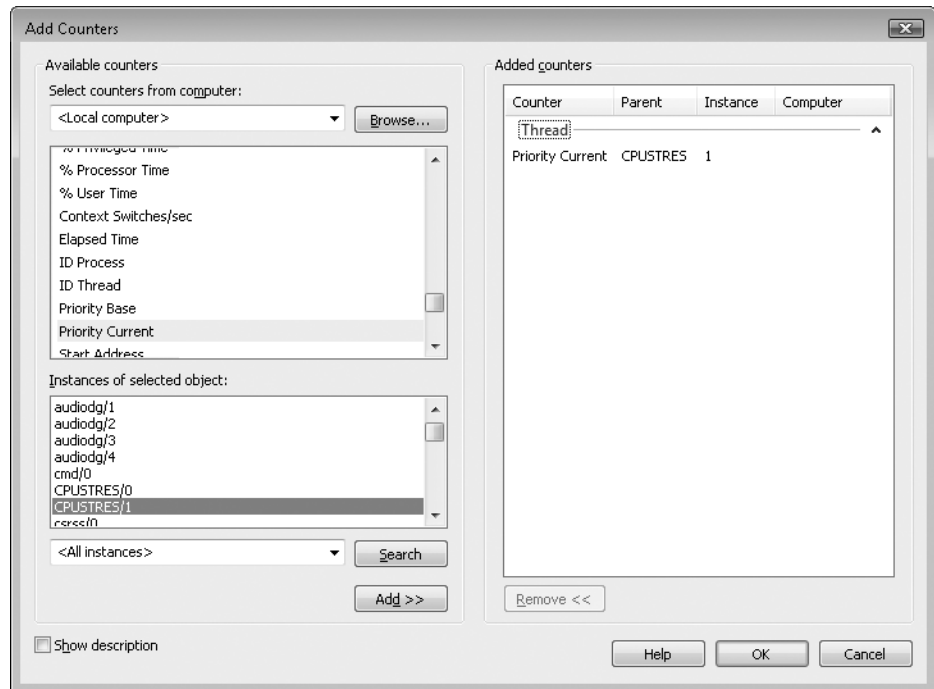
Using the CPU Stress tool, you can watch priority boosts in action. In this experiment, we'll see CPU usage change when a thread's priority is boosted. Take the following steps:

1. Run `Cpustres.exe`. Change the activity level of the active thread (by default, Thread 1) from Low to Maximum. Change the thread priority from Normal to Below Normal. The screen should look like this:



2. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
3. Click the Add Counter toolbar button (or press `Ctrl+I`) to bring up the Add Counters dialog box.

4. Select the Thread object, and then select the % Processor Time counter.
5. In the Instances box, select <All instances>, and then click Search. Scroll down until you see the CPUTRES process. Select the second thread (thread 1). (The first thread is the GUI thread.) You should see something like this:



6. Click the Add button, and then click OK.
7. Raise the priority of Performance Monitor to real time by running Task Manager, clicking the Processes tab, and selecting the Mmc.exe process. Right-click the process, select Set Priority, and then select Realtime. (If you receive a Task Manager Warning message box warning you of system instability, click the Yes button.) If you have a multiprocessor system, you will also need to change the affinity of the process: right-click and select Set Affinity. Then clear all other CPUs except for CPU 0.
8. Run another copy of CPU Stress. In this copy, change the activity level of Thread 1 from Low to Maximum.
9. Now switch back to Performance Monitor. You should see CPU activity every 6 or so seconds because the thread is boosted to priority 15. You can force updates to occur more frequently than every second by pausing the display with Ctrl+F, and then pressing Ctrl+U, which forces a manual update of the counters. Keep Ctrl+U pressed for continual refreshes.

When you've finished, exit Performance Monitor and the two copies of CPU Stress.



EXPERIMENT: “Listening” to Priority Boosting

To “hear” the effect of priority boosting for CPU starvation, perform the following steps on a system with a sound card:

1. Because of MMCSS’s priority boosts (which we will describe in the next subsection), you will need to stop the MultiMedia Class Scheduler Service by opening the Services management interface (Start, Programs, Administrative Tools, Services).
2. Run Windows Media Player (or some other audio playback program), and begin playing some audio content.
3. Run Cpustres, and set the activity level of Thread 1 to Maximum.
4. Raise the priority of Thread 1 from Normal to Time Critical.
5. You should hear the music playback stop as the compute-bound thread begins consuming all available CPU time.
6. Every so often, you should hear bits of sound as the starved thread in the audio playback process gets boosted to 15 and runs enough to send more data to the sound card.
7. Stop Cpustres and Windows Media Player, and start the MMCSS service again.

Priority Boosts for MultiMedia Applications and Games (MMCSS)

As we’ve just seen in the last experiment, although Windows’s CPU starvation priority boosts may be enough to get a thread out of an abnormally long wait state or potential deadlock, they simply cannot deal with the resource requirements imposed by a CPU-intensive application such as Windows Media Player or a 3D computer game.

Skipping and other audio glitches have been a common source of irritation among Windows users in the past, and the user-mode audio stack in Windows Vista would have only made the situation worse since it offers even more chances for preemption. To address this, Windows Vista incorporates a new service (called MMCSS, described earlier in this chapter) whose purpose is to ensure “glitch-free” multimedia playback for applications that register with it.

MMCSS works by defining several tasks, including:

- Audio
- Capture
- Distribution
- Games

- Playback
- Pro Audio
- Window Manager



Note You can find the settings for MMCSS, including a lists of tasks (which can be modified by OEMs to include other specific tasks as appropriate) in the registry keys under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile. Additionally, the SystemResponsiveness value allows you to fine-tune how much CPU usage MMCSS guarantees to low-priority threads.

In turn, each of these tasks includes information about the various properties that differentiate them. The most important one for scheduling is called the Scheduling Category, which is the primary factor determining the priority of threads registered with MMCSS. Table 5-19 shows the various scheduling categories.

TABLE 5-19 Scheduling Categories

Category	Priority	Description
High	23-26	Pro Audio threads running at a higher priority than any other thread on the system except for critical system threads.
Medium	16-22	Threads part of a foreground application such as Windows Media Player.
Low	8-15	All other threads not part of the previous categories.
Exhausted	1-7	Threads that have exhausted their share of the CPU and will only continue running if no other higher priority threads are ready to run.

The main mechanism behind MMCSS boosts the priority of threads inside a registered process to the priority level matching their scheduling category and relative priority within this category for a guaranteed period of time. It then lowers those threads to the Exhausted category so that other, nonmultimedia threads on the system can also get a chance to execute.

By default, multimedia threads will get 80 percent of the CPU time available, while other threads will receive 20 percent (based on a sample of 10 ms; in other words, 8 ms and 2 ms). MMCSS itself runs at priority 27, since it needs to preempt any Pro Audio threads in order to lower their priority to the Exhausted category.

It is important to emphasize that the kernel still does the actual boosting of the values inside the KTHREAD (MMCSS simply makes the same kind of system call any other application would do), and the scheduler is still in control of these threads. It is simply their high priority that makes them run almost uninterrupted on a machine, since they are in the real-time range and well above threads that most user applications would be running in.

As was discussed earlier, changing the relative thread priorities within a process does not usually make sense, and no tool allows this because only developers understand the importance of the various threads in their programs.

On the other hand, because applications must manually register with MMCSS and provide it with information about what kind of thread this is, MMCSS does have the necessary data to change these relative thread priorities (and developers are well aware that this will be happening).



EXPERIMENT: “Listening” to MMCSS Priority Boosting

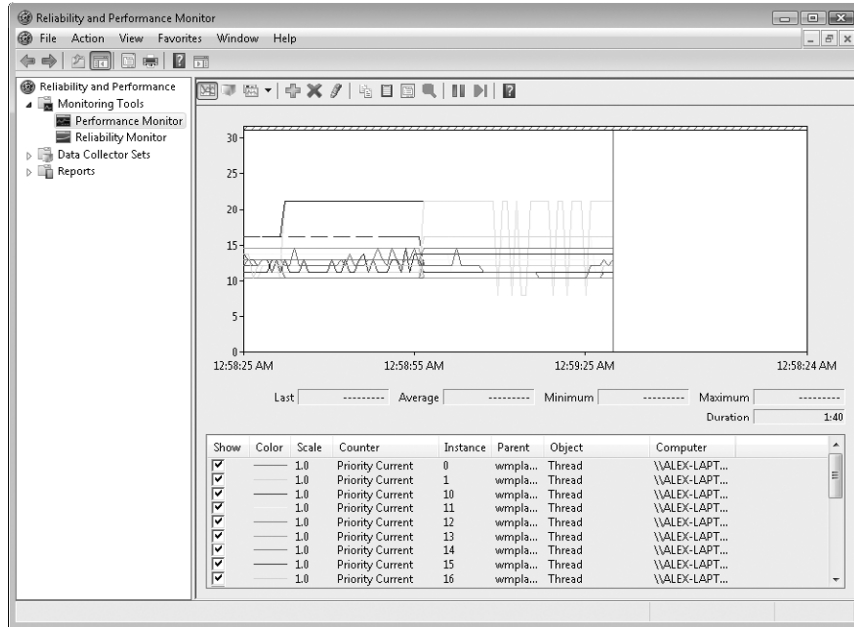
We are now going to perform the same experiment as the prior one but without disabling the MMCSS service. In addition, we’ll take a look at the Performance tool to check the priority of the Windows Media Player threads.

1. Run Windows Media Player (other playback programs may not yet take advantage of the API calls required to register with MMCSS) and begin playing some audio content.
2. If you have a multiprocessor machine, be sure to set the affinity of the Wmplayer.exe process so that it only runs on one CPU (since we’ll be using only one CPUTRES worker thread).
3. Start the Performance tool by selecting Programs from the Start menu and then selecting Reliability And Performance Monitor from the Administrative Tools menu. Click on the Performance Monitor entry under Monitoring Tools.
4. Click the Add Counter toolbar button (or press Ctrl+I) to bring up the Add Counters dialog box.
5. Select the Thread object, and then select the % Processor Time counter.
6. In the Instances box, select <All instances>, and then click Search. Scroll down until you see Wmplayer, and then select all its threads. Click the Add button, and then click OK.
7. As in the previous experiment, select Properties from the Action menu. Change the Vertical Scale Maximum to 31, set the interval to Sample Every *N* Seconds in the Graph Elements area, and click OK.

You should see one or more priority 21 threads inside Wmplayer, which will be constantly running unless there is a higher-priority thread requiring the CPU after they are dropped to the Exhausted category.

8. Run Cpustres, and set the activity level of Thread 1 to Maximum.
9. Raise the priority of Thread 1 from Normal to Time Critical.
10. You should notice the system slowing down considerably, but the music playback will continue. Every so often, you’ll be able to get back some responsiveness from the rest of the system. Use this time to stop Cpustres.

11. If the Performance tool was unable to capture data during the time CpuStres ran, run it again, but use Highest instead of Time Critical. This change will slow down the system less, but it still requires boosting from MMCSS, and, because once the multimedia thread is put in the Exhausted category, there will always be a higher priority thread requesting the CPU (CPUSTRES), you should notice Wmplayer's priority 21 thread drop every so often, as shown here.



MMCSS's functionality does not stop at simple priority boosting, however. Because of the nature of network drivers on Windows and the NDIS stack, DPCs are quite common mechanisms for delaying work after an interrupt has been received from the network card. Because DPCs run at an IRQL level higher than user-mode code (see Chapter 3 for more information on DPCs and IRQLs), long-running network card driver code could still interrupt media playback during network transfers, or when playing a game for example.

Therefore, MMCSS also sends a special command to the network stack, telling it to throttle network packets during the duration of the media playback. This throttling is designed to maximize playback performance, at the cost of some small loss in network throughput (which would not be noticeable for network operations usually performed during playback, such as playing an online game). The exact mechanisms behind it do not belong to any area of the scheduler, so we will leave them out of this description.



Note The original implementation of the network throttling code had some design issues causing significant network throughput loss on machines with 1000 Mbit network adapters, especially if multiple adapters were present on the system (a common feature of midrange motherboards). This issue was analyzed by the MMCSS and networking teams at Microsoft and later fixed.

Multiprocessor Systems

On a uniprocessor system, scheduling is relatively simple: the highest-priority thread that wants to run is always running. On a multiprocessor system, it is more complex, as Windows attempts to schedule threads on the most optimal processor for the thread, taking into account the thread's preferred and previous processors, as well as the configuration of the multiprocessor system. Therefore, while Windows attempts to schedule the highest-priority runnable threads on all available CPUs, it only guarantees to be running the (single) highest-priority thread somewhere.

Before we describe the specific algorithms used to choose which threads run where and when, let's examine the additional information Windows maintains to track thread and processor state on multiprocessor systems and the two different types of multiprocessor systems supported by Windows (hyperthreaded, multicore, and NUMA).

Multiprocessor Considerations in the Dispatcher Database

In addition to the ready queues and the ready summary, Windows maintains two bitmasks that track the state of the processors on the system. (How these bitmasks are used is explained in the upcoming section "Multiprocessor Thread-Scheduling Algorithms".) Following are the two bitmasks that Windows maintains:

- The *active processor mask* (*KeActiveProcessors*), which has a bit set for each usable processor on the system (This might be less than the number of actual processors if the licensing limits of the version of Windows running supports less than the number of available physical processors.)
- The *idle summary* (*KIdleSummary*), in which each set bit represents an idle processor

Whereas on uniprocessor systems, the dispatcher database is locked by raising IRQL to both DPC/dispatch level and Synch level, on multiprocessor systems more is required, because each processor could, at the same time, raise IRQL and attempt to operate on the dispatcher database. (This is true for any systemwide structure accessed from high IRQL.) (See Chapter 3 for a general description of kernel synchronization and spinlocks.)

Because on a multiprocessor system one processor might need to modify another processor's per-CPU scheduling data structures (such as inserting a thread that would like to run on a certain processor), these structures are synchronized by using a new per-PRCB queued

spinlock, which is held at IRQL SYNCH_LEVEL. (See Table 5-20 for the various values of SYNCH_LEVEL.) Thus, thread selection can occur while locking only an individual processor's PRCB, in contrast to doing this on Windows XP, where the systemwide dispatcher spinlock had to be held.

TABLE 5-20 IRQL SYNCH_LEVEL on Multiprocessor Systems

CPU Type	IRQL
Systems running on x86	27
Systems running on x64	12
Systems running on IA64	12

There is also a per-CPU list of threads in the deferred ready state. These represent threads that are ready to run but have not yet been readied for execution; the actual ready operation has been deferred to a more appropriate time. Because each processor manipulates only its own per-processor deferred ready list, this list is not synchronized by the PRCB spinlock. The deferred ready thread list is processed before exiting the thread dispatcher, before performing a context switch, and after processing a DPC. Threads on the deferred ready list are either dispatched immediately or are moved to the per-processor ready queue for their priority level.

Note that the systemwide dispatcher spinlock still exists and is used, but it is held only for the time needed to modify systemwide state that might affect which thread runs next. For example, changes to synchronization objects (mutexes, events, and semaphores) and their wait queues require holding the dispatcher lock to prevent more than one processor from changing the state of such objects (and the consequential action of possibly readying threads for execution). Other examples include changing the priority of a thread, timer expiration, and swapping of thread kernel stacks.

Thread context switching is also synchronized by using a finer-grained per-thread spinlock, whereas in Windows XP context switching was synchronized by holding a systemwide context swap spinlock.

Hyperthreaded and Multicore Systems

As described in the "Symmetric Multiprocessing" section in Chapter 2, Windows supports hyperthreaded and multicore multiprocessor systems in two primary ways:

1. Logical processors as well as per-package cores do not count against physical processor licensing limits. For example, Windows Vista Home Basic, which has a licensed processor limit of 1, will use all four cores on a single processor system.
2. When choosing a processor for a thread, if there is a physical processor with all logical processors idle, a logical processor from that physical processor will be selected, as opposed to choosing an idle logical processor on a physical processor that has another logical processor running a thread.



EXPERIMENT: Viewing Hyperthreading Information

You can examine the information Windows maintains for hyperthreaded processors using the `!smt` command in the kernel debugger. The following output is from a dual-processor hyperthreaded Xeon system (four logical processors):

```

1kd> !smt
SMT Summary:
-----
    KeActiveProcessors: ****----- (0000000f)
      KiIdleSummary:  -***----- (0000000e)
No  PRCB      Set Master SMT Set          #LP IAID
0  ffdff120 Master  *_*----- (00000005)  2  00
1  f771f120 Master  -*-*----- (0000000a)  2  06
2  f7727120 ffdff120 *_*----- (00000005)  2  01
3  f772f120 f771f120 -*-*----- (0000000a)  2  07

    Number of licensed physical processors: 2

```

Logical processors 0 and 1 are on separate physical processors (as indicated by the term “Master”).

NUMA Systems

Another type of multiprocessor system supported by Windows is one with a nonuniform memory access (NUMA) architecture. In a NUMA system, processors are grouped together in smaller units called nodes. Each node has its own processors and memory and is connected to the larger system through a cache-coherent interconnect bus. These systems are called “nonuniform” because each node has its own local high-speed memory. While any processor in any node can access all of memory, node-local memory is much faster to access.

The kernel maintains information about each node in a NUMA system in a data structure called `KNODE`. The kernel variable `KeNodeBlock` is an array of pointers to the `KNODE` structures for each node. The format of the `KNODE` structure can be shown using the `dt` command in the kernel debugger, as shown here:

```

1kd> dt nt!_knode
nt!_KNODE
+0x000 PagedPoolSListHead : _SLIST_HEADER
+0x008 NonPagedPoolSListHead : [3] _SLIST_HEADER
+0x020 PfnDereferenceSListHead : _SLIST_HEADER
+0x028 ProcessorMask      : Uint4B
+0x02c Color              : UChar
+0x02d Seed               : UChar
+0x02e NodeNumber        : UChar
+0x02f Flags              : _flags
+0x030 MmShiftedColor    : Uint4B
+0x034 FreeCount         : [2] Uint4B
+0x03c PfnDeferredList    : Ptr32 _SINGLE_LIST_ENTRY
+0x040 CachedKernelStacks : _CACHED_KSTACK_LIST

```



EXPERIMENT: Viewing NUMA Information

You can examine the information Windows maintains for each node in a NUMA system using the *!numa* command in the kernel debugger. The following partial output is from a 32-processor NUMA system by NEC with 4 processors per node:

```
21: kd> !numa
NUMA Summary:
-----
Number of NUMA nodes : 8
Number of Processors : 32
MmAvailablePages      : 0x00F70D2C
KeActiveProcessors    : *****-----
                        (00000000ffffffff)

NODE 0 (E00000008428AE00):
  ProcessorMask        : ****-----
  Color                : 0x00000000
  MmShiftedColor       : 0x00000000
  Seed                 : 0x00000000
  Zeroed Page Count: 0x00000000001CF330
  Free Page Count      : 0x0000000000000000

NODE 1 (E00001597A9A2200):
  ProcessorMask        : ----*****-----
  Color                : 0x00000001
  MmShiftedColor       : 0x00000004
  Seed                 : 0x00000006
  Zeroed Page Count: 0x00000000001F77A0
  Free Page Count      : 0x0000000000000004
```

The following partial output is from a 64-processor NUMA system from Hewlett-Packard with 4 processors per node:

```
26: kd> !numa
NUMA Summary:
-----
Number of NUMA nodes : 16
Number of Processors : 64
MmAvailablePages      : 0x03F55E67
KeActiveProcessors    : *****-----
                        (ffffffffffffffff)

NODE 0 (E000000084261900):
  ProcessorMask        : ****-----
  Color                : 0x00000000
  MmShiftedColor       : 0x00000000
  Seed                 : 0x00000001
  Zeroed Page Count: 0x00000000003F4430
  Free Page Count      : 0x0000000000000000
```

```

NODE 1 (E0000145FF992200):
  ProcessorMask      : -----***-----
  Color              : 0x00000001
  MmShiftedColor     : 0x00000040
  Seed                : 0x00000007
  Zeroed Page Count : 0x00000000003ED59A
  Free Page Count    : 0x0000000000000000

```

Applications that want to gain the most performance out of NUMA systems can set the affinity mask to restrict a process to the processors in a specific node. This information can be obtained using the functions listed in Table 5-21. Functions that can alter thread affinity are listed in Table 5-13.

TABLE 5-21 NUMA-Related Functions

Function	Description
<i>GetNumaHighestNodeNumber</i>	Retrieves the node that currently has the highest number.
<i>GetNumaNodeProcessorMask</i>	Retrieves the processor mask for the specified node.
<i>GetNumaProximityNode</i>	Returns the NUMA node number for the given proximity ID.
<i>GetNumaProcessorNode</i>	Retrieves the node number for the specified processor.

How the scheduling algorithms take into account NUMA systems will be covered in the upcoming section “Multiprocessor Thread-Scheduling Algorithms” (and the optimizations in the memory manager to take advantage of node-local memory are covered in Chapter 9).

Affinity

Each thread has an *affinity mask* that specifies the processors on which the thread is allowed to run. The thread affinity mask is inherited from the process affinity mask. By default, all processes (and therefore all threads) begin with an affinity mask that is equal to the set of active processors on the system—in other words, the system is free to schedule all threads on any available processor.

However, to optimize throughput and/or partition workloads to a specific set of processors, applications can choose to change the affinity mask for a thread. This can be done at several levels:

- Calling the *SetThreadAffinityMask* function to set the affinity for an individual thread
- Calling the *SetProcessAffinityMask* function to set the affinity for all the threads in a process. Task Manager and Process Explorer provide a GUI to this function if you right-click a process and choose Set Affinity. The Psexec tool (from Sysinternals) provides a command-line interface to this function. (See the *-a* switch.)

- By making a process a member of a job that has a jobwide affinity mask set using the *SetInformationJobObject* function (Jobs are described in the upcoming “Job Objects” section.)
- By specifying an affinity mask in the image header when compiling the application (For more information on the detailed format of Windows images, search for “Portable Executable and Common Object File Format Specification” on www.microsoft.com.)

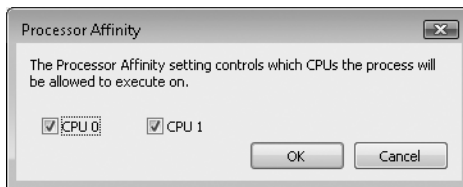
You can also set the “uniprocessor” flag for an image (at compile time). If this flag is set, the system chooses a single processor at process creation time and assigns that as the process affinity mask, starting with the first processor and then going round-robin across all the processors. For example, on a dual-processor system, the first time you run an image marked as uniprocessor, it is assigned to CPU 0; the second time, CPU 1; the third time, CPU 0; the fourth time, CPU 1; and so on. This flag can be useful as a temporary workaround for programs that have multithreaded synchronization bugs that, as a result of race conditions, surface on multiprocessor systems but that don’t occur on uniprocessor systems. (This has actually saved the authors of this book on two different occasions.)



EXPERIMENT: Viewing and Changing Process Affinity

In this experiment, you will modify the affinity settings for a process and see that process affinity is inherited by new processes:

1. Run the command prompt (Cmd.exe).
2. Run Task Manager or Process Explorer, and find the Cmd.exe process in the process list.
3. Right-click the process, and select Affinity. A list of processors should be displayed. For example, on a dual-processor system you will see this:



4. Select a subset of the available processors on the system, and click OK. The process’s threads are now restricted to run on the processors you just selected.
5. Now run Notepad.exe from the command prompt (by typing **Notepad.exe**).
6. Go back to Task Manager or Process Explorer and find the new Notepad process. Right-click it, and choose Affinity. You should see the same list of processors you chose for the command prompt process. This is because processes inherit their affinity settings from their parent.

Windows won't move a running thread that could run on a different processor from one CPU to a second processor to permit a thread with an affinity for the first processor to run on the first processor. For example, consider this scenario: CPU 0 is running a priority 8 thread that can run on any processor, and CPU 1 is running a priority 4 thread that can run on any processor. A priority 6 thread that can run on only CPU 0 becomes ready. What happens? Windows won't move the priority 8 thread from CPU 0 to CPU 1 (preempting the priority 4 thread) so that the priority 6 thread can run; the priority 6 thread has to wait.

Therefore, changing the affinity mask for a process or a thread can result in threads getting less CPU time than they normally would, as Windows is restricted from running the thread on certain processors. Therefore, setting affinity should be done with extreme care—in most cases, it is optimal to let Windows decide which threads run where.

Ideal and Last Processor

Each thread has two CPU numbers stored in the kernel thread block:

- *Ideal processor*, or the preferred processor that this thread should run on
- *Last processor*, or the processor on which the thread last ran

The ideal processor for a thread is chosen when a thread is created using a seed in the process block. The seed is incremented each time a thread is created so that the ideal processor for each new thread in the process will rotate through the available processors on the system. For example, the first thread in the first process on the system is assigned an ideal processor of 0. The second thread in that process is assigned an ideal processor of 1. However, the next process in the system has its first thread's ideal processor set to 1, the second to 2, and so on. In that way, the threads within each process are spread evenly across the processors.

Note that this assumes the threads within a process are doing an equal amount of work. This is typically not the case in a multithreaded process, which normally has one or more house-keeping threads and then a number of worker threads. Therefore, a multithreaded application that wants to take full advantage of the platform might find it advantageous to specify the ideal processor numbers for its threads by using the *SetThreadIdealProcessor* function.

On hyperthreaded systems, the next ideal processor is the first logical processor on the next physical processor. For example, on a dual-processor hyperthreaded system with four logical processors, if the ideal processor for the first thread is assigned to logical processor 0, the second thread would be assigned to logical processor 2, the third thread to logical processor 1, the fourth thread to logical processor 3, and so forth. In this way, the threads are spread evenly across the physical processors.

On NUMA systems, when a process is created, an ideal node for the process is selected. The first process is assigned to node 0, the second process to node 1, and so on. Then, the ideal processors for the threads in the process are chosen from the process's ideal node. The ideal

processor for the first thread in a process is assigned to the first processor in the node. As additional threads are created in processes with the same ideal node, the next processor is used for the next thread's ideal processor, and so on.

Dynamic Processor Addition and Replacement

As we've seen, developers can fine-tune which threads are allowed to (and in the case of the ideal processor, *should*) run on which processor. This works fine on systems that have a constant number of processors during their run time (for example, desktop machines require shutting down the computer to make any sort of hardware changes to the processor or their count).

Today's server systems, however, cannot afford the downtime that CPU replacement or addition normally requires. In fact, one of the times when adding a CPU is required for a server is at times of high load that is above what the machine can support at its current level of performance. Having to shut down the server during a period of peak usage would defeat the purpose. To meet this requirement, the latest generation of server motherboards and systems support the addition of processors (as well as their replacement) while the machine is still running. The ACPI BIOS and related hardware on the machine have been specifically built to allow and be aware of this need, but operating system participation is required for full support.

Dynamic processor support is provided through the HAL, which will notify the kernel of a new processor on the system through the function *KeStartDynamicProcessor*. This routine does similar work to that performed when the system detects more than one processor at startup and needs to initialize the structures related to them. When a dynamic processor is added, a variety of system components perform some additional work. For example, the memory manager allocates new pages and memory structures optimized for the CPU. It also initializes a new DPC kernel stack while the kernel initializes the Global Descriptor Table (GDT), the Interrupt Descriptor Table (IDT), the processor control region (PCR), the processor control block (PRCB), and other related structures for the processor.

Other executive parts of the kernel are also called, mostly to initialize the per-processor lookaside lists for the processor that was added. For example, the I/O manager, the executive lookaside list code, the cache manager, and the object manager all use per-processor lookaside lists for their frequently allocated structures.

Finally, the kernel initializes threaded DPC support for the processor and adjusts exported kernel variables to report the new processor. Different memory manager masks and process seeds based on processor counts are also updated, and processor features need to be updated for the new processor to match the rest of the system (for example, enabling virtualization support on the newly added processor). The initialization sequence completes with the notification to the Windows Hardware Error Architecture (WHEA) component that a new processor is online.

The HAL is also involved in this process. It is called once to start the dynamic processor after the kernel is aware of it, and it is called again after the kernel has finished initialization of the processor. However, these notifications and callbacks only make the kernel aware and respond to processor changes. Although an additional processor increases the throughput of the kernel, it does nothing to help drivers.

To handle drivers, the system has a new default executive callback, the processor add callback, that drivers can register with for notifications. Similar to the callbacks that notify drivers of power state or system time changes, this callback allows driver code to, for example, create a new worker thread if desirable so that it can handle more work at the same time.

Once drivers are notified, the final kernel component called is the Plug and Play manager, which adds the processor to the system's device node and rebalances interrupts so that the new processor can handle interrupts that were already registered for other processors. Unfortunately, until now, CPU-hungry applications have still been left out of this process, but Windows Server 2008 and Windows Vista Service Pack 1 have improved the process to allow applications to be able to take advantage of newer processors as well.

However, a sudden change of affinity can have potentially breaking changes for a running application (especially when going from a single-processor to a multiprocessor environment) through the appearance of potential race conditions or simply misdistribution of work (since the process might have calculated the perfect ratios at startup, based on the number of CPUs it was aware of). As a result, applications do not take advantage of a dynamically added processor by default—they must request it.

The Windows APIs *SetProcessAffinityUpdateMode* and *QueryProcessAffinityMode* (which use the undocumented *NtSet/QueryInformationProcess* system call) tell the process manager that these applications should have their affinity updated (by setting the *AffinityUpdateEnable* flag in *EPROCESS*), or that they do not want to deal with affinity updates (by setting the *AffinityPermanent* flag in *EPROCESS*). Once an application has told the system that its affinity is permanent, it cannot later change its mind and request affinity updates, so this is a one-time change.

As part of *KeStartDynamicProcessor*, a new step has been added after interrupts are rebalanced, which is to call the process manager to perform affinity updates through *PsUpdateActiveProcessAffinity*. Some Windows core processes and services already have affinity updates enabled, while third-party software will need to be recompiled to take advantage of the new API call. The System process, Svchost processes, and Smss are all compatible with dynamic processor addition.

Multiprocessor Thread-Scheduling Algorithms

Now that we've described the types of multiprocessor systems supported by Windows as well as the thread affinity and ideal processor settings, we're ready to examine how this

information is used to determine which threads run where. There are two basic decisions to describe:

- Choosing a processor for a thread that wants to run
- Choosing a thread on a processor that needs something to do

Choosing a Processor for a Thread When There Are Idle Processors

When a thread becomes ready to run, Windows first tries to schedule the thread to run on an idle processor. If there is a choice of idle processors, preference is given first to the thread's ideal processor, then to the thread's previous processor, and then to the currently executing processor (that is, the CPU on which the scheduling code is running).

To select the best idle processor, Windows starts with the set of idle processors that the thread's affinity mask permits it to run on. If the system is NUMA and there are idle CPUs in the node containing the thread's ideal processor, the list of idle processors is reduced to that set. If this eliminates all idle processors, the reduction is not done. Next, if the system is running hyperthreaded processors and there is a physical processor with all logical processors idle, the list of idle processors is reduced to that set. If that results in an empty set of processors, the reduction is not done.

If the current processor (the processor trying to determine what to do with the thread that wants to run) is in the remaining idle processor set, the thread is scheduled on it. If the current processor is not in the remaining set of idle processors, it is a hyperthreaded system, and there is an idle logical processor on the physical processor containing the ideal processor for the thread, the idle processors are reduced to that set. If not, the system checks whether there are any idle logical processors on the physical processor containing the thread's previous processor. If that set is nonzero, the idle processors are reduced to that list. Finally, the lowest numbered CPU in the remaining set is selected as the processor to run the thread on.

Once a processor has been selected for the thread to run on, that thread is put in the standby state and the idle processor's PRCB is updated to point to this thread. When the idle loop on that processor runs, it will see that a thread has been selected to run and will dispatch that thread.

Choosing a Processor for a Thread When There Are No Idle Processors

If there are no idle processors when a thread wants to run, Windows compares the priority of the thread running (or the one in the standby state) on the thread's ideal processor to determine whether it should preempt that thread.

If the thread's ideal processor already has a thread selected to run next (waiting in the standby state to be scheduled) and that thread's priority is less than the priority of the thread being readied for execution, the new thread preempts that first thread out of the standby

state and becomes the next thread for that CPU. If there is already a thread running on that CPU, Windows checks whether the priority of the currently running thread is less than the thread being readied for execution. If so, the currently running thread is marked to be preempted and Windows queues an interprocessor interrupt to the target processor to preempt the currently running thread in favor of this new thread.



Note Windows doesn't look at the priority of the current and next threads on all the CPUs—just on the one CPU selected as just described. If no thread can be preempted on that one CPU, the new thread is put in the ready queue for its priority level, where it awaits its turn to get scheduled. Therefore, Windows does not guarantee to be running all the highest-priority threads, but it will always run the highest-priority thread.

If the ready thread cannot be run right away, it is moved into the ready state where it awaits its turn to run. Note that threads are always put on their ideal processor's per-processor ready queues.

Selecting a Thread to Run on a Specific CPU

Because each processor has its own list of threads waiting to run on that processor, when a thread finishes running, the processor can simply check its per-processor ready queue for the next thread to run. If the per-processor ready queues are empty, the idle thread for that processor is scheduled. The idle thread then begins scanning other processor's ready queues for threads it can run. Note that on NUMA systems, the idle thread first looks at processors on its node before looking at other nodes' processors.

CPU Rate Limits

As part of the new hard quota management system added in Windows Vista (which builds on previous quota support present since the first version of Windows NT, but adds hard limits instead of soft hints), support for limiting CPU usage was added to the system in three different ways: per-session, per-user, or per-system. Unfortunately, information on enabling these new limits has not yet been documented, and no tool that is part of the operating system allows you to set these limits: you must modify the registry settings manually. Because all the quotas—save one—are memory quotas, we will cover those in Chapter 9, which deals with the memory manager, and focus our attention on the CPU rate limit.

The new quota system can be accessed through the registry key `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\QuotaSystem`, as well as through the standard *NtSetInformationProcess* system call. CPU rate limits can therefore be set in one of three ways:

- By creating a new value called `CpuRateLimit` and entering the rate information.

- By creating a new key with the security ID (SID) of the account you want to limit, and creating a `CpuRateLimit` value inside that key.
- By calling `NtSetInformationProcess` and giving it the process handle of the process to limit and the CPU rate limiting information.

In all three cases, the CPU rate limit data is not a straightforward value; it is based on a compressed bitfield, documented in the WDK as part of the `RATE_QUOTA_LIMIT` structure. The bottom four bits define the *rate phase*, which can be expressed either as one, two, or three seconds—this value defines how often the rate limiting should be applied and is called the `PS_RATE_PHASE`. The rest of the bits are used for the actual rate, as a value representing a percentage of maximum CPU usage. Because any number from 0 to 100 can be represented with only 7 bits, the rest of the bits are unused. Therefore, a rate limit of 40 percent every 2 seconds would be defined by the value `0x282`, or `101000 0010` in binary.

The process manager, which is responsible for enforcing the CPU rate limit, uses a variety of system mechanisms to do its job. First of all, rate limiting is able to reliably work because of the CPU cycle count improvements discussed earlier, which allow the process manager to accurately determine how much CPU time a process has taken and know whether the limit should be enforced. It then uses a combination of DPC and APC routines to throttle down DPC and APC CPU usage, which are outside the direct control of user-mode developers but still result in CPU usage in the system (in the case of a systemwide CPU rate limit).

Finally, the main mechanism through which rate limiting works is by creating an artificial wait on a kernel gate object (making the thread uniquely bound to this object and putting it in a wait state, which does not consume CPU cycles). This mechanism operates through the normal routine of an APC object queued to the thread or threads inside the process currently responsible for the work. The gate is signaled by an internal worker thread inside the process manager responsible for replenishment of the CPU usage, which is queued by a DPC responsible for replenishing systemwide CPU usage requests.

Job Objects

A *job object* is a nameable, securable, shareable kernel object that allows control of one or more processes as a group. A job object's basic function is to allow groups of processes to be managed and manipulated as a unit. A process can be a member of only one job object. By default, its association with the job object can't be broken and all processes created by the process and its descendents are associated with the same job object as well. The job object also records basic accounting information for all processes associated with the job and for all processes that were associated with the job but have since terminated. Table 5-22 lists the Windows functions to create and manipulate job objects.

TABLE 5-22 Windows API Functions for Jobs

Function	Description
<i>CreateJobObject</i>	Creates a job object (with an optional name)
<i>OpenJobObject</i>	Opens an existing job object by name
<i>AssignProcessToJobObject</i>	Adds a process to a job
<i>TerminateJobObject</i>	Terminates all processes in a job
<i>SetInformationJobObject</i>	Sets limits
<i>QueryInformationJobObject</i>	Retrieves information about the job, such as CPU time, page fault count, number of processes, list of process IDs, quotas or limits, and security limits

The following are some of the CPU-related and memory-related limits you can specify for a job:

- **Maximum number of active processes** Limits the number of concurrently existing processes in the job.
- **Jobwide user-mode CPU time limit** Limits the maximum amount of user-mode CPU time that the processes in the job can consume (including processes that have run and exited). Once this limit is reached, by default all the processes in the job will be terminated with an error code and no new processes can be created in the job (unless the limit is reset). The job object is signaled, so any threads waiting for the job will be released. You can change this default behavior with a call to *EndOfJobTimeAction*.
- **Per-process user-mode CPU time limit** Allows each process in the job to accumulate only a fixed maximum amount of user-mode CPU time. When the maximum is reached, the process terminates (with no chance to clean up).
- **Job scheduling class** Sets the length of the time slice (or quantum) for threads in processes in the job. This setting applies only to systems running with long, fixed quantum (the default for Windows Server systems). The value of the job-scheduling class determines the quantum as shown here:

Scheduling Class	Quantum Units
0	6
1	12
2	18
3	24
4	30
5	36
6	42
7	48
8	54
9	Infinite if real-time; 60 otherwise

- **Job processor affinity** Sets the processor affinity mask for each process in the job. (Individual threads can alter their affinity to any subset of the job affinity, but processes can't alter their process affinity setting.)
- **Job process priority class** Sets the priority class for each process in the job. Threads can't increase their priority relative to the class (as they normally can). Attempts to increase thread priority are ignored. (No error is returned on calls to *SetThreadPriority*, but the increase doesn't occur.)
- **Default working set minimum and maximum** Defines the specified working set minimum and maximum for each process in the job. (This setting isn't jobwide—each process has its own working set with the same minimum and maximum values.)
- **Process and job committed virtual memory limit** Defines the maximum amount of virtual address space that can be committed by either a single process or the entire job.

Jobs can also be set to queue an entry to an I/O completion port object, which other threads might be waiting for, with the Windows *GetQueuedCompletionStatus* function.

You can also place security limits on processes in a job. You can set a job so that each process runs under the same jobwide access token. You can then create a job to restrict processes from impersonating or creating processes that have access tokens that contain the local administrator's group. In addition, you can apply security filters so that when threads in processes contained in a job impersonate client threads, certain privileges and security IDs (SIDs) can be eliminated from the impersonation token.

Finally, you can also place user-interface limits on processes in a job. Such limits include being able to restrict processes from opening handles to windows owned by threads outside the job, reading and/or writing to the clipboard, and changing the many user-interface system parameters via the Windows *SystemParametersInfo* function.



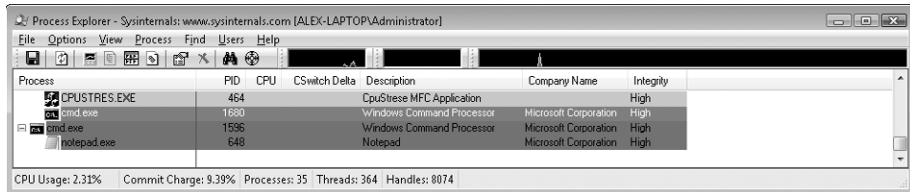
EXPERIMENT: Viewing the Job Object

You can view named job objects with the Performance tool. (See the Job Object and Job Object Details performance objects.) You can view unnamed jobs with the kernel debugger *!job* or *dt nt!_ejob* commands.

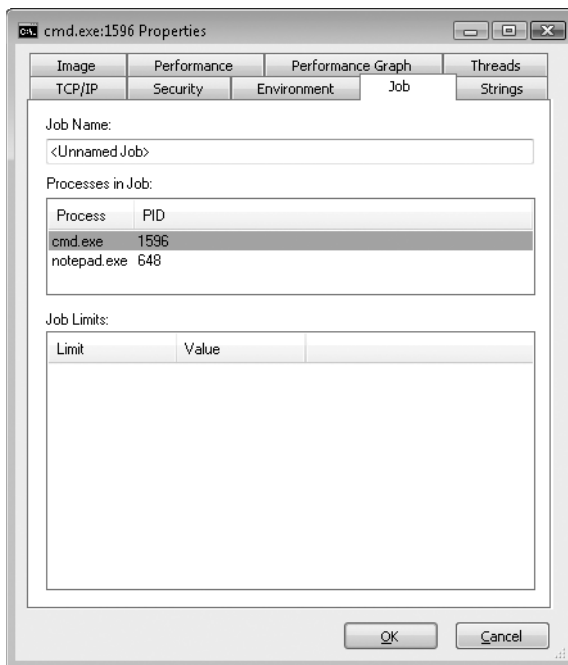
To see whether a process is associated with a job, you can use the kernel debugger *!process* command or Process Explorer. Follow these steps to create and view an unnamed job object:

1. From the command prompt, use the *runas* command to create a process running the command prompt (Cmd.exe). For example, type **runas /user:<domain>\<username> cmd**. You'll be prompted for your password. Enter your password, and a Command Prompt window will appear. The Windows service that executes *runas* commands creates an unnamed job to contain all processes (so that it can terminate these processes at logoff time).

2. From the command prompt, run Notepad.exe.
3. Then run Process Explorer and notice that the Cmd.exe and Notepad.exe processes are highlighted as part of a job. (You can configure the colors used to highlight processes that are members of a job by clicking Options, Configure Highlighting.) Here is a screen shot showing these two processes:



4. Double-click either the Cmd.exe or Notepad.exe process to bring up the process properties. You will see a Job tab in the process properties dialog box.
5. Click the Job tab to view the details about the job. In this case, there are no quotas associated with the job, but there are two member processes:



6. Now run the kernel debugger on the live system, display the process list with *!process*, and find the recently created process running *Cmd.exe*. Then display the process block by using *!process <process ID>*, find the address of the job object, and finally display the job object with the *!job* command. Here's some partial debugger output of these commands on a live system:

```

1kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
.
.
PROCESS 8567b758 SessionId: 0 Cid: 0fc4 Peb: 7ffdf000 ParentCid: 00b0
  DirBase: 1b3fb000 ObjectTable: e18dd7d0 HandleCount: 19.
  Image: Cmd.exe

PROCESS 856561a0 SessionId: 0 Cid: 0d70 Peb: 7ffdf000 ParentCid: 0fc4
  DirBase: 2e341000 ObjectTable: e19437c8 HandleCount: 16.
  Image: Notepad.exe

1kd> !process 0fc4
Searching for Process with Cid == fc4
PROCESS 8567b758 SessionId: 0 Cid: 0fc4 Peb: 7ffdf000 ParentCid: 00b0
  DirBase: 1b3fb000 ObjectTable: e18dd7d0 HandleCount: 19.
  Image: Cmd.exe
  BasePriority                8
  .
  .
  Job                        85557988

1kd> !job 85557988
Job at 85557988
  TotalPageFaultCount      0
  TotalProcesses           2
  ActiveProcesses          2
  TotalTerminatedProcesses 0
  LimitFlags               0
  MinimumWorkingSetSize    0
  MaximumWorkingSetSize    0
  ActiveProcessLimit       0
  PriorityClass             0
  UIRestrictionsClass      0
  SecurityLimitFlags       0
  Token                    00000000

```

7. Finally, use the *dt* command to display the job object and notice the additional fields shown about the job:

```

1kd> dt nt!_ejob 85557988
nt!_EJOB
+0x000 Event           : _KEVENT
+0x010 JobLinks        : _LIST_ENTRY [ 0x81d09478 - 0x87f55030 ]
+0x018 ProcessListHead : _LIST_ENTRY [ 0x87a08dd4 - 0x8679284c ]
+0x020 JobLock         : _ERESOURCE
+0x058 TotalUserTime   : _LARGE_INTEGER 0x0

```

```

+0x060 TotalKernelTime : _LARGE_INTEGER 0x0
+0x068 ThisPeriodTotalUserTime : _LARGE_INTEGER 0x0
+0x070 ThisPeriodTotalKernelTime : _LARGE_INTEGER 0x0
+0x078 TotalPageFaultCount : 0
+0x07c TotalProcesses : 2
+0x080 ActiveProcesses : 2
+0x084 TotalTerminatedProcesses : 0
+0x088 PerProcessUserTimeLimit : _LARGE_INTEGER 0x0
+0x090 PerJobUserTimeLimit : _LARGE_INTEGER 0x0
+0x098 LimitFlags : 0
+0x09c MinimumWorkingSetSize : 0
+0x0a0 MaximumWorkingSetSize : 0
+0x0a4 ActiveProcessLimit : 0
+0x0a8 Affinity : 0
+0x0ac PriorityClass : 0 ''
+0x0b0 AccessState : (null)
+0x0b4 UIRestrictionsClass : 0
+0x0b8 EndOfJobTimeAction : 0
+0x0bc CompletionPort : 0x87e3d2e8
+0x0c0 CompletionKey : 0x07a89508
+0x0c4 SessionId : 1
+0x0c8 SchedulingClass : 5
+0x0d0 ReadOperationCount : 0
+0x0d8 WriteOperationCount : 0
+0x0e0 OtherOperationCount : 0
+0x0e8 ReadTransferCount : 0
+0x0f0 WriteTransferCount : 0
+0x0f8 OtherTransferCount : 0
+0x100 ProcessMemoryLimit : 0
+0x104 JobMemoryLimit : 0
+0x108 PeakProcessMemoryUsed : 0x19e
+0x10c PeakJobMemoryUsed : 0x2ed
+0x110 CurrentJobMemoryUsed : 0x2ed
+0x114 MemoryLimitsLock : _EX_PUSH_LOCK
+0x118 JobSetLinks : _LIST_ENTRY [ 0x8575cff0 - 0x8575cff0 ]
+0x120 MemberLevel : 0
+0x124 JobFlags : 0

```

Conclusion

In this chapter, we've examined the structure of processes and threads and jobs, seen how they are created, and looked at how Windows decides which threads should run and for how long.

In the next chapter we'll look at a part of the system that's received more attention in the last few years than ever before, Windows security.

Index

Symbols and Numbers

! commands. *See* kernel debugger commands

!analyze command, 1149

!apic command, 91

!devhandles command, 155–156

!devstack command, 579

!drvobj command, 671

!exqueue command, 200

!gflag command, 202

!handle command, 133, 154

!htrace command, 156

!idt command, 88–89, 102

!ioapic command, 91

!irp command, 1145–1147

!irq command, 95

!locks command, 191

!numa command, 437

!object command, 144, 489, 552–553

!obtrace command, 156

!pccr command, 62–64, 95

!pic command, 91

!process command

- data displayed by, 345
- identifying processes with, 144
- job process associations, displaying, 447–449
- process page directory physical addresses, 765
- running processes, viewing all, 1144

!qlocks command, 176

!ready command, 391

!reg dump pool, 273

!reg findkcb, 278

!reg hivelist, 274

!reg kcb, 278

!reg openkeys, 278

!thread command, 376, 569

!timer command, 109

!vm command

- memory data listed by, 703
- memory exhaustion, checking for, 1145

!wsl command, 830

.NET Framework components, 3

32-bit emulation on 64-bit Windows. *See* Wow64

3DES (Triple-DES), 990–991

64-bit Windows versions (x64)

- address space layouts, 745–746
- address spaces, 15
- CMPXCHG16B processor instruction issue, 749–750
- execution prevention, 714
- Itanium versions. *See* IA64 CPUs
- kernel mode code signing, 17
- memory and processor support, table of, 44–45
- physical memory limits for, 818–819
- process size limitations, 699
- singly linked lists issue, 749–751
- technologies supported by, 38
- UAC virtualization disabled for applications, 522
- virtual addressing limitations, 749–751
- x64 address translation, 773

A

AAM (Admin Approval Mode), 528–533

AcceptEx function, 1008

access checks

- AccessCheck. *See* AccessCheck tool
- default security, 460
- discretionary. *See* discretionary access checks
- events causing, 459–460
- file object security, 460
- integrity levels, 464–473
- mandatory policies, 473
- masks. *See* access masks
- mechanics of, 459–461
- ObCheckObjectAccess function for, 459–460
- ordering of ACEs, 497
- referencing events causing, 460
- SeAccessCheck function, 460
- SIDs for, 461–464
- tokens. *See* tokens
- user-mode checks, 461

access control

- access-denied errors, 909
- ACEs. *See* ACEs (access control entries)
- ACLs. *See* ACLs (access control lists)
- checks. *See* access checks
- desired access rights, 157, 1158
- determining for objects, 492–501
- discretionary. *See* discretionary access control
- effective permissions, 500
- entries. *See* ACEs (access control entries)
- lists. *See* ACLs (access control lists)
- mandatory integrity control, 22
- masks. *See* access masks
- object access checks. *See* access checks
- owner rights of objects, 495–497
- privileged, 22
- protected processes, 346–348
- security descriptors for. *See* security descriptors
- stop codes from violations, 1122
- tokens. *See* tokens
- types of, 22
- user logon test for, 518

access faults, page. *See* page faults

access masks

- algorithm for applying, 496–497
- DACL algorithm, role in, 495
- defined, 486–487
- granted-access masks, 495
- protected processes, table for, 347

access tokens. *See* tokens

AccessCheck tool

- object access, displaying, 158
- object integrity levels, viewing, 472–473
- security descriptors, viewing, 488

account rights

- defined, 501
- displaying, 501
- functions for changing, 503
- mechanism for, 502–503
- table of user rights, 503

accounts

- HKLM\SAM subkey, 258
- local service account, 288, 291
- local system. *See* local system accounts
- network service account, 290–291
- privileges of. *See* privileges, account
- rights of. *See* account rights
- service applications, for, 288–291
- trusted facility management, 453

- ACEs (access control entries)
 - access masks, 496–497
 - ACLs, relation to, 1153
 - callback ACEs, 486–487
 - DACLs, types appearing in, 486–487
 - discretionary access checks using, 495
 - effective permissions, 500
 - GUI permissions editors issue, 498–499
 - inheritance flags, 487
 - inheritance mechanics, 490–492
 - order of, results from, 497–499
 - security descriptor interactions with, 485–488
 - SIDs for, 462
 - system audit types, 488, 1176
- ACLs (access control lists)
 - accumulation of ACE access rights, 487
 - assignment to new objects, 490–491
 - composition of, 486–487
 - defined, 1153
 - discretionary. *See* DACLs (discretionary access control lists)
 - displaying for objects, 157–158
 - entries in. *See* ACEs (access control entries)
 - inheritance mechanics, 490–492
 - section objects, of, 713
 - service accounts, isolation issues with, 294–300
 - system. *See* SACLs (system access control lists)
- ACPI (Advanced Configuration and Power Interface), 636–638
- act as part of operating system privilege, 508, 510
- activation stacks, thread, 381
- Active Directory
 - ACE types for, 487
 - ADSI service interfaces, 1067
 - APIs for accessing, 1066–1067
 - authentication against, 518
 - connection security rules management, 1052
 - defined, 454–455
 - Directory Services Restore mode, 1102
 - Distributed File System (DFS) service, 1069–1071
 - ID lookups, 1067
 - IPSec for policies, 1050
 - MAPI support, 1067
 - SAM (Security Accounts Manager) service, 1067
 - server name publishing, 1016
 - shadow copies, enabling, 688–689
 - Windows NT 4 networking APIs, 1067
 - Winsock support for, 1009, 1011
- Active page state, 804
- active partitions, 653–654
- active processor masks, 434
- add-device routines, 548, 607, 1153
- address spaces
 - 64-bit, 15, 745–751
 - ASLR for user address space, 757–761
 - AWE. *See* AWE (Address Windowing Extension)
 - committing pages, 706–708
 - CreateProcess function, setup by, 355–357
 - granularity of allocations, 708
 - hypervisor managed, 238–239
 - kernel mode, 16
 - large address space aware images, 699, 739, 746
 - layouts of. *See* virtual address space layouts
 - marshalling areas, 1167
 - pools. *See* pools, memory
 - private, 712
 - processes marked for extra space, 14–15
 - reserving pages, 706–708
 - section objects for mapping. *See* section objects
 - shareable memory, 758
 - sizes of, 14–15
 - system. *See* system address spaces
 - user mode, 16
 - viewing with Vmmap utility, 758–759
 - virtual. *See* virtual address spaces
- Wow64, 211
- x86 address space layouts, 737–740
- address translation
 - byte index fields, 768
 - caching of results, 768–769
 - defined, 761
 - IA64, 772
 - PAE, 769–771
 - page directory mechanics, 764–766
 - page table mechanics, 766–767
 - PTE mechanics, 762–763, 766–767
 - steps for, 764
 - translation look-aside buffers, 767–769
 - x64, 773
 - x86 mechanics overview, 762–764
- Address Windowing Extension. *See* AWE (Address Windowing Extension)
- Admin Approval Mode (AAM), 528–533
- administrative rights, 473–474, 528–533
- Advanced Configuration and Power Interface (ACPI), 636–638
- Advanced Encryption Standard (AES), 990–991
- Advanced Local Procedure Calls. *See* ALPCs (Advanced Local Procedure Calls)
- AES (Advanced Encryption Standard), 990–991
- AFD (Ancillary Function Driver), 1012
- affinity manager, LFH, 733
- affinity masks
 - defined, 1153
 - drawbacks of, 440
 - functions for, 395
 - image headers, setting in, 439
 - job processor affinity setting, 447
 - job wide, 439
 - NUMA functions, 438
 - SetProcessAffinityMask function, 438
 - SetThreadAffinityMask function, 438
 - setting for processes, 439
 - thread affinity masks, 438–439
 - uniprocessor flags, 439
- aging, 1153
- AIS (application information service), 529–531
- alertable wait states, 1153–1154
- AllocateUserPhysicalPages function, 720
- allocation. *See* memory manager
- allocation granularity values, 708, 1153–1154
- Allow logon rights, table of, 503
- ALPCs (Advanced Local Procedure Calls)
 - client-server connections, 205–206
 - components using, 202–203
 - defined, 202, 1153
 - LPCs, 203
 - message exchange methods, 204–205
 - port objects, 205–206, 1170
 - role in executive, 60
 - RPC use of, 202–204, 1017–1018
 - securing, 206
 - security component use of, 457
 - UMDF support, 203
 - viewing, 203–204
 - Winlogon to Lsass, 515
- altitudes, 170, 280, 1154
- AMD64, 38

- ANSI string parameters, 23–24
- APCs (asynchronous procedure calls)
 - APC level IRQL, 97
 - defined, 1154
 - device driver use of, 113
 - disabling, 112
 - executive use of, 112
 - I/O completion with, 575–577
 - interrupts, 112–114
 - KTHREAD field for, 373
 - modes with, 112
 - objects, 112
 - queues, 112, 1154
 - thread control with, 112
 - types of, 112–113
 - user mode, 113
 - wait queue reordering by, 114
- APICs (Advanced Programmable Interrupt Controllers)
 - BCD option for maximum processor number, 1080
 - hypervisors with, 238
 - IRQ to IRQL mapping, 96
 - mechanics of, 89–91
- APIs (application programming interfaces)
 - AuthZ, 500–501
 - DeviceIoControlFile API, 214–215
 - HTTP Server API, 1019–1021
 - Hyper-V API library, 232
 - I/O subsystem API, 1162
 - job related, table of, 446
 - scheduling related, 395–396
 - translating DLL API functions to service calls, 54
 - Win32. *See* Win32 API
 - Windows. *See* Windows API
 - Windows Networking API, 1033–1036
- application information service (AIS), 529–531
- application layer, OSI, 1002
- Application Verifier tool, 604
- applications
 - compatibility shims, 522–523
 - crashes following new installations of, 1124–1125
 - foreground, 1161
 - installing, UAC issues with. *See* UAC (User Account Control)
 - legacy, well-known installers, 1111–1112
 - native applications, 1168
 - registries, reading during startup, 250
 - services. *See* service applications
 - Software Restriction Policies controls for, 533–535
 - user, 36
- architecture
 - applications in separate mode, 34
 - assembly language components, 38–39
 - boot process abstraction, 1073
 - C language, 35
 - checked build version, 47–49
 - core components, file names of, 37–38
 - design goals of Windows NT, 33–34
 - device drivers, 68–69
 - diagram of, 35–36, 49
 - executive component of, 37
 - fast LPC, 55
 - HAL (hardware abstraction layer), 37–39
 - hardware isolation by kernel, 64–65
 - Hyper-V, 230–231, 235–236
 - kernel mode, 34–35, 37
 - layered design, 38
 - microkernel-based operating systems, 34–35
 - monolithic operating systems, 34
 - networking, overview of, 1001
 - Ntdll.dll, 57–58
 - object-oriented kernel code, 35
 - operating system model, 34–35
 - portability, 38–39
 - scalability of, 43
 - server versions compared to client versions, 43–47
 - subsystem DLLs, 36–37
 - system processes. *See* system processes
 - user-mode components, 36
 - windowing and graphics system, 37
 - Windows Driver Model. *See* WDM (Windows Driver Model) drivers
 - Windows subsystem. *See* Windows subsystem
 - WMI, 318–319
- ARIES (Algorithm for Recovery and Isolation Exploiting Semantics), 910
- ASLR (Address Space Layout Randomization)
 - disabling, 759
 - heap randomization, 760
 - image randomization by, 759–760
 - mechanics of, 759
 - purpose of, 757
 - stack randomization, 760
 - viewing processes for support status, 761
- ASMP (asymmetric multiprocessing), 39, 1154
- ASR (Automated System Recovery). *See* Complete PC Restore
- assemblies, 364–366
- assembly language components of Windows, 38–39
- asynchronous I/O
 - cancellations, 588
 - defined, 1154
 - Driver Verifier completion simulation, 606
 - mechanics of, 563
 - requests to layered drivers, 578–587
- asynchronous read-ahead with history, 876, 1154
- asynchronous RPC, 1015–1016
- Ataport.sys, 647
- ATL (Active Template Library) DEP compatibility, 716
- atomic execution, 197
- atomic transactions, 918–919, 1154. *See also* NTFS recovery support
- Attachment Execution Service, 921
- attribute lists, NTFS, 1154–1155
- attributes, file
 - attribute definition tables, 940–941
 - attribute streams, 942
 - resident attributes, 948–951
 - table of, 943–944
 - type codes, 944
 - unnamed data attribute, 942
- attributes, object. *See* object attributes
- audio, 346–347, 430–434
- auditing, security
 - audit events, 511
 - defined, 1173–1174
 - local security policy, 511
 - Lsass support for, 511
 - privilege for, 505
 - privileges related to, 511
 - queuing of messages for, 512
 - TCSEC rating criteria for, 452
- authentication
 - AuthIP, 1051–1052
 - calls to, 514
 - credential providers, 79
 - data origin authentication, 1050
 - IPSec-supported methods, 1051
 - Kerberos. *See* Kerberos authentication
 - levels of, 1017
 - LSASS. *See* LSASS (local security authentication server process)
 - MSV1_0, 513, 517–518
 - packages (DLLs), 455, 513, 1155
 - RPC, for, 1017
 - user logons, packages for, 517

authority, P2P, 1040
 AuthZ API, 500–501
 Autochk, 1094
 automatic working set trimming, 1155
 auto-reset events, 195
 Autoruns tool, 1011, 1100
 AWE (Address Windowing Extension)
 defined, 1153
 mechanics of, 719–721
 purpose of, 15

B

Background Intelligent Transfer Service (BITS), 1030
 backups, 506, 692–693, 999
 bad cluster files, 940–941, 1155
 Bad page state, 804
 bad-cluster remapping, 1155
 balance set manager
 defined, 1155
 mechanics of, 427–430
 memory manager, as a component of, 700
 system thread, as a, 76
 working set expansion and trimming by, 831–832
 bandwidth reservation for I/O, 603–604
 base LSNs, 973
 base priority determination, 393
 BaseNamedObjects namespace, 167
 basic disks
 defined, 652–653, 1155
 enumeration of, 655
 GPT partitioning, 654
 MBR partitioning, 653–654
 types of devices limited to, 653
 volume manager with, 655
 basic volumes, mounting, 667–668
 .bat files, process creation for, 353
 BCD (Boot Configuration Database)
 bad physical pages list, 1079
 BCDEdit utility, 256
 Bootmgr file read of, 1077
 debugging options, 1079–1080, 1148–1149
 default selection by Bootmgr, 1084
 editing tool for, 1078
 entry storage, 255
 HKEY_LOCAL_MACHINE\BCD, 255–257
 hypervisor options, 1081
 multiple boot selection entries in, 1078
 optional argument for Bootmgr, 1078
 options for boot applications, table of, 1079–1080
 preparation by Setup, 1077
 Registry Editor for editing, 256–257
 troubleshooting configuration issues, 1109–1110
 watermarked desktop option, 1080
 Winload options, 1080–1084
 BFE (base filtering engine), 1047
 binding, network, 1064–1065
 bins, registry, 269–272
 BIOS (basic I/O system)
 BitLocker boot process, 683
 boot process files, table of, 1074–1075
 MBR reads by, 653
 setup process for, 1073–1077
 x64 vs. x86 boots, 1073
 BitLocker
 architecture of, 677–678
 boot process, 683–684
 cryptography algorithms of, 679
 defined, 1155
 Elephant diffuser algorithm, 681, 686
 foreign volumes, 685
 FVE (full volume encryption), 686
 FVEK with, 679
 Group Policy for, 687
 key escrow services, 687
 keys, 679–681
 major protection capabilities of, 677
 management of, 687
 passwords for recovery, 684–685
 PCRs (platform configuration registers), 681–683
 programmatic access to, 687
 purpose of, 677
 recoveries, 684–685
 sealing VMKs, 681–682
 sector encryption keys, 686
 system change recoveries, 685
 TPM for, 677, 681–683, 687
 unsealing VMKs, 684
 VMKs, 679–681
 bitmap files (\$BitMap), 940, 1155
 BITS (Background Intelligent Transfer Service), 1030
 B-level security, 453
 BLFs (base log files), 913–914
 blocks, hive, 269
 blue screen of death
 causes of, 1119–1120
 debugging causes. *See* crash dump analysis
 example, 1120
 screen saver simulation of, 1152
 stop code interpretation. *See* stop codes
 Bluetooth, 616, 1011
 boot code, defined, 1155
 boot loader
 Bootmgr loads, 1084
 image system addresses, 752
 parameter block structure, 1088
 boot process. *See also* EFI (Extensible Firmware Interface)
 architecture, abstraction of, 1073
 automatic process startups, 1100
 BCD file reads, 1077
 BIOS role in. *See* BIOS (basic I/O system)
 BitLocker boot process, 683–684
 boot disk based, recovery options, 1106
 Boot Manager. *See* Bootmgr (Windows Boot Manager)
 boot start driver issues, 1106
 boot status files, 1108
 boot-selection menu
 presentation, 1078
 Bootvid initialization, 1089
 code, defined, 1155
 Csrss, starting and initialization, 1095–1096
 disk access during boots, 646
 DPC stack allocation, 1090
 driver enumeration and symbol loads, 1089
 Driver Verifier initialization, 1089
 EFI boot process, 1086–1087
 executive Phase I initialization, 1089
 HAL initiation steps, 1088–1089
 hard disk setup for, 1073–1074
 HKLM\SYSTEM subkey, 259
 I/O manager initializations, 1091–1093
 InitBootProcessor responsibilities, 1088–1090
 KiInitializeKernel responsibilities, 1088–1099
 last known good. *See* last known good control sets
 logging in safe mode, 1104–1105, 1114
 MBRs, control transfer to, 1076. *See also* MBR (Master Boot Record)
 memory manager, initializations by, 1089, 1091
 NTFS recoveries on, 981
 Ntoskrnl phases, 1088–1093
 object manager initializations, 1090
 page file reads, 781
 Phase 0 steps, 1088–1090

- Phase 1 steps, 1090–1093
- prefetching, 823–827
- process manager initializations, 1090, 1093
- ReadyBoot, 1099
- registry fixes during, 279
- registry initializations, 1095
- registry, reading of, 250
- safe mode, 1101
- SCM with, 300, 304, 1097
- screen crashes during, 1112–1114
- security reference monitor initializations, 1090
- Session Manager initialization, 1093–1096
- system hive corruption, 1112
- tracing with Process Monitor, 265–266
- troubleshooting, 1106–1114
- Wininit steps of, 1097
- Winload role in, 1084–1086
- Winlogon steps for, 1097–1098
- x64 vs. x86, 1073
- boot sectors
 - creation of, 1076
 - defined, 1155
 - format dependence of, 1077
 - FSD use of, 896
 - system partition, relation to, 1076
 - troubleshooting corruption issues, 1109
- boot volumes, 1155
- Bootmgr (Windows Boot Manager)
 - BCD optional arguments for, 1078
 - BIOS-interfacing functions, 1077
 - boot loader loads, 1084
 - boot-selection menu presentation, 1078
 - crashes, detecting on reboot, 1124
 - defined, 1074
 - hibernation resumes, 1078
 - paging enabling by, 1077
 - protected mode, switching to, 1077
 - reading in of file, 1077
 - real mode execution, 1077
 - screen clears by, 1078
 - storage read role, 646
- Bootvid initialization, 1089
- Bootvid.dll, 67
- boundary descriptors, 164
- BSD Sockets, 1006
- buckets, 732–733
- buffer overflow errors, file systems, 910
- buffers, I/O
 - buffer management types, 570–571
 - Direct I/O, 570

- driver use of, 571
- FILE_FLAG_NO_BUFFERING flag, 875
- IRP references for, 571
- mechanics of, 570–571
- Neither I/O, 571
- overruns, 1140–1142
- pool corruption, 1140–1142
- bugcheck codes. *See* stop codes
- bus drivers
 - defined, 69, 542–543, 1155
 - device enumeration role, 624–628
 - HAL acting as, 624–625
 - instance IDs, 630
 - IRQ mapping IRQs, 96
 - Multipath Bus Driver, 649
 - Plug and Play support, 621
 - responsibilities of, 542–543
 - Root virtual bus driver, 624–625
 - Bypass Traverse Checking privilege, 505, 509

C

- C language, 35, 38–39
- C2 security ratings, 451–452
- cache manager
 - !defwrites debugger command, 886
 - !filecache debugger command, 857
 - architecture of, 850
 - boots, initialization during, 1091
 - buffer management, 871
 - CcDbg debugging variables, 861–862
 - client/server example, 858
 - coherency of data, 850–852
 - copying method, 869
 - data structure overview, 859–860
 - defined, 849, 1155
 - determining if files are in the cache, 862–863
 - dirty page threshold, 885–886
 - disabling lazy writing, 883
 - DMA caching, 872
 - fast I/O, 873–875
 - fast teardown queues, 886–887
 - file I/O not handled by, 875
 - file mapping objects with, 850
 - file system drivers, 868, 904–905
 - file system interface overview, 868
 - flushing mapped files, 883–884
 - flushing of cache pages, 877
 - global look-aside lists, 887
 - hint support, 849
 - I/O operation interactions, 868
 - initialization for files not in cache, 868
 - intelligent read-ahead, 875–876, 878–883, 1163
 - key features of, 849
 - lazy writing, 877–883, 906, 1165
 - least recently used VACBs, 861
 - logical sequence numbers (LSNs), 853–854
 - mapped files, 566, 710
 - mapped page writer thread of, 881
 - mapping interfaces, 870–871
 - memory manager use by, 849–850
 - metadata, functions for finding locations of, 870
 - modified page lists, 858–859
 - NTFS cache flushing operations, 977
 - NTFS file system driver with, 935
 - per-file data structures, 862–868
 - per-processor look-aside lists, 887
 - physical memory size, 858–859
 - pinning interfaces, 870–871
 - post tick queues, 887
 - private cache map structures, 862–867
 - private cache maps, 1171
 - purpose of, 59
 - queue types used by, 886–887
 - read activity, examining, 869–870
 - read-ahead. *See* intelligent read-ahead
 - recoverable file system support, 853–854
 - remote file system driver oplock protocol, 898–899
 - section objects, 850, 862
 - shared cache map structures, 862–867, 1174
 - size computation, 855–859
 - size system variables, table of, 856
 - standby lists, 858–859
 - stream-based caching, 852
 - system space to application buffer issue, 869
 - system threads of, 886–887
 - systemwide data structures, 860–862
 - VACBs. *See* VACBs (virtual address control blocks)
 - viewing cache maps, 866–867
 - viewing file system activity of, 878–883
 - views, 854–855, 862–863
 - virtual address space model, 850
 - virtual block caching, 852
 - virtual memory management, 854–855
 - virtual size, 855–856
 - working set size, 856–857

- cache manager, *continued*
 - write throttling, 885–886
 - write-back caching, 877–883
 - write-behinds, 878–882
 - write-through caching, 883
- cache-aware pushlocks, 193
- caches, CPU-specific code variants, 64
- caches, HTTP, 1020–1021
- call managers, NDIS, 1057
- callbacks
 - bugcheck callbacks, 1120
 - callback ACEs, 486–487
 - driver, 170
 - Evtlo callback routines, 607
 - fast dispatch routines, 549
 - KMDF, 607, 614
 - reason callbacks for crash dumps, 1120
 - Wow64, 212
- callouts, WFP, 1004, 1047
- cancel I/O routines, 549, 587–592
- case sensitivity of object names, 163
- .cat files, 538–539
- catalog files, assembly, 365
- catalog files, device driver, 635–636
- CBS (Component Based Servicing), subkey for, 257–258
- CC (Common Criteria), 453
- CDFS (CD-ROM file system), 890–891, 896
- CD-ROM drives, 890–891, 896
- cells, registry
 - bins, 270–271
 - configuration management of, 272–276
 - data types of, 270
 - defined, 269–270
 - examining, 271
 - indexes, 271
 - maps, 272–276
 - organization of, 271
- certificates, 533, 1051
- change journal files, 941, 956–959, 1155–1156
- change logging, NTFS, 927–928
- charge notification, 834
- checked build version, 47–49, 1156
- checkpoint records, 1156
- child partitions, hypervisor, 230, 232–234
- chimney offloading, 1054
- Chkdsk utility
 - Autochk run during boots, 1094
 - bad sector fixes by, 986–988
 - NTFS preventing need for, 974
 - system file corruption, fixing, 1110
- chunked transfer encoding, 1019
- Ci.dll, 67
- CIFS (Common Internet File System) protocol, 898
- CIM (Common Information Model) CIMOM (CIM Object Manager), 319
 - class model of, 320–321
 - common model, 1156
 - core model, 1157
 - Event Log provider use of, 321–322
 - MOF with, 322
 - specification for, 319–320
- class drivers, 543, 1156
- classifier, P2P, 1040
- CLFS (Common Log File System)
 - ARIES, 910
 - base log files, 913–914
 - Clfs.sys, 67
 - common logs, 911
 - containers, 913–914
 - dedicated logs, 911–913
 - dump counts, 913–914
 - flush queues, 912–913
 - identifiers of containers, 914
 - layout of logs, 913–914
 - log blocks, 915
 - log space reservation, 911
 - LSNs (log sequence numbers), 914–915, 917
 - management policies, 918
 - marshalling, 910–911
 - multiplexed logs, 911–913
 - owner pages, 912, 915–916
 - pages, 915
 - physical vs. virtual clients, 911
 - purpose of, 910
 - regions, 915
 - sector signatures, 915
 - torn writes, 915
 - TxF log files, 970
 - types of logs, 911–913
 - virtual logs, 916
 - virtual LSNs, 916–917
- client versions of Windows
 - architecture compared to server versions, 43–47
 - list of, 43–44
 - optimization compared to servers, 46
 - physical memory limits for, 818–822
- client/server model
 - impersonation with, 480–483
 - LogonUser function with, 481–483
 - remote file system drivers for, 897–899
- Client/Server Run-Time Subsystem. *See* Csrss.exe
- clock algorithm, 827–828, 1156
- clock interrupt handler, 1156
- clock interval timer frequencies, 406–407
- clock, system, 97
- Clockres, 407
- clone shadow copies, 688
- C-LOOK algorithm, 647
- close object method, 148
- clouds, P2P, 1040
- CLR. *See* Common Language Runtime (CLR)
- clustered page faults, 779–780
- clustering, memory manager, 1156
- clusters, file
 - bad clusters, 940–941, 985–988, 1155
 - defined, 889, 1156
 - dynamic bad-cluster remapping, 923
 - factors, defined, 1156
 - FAT size variations, 891–894
 - LCNs (logical cluster numbers), 938–940, 950–951
 - NTFS, 937–938
 - sectors, relationship to, 937–938
 - size of, setting, 937
 - sizes of, 895
 - VCNs (virtual cluster numbers), 938–940, 950–951
- .cmd files, process creation for, 353
- CMFXCHG16B processor instruction, 749–750
- code corruption from drivers, 1143–1144
- code integrity, 246–247, 1156
- collided page faults, 779, 1156
- COM (Component Object Model) distributed. *See* DCOM (Distributed Component Object Model)
- HKEY_CLASSES_ROOT class registries, 255
- WMI COM API, 319
- comctl32.dll, 365
- command prompts
 - safe mode with, 1101, 1104
 - troubleshooting with, 1106
- commit charge notification, 834
- commit charges, 782, 833–834
- committed pages, 706–708, 710, 1156
- common controls DLL, 365
- Common Criteria (CC), 453
- Common Internet File System (CIFS) protocol, 898
- Common Language Runtime (CLR), 3
- Common Log File System. *See* CLFS (Common Log File System)
- common logging file system driver (Clfs.sys), 67

- common model, CIM, 1156
- compatible IDs, 634
- complete memory dumps, 1126–1128, 1156
- Complete PC Restore, 1106, 1110
- completion ports. *See* I/O completion ports
- Compound TCP (CTCP), 1044
- compression support, NTFS
 - file, directory, and volume basis, 951–952
 - nonsparse data compression, 954–956
 - sparse data compression, 952–954
 - sparse files, 956
 - types of compression, 927
- concurrency values, 593–595
- CoNDIS (connection-oriented NDIS), 1055, 1057–1060
- condition variables, 195–196
- confidentiality, IPsec for, 1050
- configuration manager
 - altitude, 280
 - ASCII vs. Unicode storage, 281
 - bins, 269–272
 - blocks, 269
 - cell indexes, 271
 - cells, 269–272
 - defined, 266, 1157
 - dirty sector arrays, 279
 - hive syncs, 279
 - hives, loading of, 266
 - kernel handle tables, 268
 - key control blocks, 276–278, 281
 - key object type, 276
 - lazy write operations, 279
 - log hives, 278–279
 - memory mapping of hives, 272–276
 - namespace integration, 276
 - new keys created by applications, 277
 - operation flow control, 277–278
 - performance optimizations by, 280–281
 - registry filtering, 280
 - registry initialization during boots, 1095
 - search methods of, 272
 - self-healing operations, 279
 - symbolic links, 268
 - Windows executive, as component of, 59
- ConnectEx function, 1009
- connecting an interrupt object, 105–106
- connections, network, 1007–1008
- consent elevations, 529
- consistency check stop codes, 1122–1123
- console windows, DLLs supporting, 54
- consumers, ETW, 207
- container objects, 1157
- containers, log system, 913–914
- context switching
 - address space sharing within processes, 764–765
 - address translation after, 764
 - architecture dependence of, 64–65
 - defined, 1157
 - mechanics of, 418
 - minimizing, goal of, 592–593
 - multiprocessor threads, 435
 - quantum ends allowing for, 407
 - thread scheduling requiring, 392
- control areas of section objects, 796–798
- control objects, 62, 1157
- control sets, 1113–1114
- controllers, ETW, 207
- cookies, stack, 717–718
- copy-on-write page protection, 718–719
- copy-on-write shadow copies, 688
- core model, CIM, 1157
- counters
 - descriptions of, viewing, 25
 - disk performance, 665–666
 - performance. *See* performance counters
- CPs. *See* credential providers (CPs)
- CPU Stress tool
 - CPU starvation boosts, watching, 428–430
 - priority boosts, viewing, 423–425
- CPU utilization, 9, 382
- CPUs
 - dynamic processor addition, 238
 - GetLogicalProcessorInformation function, 395
 - ideal processors, 1162
 - Itanium. *See* IA64 CPUs
 - memory cache entries, 815
 - memory protection built into, 712
 - mode-related performance counters, 17–19
 - multiple-core, 40. *See also* SMP (symmetric multiprocessing)
 - portability between architectures, 38
 - privilege levels of, 16
 - rate limits on, 444–445
 - starvation, priority level boosts for, 427–430
 - unified kernel, 41
 - utilization. *See* CPU utilization
 - virtual, 237–238
- CR3 register, 764–765
- crash analysis server, Microsoft, 121
- crash dump analysis
 - !analyze command, 1149
 - !irp command, 1145–1147, 1150
 - !locks command, 1150
 - !process command, 1144, 1149–1150
 - !stacks command, 1150
 - !thread command, 1149–1150
 - !vm command, 1145
 - advanced, using manual commands, 1144–1145
 - auxiliary computer dump captures, 1130, 1148–1149
 - basic analysis overview, 1134
 - booting into debugging mode, 1148–1149
 - bucket ID entries, 1134
 - causes of crashes, 1119
 - checksums, pre-write, 1130
 - code corruption from drivers, 1143–1144
 - complete memory dumps, 1126–1128
 - deadlock detection, 1147
 - dedicated dump files, 1131
 - default generation of crash dumps, 1125
 - defined, 1119
 - device driver secondary dumps, 1127
 - dps (dump pointer with symbols) command, 1146–1147
 - Drive Verifier as an aid to, 1139
 - driver information, viewing, 1138–1139
 - exceptions, kernel-mode, 1119
 - fail fast policy of Windows, 1119
 - generation of dumps, mechanics of, 1130–1131
 - hung systems, troubleshooting, 1147–1150
 - IRP analysis, 1146
 - Kd for, 1134
 - KeBugCheckEx function, 1120–1121
 - kernel memory dumps, 1126–1129
 - !m kv command, 1144
 - manually crashing hung systems, 1148
 - miniport drivers for dumps, 1130
 - missing dumps, 1150–1152
 - Notmyfault utility, 1134–1139
 - OCA (Online Crash Analysis), 1133–1134
 - parameter information with stop codes, 1120–1121
 - pool corruption, 1140–1142
 - recursive faults, 1151

- crash dump analysis, *continued*
 - reporting settings, 1131–1132
 - sending reports to Microsoft, 1132
 - small memory dumps, 1127–1128
 - stack traces of executing threads, 1138, 1149–1150
 - stack trashes, 1145–1147
 - Startup And Recovery settings, 1125
 - stop codes, 1120–1123
 - troubleshooting crashes, 1124–1125
 - WER for. *See* WER (Windows Error Reporting)
 - WinDbg for, 1134, 1137
 - crash dumps
 - analysis of. *See* crash dump analysis
 - defined, 1157
 - system space for, 737
 - create global object privilege, 168
 - CreateEventEx for access checks, 159
 - CreateFile function
 - asynchronous I/O flag for, 563
 - file system driver operations after calling, 902–906
 - mailslots with, 1024
 - named pipes with, 1024
 - read-ahead flags of, 876
 - write-through caching flag, 883
 - CreateProcess functions
 - address space setup steps, 357
 - debugging notifications, 363
 - decision tree for image types, table of, 353
 - elevation check, 362
 - EPROCESS block setup, 354–355
 - executive objects vs. subsystem process creation, 349
 - executive process object creation stage, 354–359
 - executive process object finalization, 359
 - handle table initialization, 355
 - initial process address space creation, 355–356
 - initialization in new process context stage, 363–370
 - job binding, 359
 - LdrpInitializeProcess, 369
 - NtCreateUserProcess with, 349, 351–352, 360
 - opening of executable images, 351–354
 - parameter validation and conversion step, 350–351
 - parent process inheritance, 354–355
 - PEB setup, 357–358
 - performance options, applying, 355
 - prefetch checks, 363, 369
 - process creation overview, 348–349
 - PspAllocateProcess, 354–359
 - PspInsertProcess, 359
 - PspUserThreadStartup, 363–364
 - registry checks, 352, 367
 - section object creation, 351–352
 - single CPU flag, 358
 - stages, list of, 349
 - support images, 352
 - SxS setup, 361
 - table of, 344
 - thread execution start stage, 362
 - thread initialization stage, 359–360
 - thread seed initialization, 356
 - thread stack creation, 360
 - user vs. kernel mode, viewing, 367–368
 - validation checks, 361
 - viewing startup process, 366–370
 - virtualization check, 362
 - Windows subsystem operations stage, 360–362
 - CreateThread function, 117–118, 380–381
 - Creator Group ID group, 462
 - Creator Owner ID group, 462
 - creator process IDs, 5–6
 - credential manger, 508
 - credential providers (CPs)
 - default and third-party, 514
 - defined, 455, 1157
 - logon role of, 79
 - user logon role, 516–517
 - critical code sections
 - advantages of, 194
 - condition variables, 195–196
 - defined, 171, 1157
 - EnterCriticalSection function, 187–188
 - kernel critical sections, 172
 - keyed events, 187–189
 - limitations of, 195
 - mutexes in, 194–195
 - semaphores in, 194–195
 - shared mode, 194
 - spinlocks for, 173–174, 177
 - SRW locks as replacement for, 196
 - user mode, staying in, 194
 - critical object stop codes, 1123
 - critical sections. *See* critical code sections
 - critical worker threads, 199
 - CryptoAPI, 992–998
 - cryptographic services, 930
 - Csr functions, 58, 361
 - Csrss.exe. *See also* Windows subsystem
 - boot process initialization of, 1095–1096
 - critical nature of, 1096
 - DLLs loaded by, 54
 - integrity levels of threads created by, 494
 - process creation operations by, 361–362
 - process data structures of, 335–336
 - purpose of, 51–52
 - session manager launching of, 78
 - shutdowns, 1115–1118
 - CTCP TCP (Compound), 1044
 - Ctrl+Alt+Delete logon attention
 - beginning user logon, 516–517
 - desktop chosen by, 515
 - SAS implementation with, 516
 - trusted path functionality of, 453
 - CurrentControlSet subkey, 308–309
- ## D
- DACLs (discretionary access control lists)
 - ACE flags, 487
 - ACEs of, 486–488
 - assignment to new objects, 490–491
 - Create APIs with, 159
 - discretionary access checks using, 495
 - inheritance mechanics, 490–492
 - security descriptors, in, 485–487
 - tokens, of, 476
 - daemon processes. *See* services
 - Data Collector Set Kernel Trace logs, 208–210
 - data execution prevention. *See* DEP (data execution prevention)
 - data integrity, IPsec for, 1050
 - data origin authentication, 1050
 - data structures, process, 335–342
 - datagrams, 1008, 1013
 - data-link layer, OSI, 1003
 - DbgUi functions
 - DbgPrintEx function, 48–49
 - native application support, 217–218
 - Ntdll support for, 58
 - responsibilities of, 216
 - DCE (distributed computing environment), 1014
 - DCOM (Distributed Component Object Model), 1015, 1031–1032
 - DDE (Dynamic Data Exchange)
 - impersonation, 481

- deadlocks, 194, 1147–1150, 1157
- debug objects, 216
- debug ports, 216
- debug programs privilege, 346, 506, 509–510
- debugger commands, kernel. *See* kernel debugger commands
- debugging
 - BCD options for, 1079–1080
 - booting into debugging mode, 1148–1149
 - checked build version use, 47–49
 - commands, kernel. *See* kernel debugger commands
 - copy-on-write with, 719
 - DbgPrintEx function, 48–49
 - DbgUi functions, 58
 - Debugging Tools. *See* Debugging Tools for Windows
 - Driver Verifier, 604–606
 - HD content playback prohibition, 348
 - heap manager features for, 734–735
 - kernel data structures. *See* kernel debugging
 - process creation checks, 363
 - tools for Windows. *See* Debugging Tools for Windows
 - user mode. *See* user-mode debugging
 - user-mode debugging framework of executive, 60
- Debugging Tools for Windows
 - attaching to user-mode processes, 27
 - capabilities for kernel debugging, 28
 - command-line version, 28
 - crash dump files, 28
 - debugger extension commands, 29
 - dt command, 29–30
 - help file for, 29–30
 - invasive attachment option, 27
 - LiveKd tool with, 30–31
 - local kernel debugging, 28
 - noninvasive attachment option, 27
 - pageheap, 735
 - remote connection kernel debugging, 28
 - source for latest versions, 27
 - stop codes, viewing, 1121. *See also* stop codes
 - substructures, viewing, 30
- decrementing, interlocked, 172–173
- decryption, EFS, 998–999
- default resource manager, TxR, 969–971
- deferred deletion of objects, 160
- deferred ready lists, 435
- Deferred Ready thread execution state, 400, 1157
- defragmentation
 - API for, 931
 - fragmentation defined, 931
 - paging files, special tool for, 781
 - SuperFetch service use of, 826–827
 - tool for, 931–932
- delayed worker threads, 199
- delegation, privilege for, 506
- deletion of objects, 148, 160
- demand paging, 823, 1157
- demand zero, 706–707
- Deny logon rights, table of, 503
- DEP (data execution prevention)
 - ATL framework compatibility, 716
 - BCD options for, 714–716
 - bugcheck codes from, 713–714
 - changing computer settings for, 714
 - defined, 713–714
 - PAE mode requirement, 714
 - pointer encoding, 717–718
 - registry values for exclusions, 714
 - self-disabling applications, 716
 - SetProcessDEPPolicy function, 716
 - software DEP, 717–718
 - stack cookies, 717–718
 - viewing, 715–717
- dependencies, loading, 222–227
- Dependency Walker tool, 52, 66
- dereference segment threads, 701
- desired access rights, 157, 1158
- desktops
 - application, created by Winlogon, 515
 - association of windows with by CreateProcess, 351
 - crashes or hangs, troubleshooting, 1112–1114
 - Desktop object type, 137, 148–149
 - watermarked desktop BCD option, 1080
 - Winlogon desktop, 515
- device driver functions, 58, 61, 547–550
- device drivers
 - add-device routines, 548
 - APCs with, 113
 - architecture, place in, 37, 68–69
 - autostarting by SCM, 303–307
 - blue screens, data displayed in, 1120–1121
 - boot phase designation, 624
 - break on symbol load boot option, 1089
 - bugcheck callbacks, 1120
 - bus. *See* bus drivers
 - cancel I/O routines, 549, 587–592
 - catalog files, 635–636
 - checked build version use, 47–49
 - class drivers, 543
 - completion ports for. *See* I/O completion ports
 - crash dumps, secondary, 1127
 - crashes following new installations of, 1124–1125
 - CRITICAL_STRUCTURE_CORRUPTION crashes, workarounds, 245–246
 - defined, 68, 538, 1158
 - detecting illegitimate modifications, 247
 - development framework for. *See* WDF (Windows Driver Foundation)
 - device identifiers, 630
 - device objects, 550–555
 - DeviceIoControlFile API, 214–215
 - dispatch routines, 548, 568
 - DPC routines, 549
 - driver objects, 550–555
 - driver signing policies, 634–636
 - Driver Verifier tool. *See* Driver Verifier
 - enumeration, PnP, 624–628
 - error logging routines, 550
 - fast dispatch routines, 549
 - file objects, 555–562
 - file system. *See* file system drivers (FSDs)
 - filter. *See* filter drivers
 - forcing IRQL checking with Driver Verifier, 802
 - FSDs. *See* file system drivers (FSDs)
 - function drivers. *See* function drivers
 - functions, 58, 61, 547–550
 - Group values, 623–624
 - HAL, relationship to, 68
 - hardware device drivers, 68
 - I/O completion routines, 549
 - I/O requests to layered drivers, 578–587
 - I/O requests to single-layered drivers, 572–578
 - INF files, 538–539, 632–634
 - initialization routines, 548
 - installation of, PnP, 632–636
 - IRP processing, 540
 - ISRs. *See* ISRs (interrupt service routines)
 - kernel access with, 69
 - kernel address space access dangers, 16
 - kernel-mode operation of, 68

- device drivers, *continued*
 - Kernel Patch Protection (KPP) against, 244–246
 - kernel streaming filter drivers, 68
 - kernel-mode, categories of, 542
 - KMDF. *See* KMDF (Kernel-Mode Driver Framework)
 - layering of, 543–544
 - listing loaded, 71
 - load and unload privilege, 507, 510
 - load order of, 623–624, 626–627, 631–632
 - low resources simulation by Driver Verifier, 802
 - miniport drivers, 543
 - names of devices, displaying, 552–553, 1125
 - network redirectors and servers, 68
 - Non–Plug and Play. *See* Non–Plug and Play drivers
 - opening devices, 555–562
 - PAE support for, 770, 819–820
 - paged pools for, 721
 - Plug and Play. *See* Plug and Play drivers
 - pools used by, monitoring, 725–726
 - port drivers, 543
 - portability of, 68
 - printer drivers, 542
 - protected driver lists, 636
 - protocol drivers, 68
 - reason callbacks for crash dumps, 1120
 - registry key records of, 628
 - safe mode loading of, 1102–1103
 - services registry parameters, table of, 283–287
 - spinlocks with, 174–175
 - stacks, viewing, 579
 - start I/O routines, 548
 - Start values, 623–624
 - startup errors, 307–308
 - structures of, 547–550
 - system thread creation, 76
 - system threads, mapping to, 76–77
 - system worker threads for, 198–199
 - system-shutdown notification routines, 550
 - Tag values, 624
 - test-signing mode, 1080
 - types of, 68, 541–542
 - UMDF. *See* UMDF (User-Mode Driver Framework)
 - unkillable processes, debugging, 590–592
 - unload routines, 550
 - VDDs (virtual device drivers), 542
 - viewing lists of, 546–547
 - WDM. *See* WDM (Windows Driver Model) drivers
 - Windows Driver Foundation. *See* WDF (Windows Driver Foundation)
 - Windows Driver Model. *See* WDM (Windows Driver Model) drivers
 - Wow64 I/O request issues, 214–215
- device enumeration, 624–628
- device IDs, 630, 1158
- Device Manager
 - devnodes, viewing, 631
 - memory, viewing reserved, 821
 - viewing devices, 626
- device objects
 - !devobj command, 671
 - creation by drivers, 550–551
 - defined, 550, 1158
 - enumeration of, 551
 - listing addresses of, 671
 - names of, viewing, 552
 - partition objects, 651–652
 - symbolic links for, 551
 - VPB data structures of, 670
- device trees
 - defined, 1158
 - devnodes of, 1158
 - PnP manager creation of, 625–628
- devices
 - device instance IDs. *See* DIDs (device instance IDs)
 - drivers for. *See* device drivers
 - registry key records of, 628
- device-specific modules (DSMs), 648–649
- devnodes
 - defined, 1158
 - device identifiers, 630
 - device objects of, 628–629
 - DIDs (device instance IDs), 630
 - driver loading, 630–632
 - enumeration, use in, 625–627
 - FDOs (functional device objects), 629, 1161
 - FiDOs (filter device objects), 628–629, 1160–1161
 - IRP flow, 627, 629
 - PDOs (physical device objects), 628
 - unloaded drivers during boots, 632
 - viewing with !devnode command, 627–628
- viewing with Device Manager, 631
- DFS (Distributed File System) service, 1069–1071
- Diagnostic Policy Service (DPS), 330–331
- diagnostics, WDI for, 329–333
- Differentiated Services Code Point (DSCP), 1062
- digital rights management, 346–348
- DIDs (device instance IDs), 630, 1158
- direct I/O, 570–571
- directories, file
 - Bypass Traverse Checking privilege, 505, 509
 - index buffers, 950
 - nonresident attributes of, 950
 - restore files and directories privilege, 507, 510
 - synchronization services privilege, 508
 - virtualization, UAC, 523–526
- directories, object, 163
- Directory object type, 137
- Directory Services Restore mode, 1102
- DirectX Win32k.sys wrappers, 54
- dirty pages
 - cache manager flushes to disk, 541–542
 - mapped page writer, 700
 - modified page writer, 700
 - threshold, 885–886, 1158
- DisconnectEx function, 1009
- disconnecting an interrupt object, 105–106
- discretionary access checks, 492, 494–497
- discretionary access control
 - defined, 22, 1158
 - object protection, 458
 - TCSEC rating criteria for, 452
- Disk (Disk.sys) driver, 647, 649–651
- Disk Defragmenter, 931–932
- disk groups, 1158
- Disk Management MMC snap-in, 661–663, 665
- disk storage drivers
 - class drivers, 647
 - class/port/miniport architecture of, 647
 - C-LOOK algorithm, 647
 - Disk driver, 647
 - disk object representation, 650–651
 - DSMs (device-specific modules), 648–649
 - iSCSI drivers, 648
 - miniport drivers, 543, 647

- MPIO (Multipath I/O) drivers, 648–649
- port drivers, 647
- request mechanics, 647
- Scsiport.sys, 647
- startup by Windows I/O manager, 647
- symbolic links for disk objects, 650–651
- Diskmon tool, 649
- disks
 - basic. *See* basic disks
 - CD. *See* CD-ROM drives
 - defined, 645
 - defragmenting. *See* defragmentation
 - diagnostics for, 332
 - drivers for. *See* disk storage drivers
 - dynamic. *See* dynamic disks
 - IDs of, 652
 - integrity checking with Driver Verifier, 606
 - letters for. *See* drive letters
 - monitoring disk activity, 649
 - names, backward compatibility for, 650–651
 - objects representing, 650–651
 - partitions of. *See* partitions, disk
 - paths to, 648–649
 - ReadyDrive for H-HDD, 845–846
 - sectors of. *See* sectors
 - seek times, improvement with prefetching, 823
 - setup for boot process, 1073–1074
 - virtual memory saved to. *See* paging virtual memory to disks
 - volumes. *See* volumes
- dispatch code, 101, 1158
- dispatch routines, device driver, 548
- dispatcher databases
 - defined, 1159
 - mechanics of, 404–405
 - multiprocessor thread scheduling considerations, 434–435
- dispatcher headers
 - debug active flag, 183
 - defined, 182–183, 1159
 - dt command, viewing with, 184
 - flags, table of, 185
 - KTHREAD field for, 372
 - Type field, 185
- dispatcher objects
 - defined, 62, 178–179, 1159
 - dispatcher headers, 182–185
 - fast mutexes built on, 189
 - key traits of, 178
 - manual reset events, 182
 - nonsignaled state, 179
 - signaled state, 179–180
 - wait blocks, 183–186
 - WaitForMultipleObjects function, 178–179
 - WaitForSingleObject function, 178–179
- dispatchers
 - defined, 1158–1159
 - headers, 1159
 - KMDF queue dispatch methods, 614
 - ready queues, 1159
- dispatching
 - dispatcher locks, 594
 - exception. *See* exception
 - dispatching
 - fast dispatch routines, 549
 - interrupt. *See* interrupt
 - dispatching
 - objects for. *See* dispatcher objects
 - routines implementing. *See* dispatchers
 - threads, 392
 - trap. *See* trap dispatching
- display device drivers, 1122, 1159. *See also* video
- Distributed File System (DFS) service, 1069–1071
- distributed link-tracking service, 929–930
- distributed transaction coordinator. *See* DTC (distributed transaction coordinator)
- DLLs (dynamic-link libraries)
 - core subsystem, file names of, 37
 - defined, 4, 1159
 - HAL, 65
 - hotpatches to, 242–244
 - isolating application distributions of, 364
 - known, opening during boots, 1095
 - loading by image loader, 222–227
 - mapped files, viewing as, 711
 - memory-mapping of, updating issues from, 1096
 - randomization of image file addresses, 759–760
 - subsystem DLLs, 36–37, 1175
 - translating API functions to service calls, 54
 - user address space for, 757
- DMA (direct memory access)
 - caching with, 872
 - defined, 1158
 - Driver Verifier, checking with, 606
 - I/O buffering, 570–571
 - pool corruption from, 1140
- DMDiskManager, 661
- DNS (Domain Name System)
 - client service account, 290–291
 - implementation of, 1039
 - NetBIOS names, 1027
 - PNRP advantages over, 1040
 - Winsock service provider for, 1011
- domain controllers, 454–455
- domains, 454–455, 473–474
- DOS machine (VDM) processes, 54
- DosDevices namespace, 167–168
- doubly linked list ExInterlocked functions, 176–177
- DPC stacks, 787
- DPC/dispatch level IRQLs, 97, 100–101, 106–107, 171–172
- DPCs (deferred procedure calls)
 - boot process, stack allocation during, 1090
 - clock IRQL with, 109
 - CPU time of, 75
 - defined, 1157
 - device interrupts, mechanics of, 573–574
 - DPC/dispatch IRQL (interrupt request level), 97, 106–107
 - idle thread role for, 419
 - interrupt dispatching, 106–111
 - interrupt generation rules, table of, 108
 - kernel use of, 109
 - monitoring with Process Explorer, 110–111
 - multiple processor issues, 108
 - object representation of, 107, 1157
 - purpose of, 107
 - queues, 107
 - restrictions requiring scheduler thread changes, 100–101
 - routines for device drivers, 549
 - spinlocks with. *See* spinlocks
 - threaded, 110
 - unresponsiveness from, 110
- DPS (Diagnostic Policy Service), 330–331
- DPWS (Device Profile for Web Services), 1033
- drive letters
 - assigned by mounting. *See* mounting devices
 - checking on boot, 673
 - mapping by network providers, 1036
 - registry storage of, 667–668
- driver objects, 550–555, 1159
- Driver Verifier
 - active items in free memory checks, 802–803
 - boot initializations by, 1089
 - deadlock detection, 1147, 1157
 - defined, 1159

Driver Verifier, *continued*
 enabling system code write protection, 1144
 forcing IRQL checking, 802
 I/O verification, 604–606
 locked memory page tracking, 801
 low resources simulation by, 802
 MDL checks, 803
 mechanics of verification, 799
 memory management options, 799–803
 overview of troubleshooting
 crashes with, 1139
 Pool Tracking, 801
 purpose of, 799
 registry settings for, 799
 special pool option, 1141–1142
 Special Pool verification, 799–801
 drivers. *See also* device drivers
 altitude filtering, 170
 boot start driver issues, 1106
 callbacks, 170
 code corruption from, 1143–1144
 detecting illegitimate modifications, 247
 driver host processes, 616
 Driver Verifier tool. *See* Driver Verifier
 file system. *See* file system drivers (FSDs)
 function. *See* function drivers
 image ranges of, 751–752
 installation of, PnP, 632–636
 IRP processing, 540
 lightweight filter drivers (LWDs), 1057
 NDIS, 1053–1054
 pool corruption caused by, 1140–1142
 protected driver lists, 636
 drives, disk. *See* disks
 drives, logical, 653–654
 DSCP (Differentiated Services Code Point), 1062
 DSMs (device-specific modules), 648–649, 1158
 dt nt!_kinterrupt command, 102–106
 dt nt!_OBJECT_HEADER command, 144
 dt nt!_OBJECT_INITIALIZER command, 145
 dt nt!_OBJECT_TYPE command, 145
 DTC (distributed transaction coordinator), 240, 969–970
 dump analysis. *See* crash dump analysis

Dumpbin utility, 738–739
 DVDs, UDF (Universal Disk Format), 891
 dynamic disks
 advantages over basic disks, 656
 defined, 652–653, 1159
 LDM partitioning for, 656–660
 MBR or GPT partitioning for, 660
 volume manager for, 661
 dynamic-link libraries. *See* DLLs (dynamic-link libraries)
 dynamic processor additions
 HAL support for, 441
 overview of, 441
 thread synchronization for, 441–442
 dynamic volumes, drive letters for, 667–668
 dynamic worker threads, 199–200

E

ECN (Explicit Congestion Notification), 1044
 editions of Microsoft Windows. *See* versions of Microsoft Windows
 EFI (Extensible Firmware Interface)
 Boot Manager, 1087
 boot process mechanics, 1086–1087
 GPT structure of, 654
 hardware detection, 1087
 System partition for, 1087
 x64 vs. x86 boots, 1073
 EFS (Encrypting File System)
 algorithm choices with, 990–991
 architecture of, 991
 backups of encrypted files, 999
 crashes during encryption, 997–998
 CryptoAPI with, 992–998
 DDF (Data Decryption Field), 994–1000
 decrypted FEK caching, 998
 decryption process, 998–999
 disadvantages for system file protection, 677
 DRF (Data Recovery Field), 994–1000
 Encrypt Contents command of Properties dialog box, 990
 FEKs (file encryption keys), 990–991
 file data encryption mechanics, 995–996
 first time file encryption, 993
 key pair assignment, 990
 key ring construction, 994–995
 KSecDD functions for, 455
 Lsasrv with, 992–999
 Lsass management of, 992
 NTFS file attribute supporting, 944
 NTFS use of, 930
 overview of, 990
 private keys, 991
 public keys, 991
 read/write process by applications, 999
 Recovery Agents, 994–1000
 steps for encryption, outline of, 997–998
 symmetric file data encryption, 991
 user master keys, 991
 viewing with EFSDump, 1000
 elevations, UAC, 528–533
 EM64T, 38
 Embedded Standard, Windows, 104–106
 EMS (Emergency Management Services), 1079, 1081
 encryption
 EFS. *See* EFS (Encrypting File System)
 hashes, 1161
 NTFS support for, 930
 RPC, types available for, 1017
 symmetric algorithms for, 1176
 Enlistment objects, KTM, 241
 enumeration of devices, 624–628
 enumeration-based driver loading, 623
 environment subsystems
 defined, 1159
 DLLs loaded by Csrss.exe, 54
 executive objects, relationship to, 136
 role of, 51
 server processes, 36
 environment variables, setting during boots, 1095
 EPROCESS blocks
 !process command, displaying with, 345
 DeviceMap field, 164–167
 protected process access bit, 347–348
 setup by CreateProcess function, 354–355
 structure of, 335–342
 ERESOURCE. *See* executive resources
 errata manager, 60, 1092–1093
 error logging routines, device driver, 550
 errors
 access-denied errors, 909
 blue screens, causing. *See* stop codes

- difficulties for recovery, 240
- KTM for recovery. *See* KTM (Kernel Transaction Manager)
- memory pages not found. *See* page faults
- startup errors, 307–308
- troubleshooting. *See* crash dump analysis; debugging
- Windows hardware error architecture, 61
- ETHREAD blocks
 - creation by PspAllocateThread, 360
 - defined, 370
 - displaying, 374
 - fields of, 370–372
- ETW (Event Tracing for Windows)
 - application types with, 207
 - defined, 61
 - enabling of, 207
 - Etw functions, 58
 - kernel logger, enabling, 208–210
 - kernel trace classes for, 207–208
- Event Log
 - CIM basis for, 321–322
 - permissions for, 511
 - provider for, 320
- Event Logger, 511
- Event object type, 137
- Event Tracing for Windows. *See* ETW (Event Tracing for Windows)
- events
 - defined, 1159
 - gate primitives, 189
 - keyed events, 186–187
 - manual reset events, 182
 - priority level boosts after waits for, 421–422
 - synchronization based, 179
- Everyone group, 462
- Evtlo callback routines, 607
- exception dispatching
 - 64-bit application, 116
 - architecture-independent exceptions, 114
 - debug event for, 217
 - debug registry key values, 120
 - debuggers with, 116–117
 - defined, 1159
 - exception handling role of, 114
 - frame-based exception handlers, 115
 - interrupt numbers, 114–115
 - kernel traps, 115
 - mechanism of, 116–117
 - POSIX, 117
 - start-of-thread function, 117–118
 - structured exception handling, 114
 - unhandled exceptions, 117–120
 - vectored exception handling, 116
 - WER. *See* WER (Windows Error Reporting)
 - Wow64, 212
- exception handling
 - basic concepts of, 114
 - mechanism of, 116–117
 - predefined interrupt numbers for, 114
 - RtlUserThreadStart setup for, 364
 - safe structured exception handling, 717–718
- exceptions
 - defined, 86, 1159
 - dispatching. *See* exception dispatching
 - handling. *See* exception handling
 - kernel-mode, 1119
 - stop codes from, 1122
 - structured exception handling, 114
 - table of, 114–115
 - unhandled exceptions, 117–120
- Exchange Server shadow copies, 688–689
- executable files
 - accessed as data, then as executable, 795–796
 - load offset for address ranges, 759
 - processes from. *See* processes section objects for, 792
- executive mutexes. *See* fast mutexes
- executive objects
 - creation of, 136
 - defined, 135, 1160
 - environment subsystems, relationship to, 136
 - table of, 136–137
 - Windows subsystem use of, 136
- executive process object creation, 354–359
- executive resources
 - boosts during executive resource waits, 422–423
 - defined, 190, 1160
 - key traits of, 178
 - listing, 191
 - mechanism of, 191
 - synchronization, 179
- executive system services, 1176–1177. *See also* native system services
- executive, Windows
 - ALPC, 60
 - APCs with, 112
 - boot process initialization of subsystems, 1088–1093
- cache manager, 59
- common run-time library
 - functions of, 60
- configuration manager, 59
- defined, 37, 1160
- device driver functions, 58
- Driver Verifier, 61
- errata manager, 60
- Event Tracing for Windows, 61. *See also* ETW (Event Tracing for Windows)
- fast mutex acquisition functions, 189
- function prefixes, table of, 72–73
- hypervisor library, 60
- I/O manager component of. *See* I/O manager
- Interlocked functions, 176–177
- internal memory functions of, 58
- lop functions of, 58
- kernel debugger library, 60
- kernel of. *See* kernel, Windows
- kernel transaction manager, 60
- logical prefetcher, 60. *See also* logical prefetching
- major components of, 59–60
- memory manager as part of, 59
- Ntoskrnl.exe, place in, 58
- object manager. *See* object manager
- PnP manager, 59. *See also* Plug and Play manager
- power manager component. *See* power manager
- process and thread manager, 59
- SRM (security reference monitor), 59
- subsystem DLL function calls to, 53
- SuperFetch, 60. *See also* SuperFetch service
- support routines, 60
- system services functions, 58
- user-mode debugging framework, 60
- WDK functions, 58
- Windows diagnostic infrastructure, 61
- Windows Driver Model Windows Management instrumentation routines, 59
- Windows hardware error architecture, 61
- exFAT, 894
- ExInterlocked functions, 176–177
- exit from system service operations, 364
- Explicit Congestion Notification (ECN), 1044

explicit driver loading, 623
 Explorer.exe, Userinit process for, 80
 extended partitions, 653–654, 1160
 Extensible Firmware Interface (EFI),
 654

F

facilities, displaying, 47
 fail fast policy of Windows, 1119
 failed control sets, 1113–1114
 fast dispatch routines, 549
 fast I/O
 defined, 1160
 determination when used, 873
 displaying file system driver
 object table for, 564–565
 performance counters, table of,
 874–875
 purpose of, 873
 steps for I/O with, 873–874
 fast LPC, 55
 fast mutexes, 178, 189–190
 fast user switching, 20, 840
 FAT file systems
 cluster sizes, table of, 892
 exFAT, 894
 FAT32, 894
 file names, 945
 format organization, 892
 mechanics of, 891–894
 recovery issues, 985–988
 fault tolerance, 923. *See also* RAID
 level 5
 faults, page. *See* page faults
 FCBs (file control blocks), 936
 FCL. *See* Framework Class Library
 (FCL)
 FDOs (functional device objects),
 1161
 FEKs (file encryption keys), 990–991
 fibers, 12
 FiDOs (filter device objects),
 1160–1161
 file attributes, NTFS, 940–944,
 948–951
 file control blocks (FCBs), 936
 file mapping objects, 12, 136, 1160.
 See also section objects
 file names, 255, 945–948
 file objects
 !fileobj command, 560
 attributes, table of, 555–556
 cached file pointers, 862
 defined, 555
 extension fields, table of, 556
 File object type, 136
 handles to devices, viewing,
 559–560
 NTFS interactions with, 935–936

opening of files, 557
 security of, 558
 shared underlying resources,
 mechanics for, 560–561
 symbolic links for, 556, 561–562
 viewing data structures of, 557
 viewing name mappings, 561–562
 file records, NTFS, 938
 file references, 1160
 file system drivers (FSDs)
 boot sectors, use of, 896
 cache manager lazy writer, 906
 cache responsibilities, 868, 897
 caching file metadata with
 mapping and pinning, 870–871
 caching operations, 904–905
 defined, 68, 542, 1160
 dismount operations, 897
 explicit file I/O operations,
 902–906
 file change notification support,
 909
 filter drivers. *See* file system filter
 drivers
 I/O manager interactions, 896
 invocation path overview,
 901–902
 IRP handling, 903–906
 kernel-mode drivers, differences
 with standard, 895–896
 LANMan, 897–899
 layering of, 543–544
 local, 896–897
 mailslots with, 1025
 memory manager modified and
 mapped page writer, 906
 memory manager page fault
 handler, 906–907
 named pipes with, 1025
 neither I/O for, 571
 NTFS driver, 934–936
 operations of, 896–897
 oplock protocol, 898–899
 Raw, 896
 remote, 897–899
 requests to layered drivers,
 578–587
 shrinking engine with, 932–933
 volume device objects, 896
 volume I/O operations, 674–675
 volume mounting, 670–674
 volume recognition, 672–673, 896
 VPBs with, 896
 WDK required to build, 895–896
 WebDAV, 897
 file system filter drivers
 defined, 68, 907, 1160
 Filesystem Filter Manager, 907
 Process Monitor as example of,
 907–908

virus scanner use of, 907
 file system runtime library, 61
 file systems
 access-denied errors, 909
 buffer management by cache
 manager, 871
 buffer overflow errors, 910
 caching for. *See* cache manager
 CDFs, 890–891
 clusters. *See* clusters, file
 common logging. *See* CLFS
 (Common Log File System)
 common logging file system
 driver (Clfs.sys), 67
 defragmentation, 931–932
 drivers. *See* file system drivers
 (FSDs)
 exFAT, 894
 FAT. *See* FAT file systems
 formats, 889–890, 1160
 logical sequence numbers (LSNs),
 853–854
 metadata defined, 890
 metadata update steps ensuring
 recovery, 853
 NTFS. *See* NTFS (NT File System)
 overview of, 889–890
 recovery from corruption of,
 986–988
 sectors of. *See* sectors
 stop code for fatal errors, 1123
 troubleshooting, 908–910
 UDF (Universal Disk Format), 891
 viewing registered, 900–901
 Wow64 file system redirection,
 212–213
 FileInfo driver, 839
 files
 access-denied errors, 909
 attributes, NTFS. *See* file
 attributes, NTFS
 CreateFile function, 557
 creation of objects for, 136
 file change notifications, 909
 handles to open files, searching
 for, 155–156
 NTFS. *See* NTFS (NT File System)
 object IDs for, 929–930
 object security for, 460
 objects, default, 146
 reference numbers, NTFS, 942
 restore files and directories
 privilege, 507, 510
 symbolic links, 167
 virtualization, UAC, 523–526
 writing to, access checks on, 460
 filter drivers
 defined, 543, 1161
 file system. *See* file system filter
 drivers

FVE (full volume encryption), 686
 lightweight filter drivers (LWDs), 1057
 loading order, 631–632
 Plug and Play support, 621
 purpose of, 69–70
 UMDF reflectors, 617
 filter engine, WFP, 1047
 filtered admin tokens, 483–484
 fingerprint readers, 79
 firmware environment modify privilege, 508
 flags object header field, 139
 flags, global, 200–202
 flash memory, 844–846
 floating point state, 101–102
 foreground processes, 423–425, 1161
 foreign volumes, 685
 frame-based exception handlers, 115, 1161
 frames, stack, 1175
 Framework Class Library (FCL), 3
 free builds of Windows, 1161
 Free page state, 804
 freeing virtual memory pages, 707
 FRS (File Replication Service), Windows, 956
 FSDs. *See* file system drivers (FSDs)
 Fsutil.exe
 default resource manager queries, 971
 NTFS volume information, 941
 self-healing checks, 989
 transaction queries, 968
 FTP, WinInet API for, 1019
 function drivers
 defined, 69, 543, 1161
 FDOs (functional device objects), 1161
 loading order, 631–632
 functional device objects. *See* FDOs (functional device objects)
 functions
 cache copying, 869
 fiber functions, 12
 kernel support. *See* kernel support functions
 Ntdll.dll system support library, 57–58
 prefixes, table of, 72–73
 process-related, table of, 344–345
 system services, 58
 thread creation and manipulation, table of, 380
 time in, determining, 97–100
 Windows API, 4

G

games, MMCSS with, 430–434
 gates, 178, 189–190, 192–193
 Gate Waiting thread execution state, 401
 GDI (Graphics Device Interface)
 batched operations, 55
 direct access for, 55
 functions for, 54–55
 GDI/User objects, 135
 service descriptor tables for, 130–133
 Win32k.sys, contained in, 54
 windowing and graphics system, 37
 GDTs (global descriptor tables), 64
 Get process functions, table of, 344–345
 GetThreadContext function, 12
 Gflags.exe tool
 enabling special pool with, 1141
 image loader, viewing debug output from, 221
 viewing and editing flags, 200–202
 global clouds, P2P, 1040
 global flags, 200–202
 global replacement policies, 828
 GPT partitioning, 654, 660
 granted access rights. *See also* access control
 defined, 1161
 granted-access masks, 495
 mechanics of, 157
 granularity of memory allocations, 708
 graphics device drivers, 54
 Graphics Device Interface. *See* GDI (Graphics Device Interface)
 guarded mutexes, 178, 189–190
 GUI permissions editors, 498–499
 GUID Partition Table. *See* GPT partitioning

H

HAL (hardware abstraction layer)
 architecture, role in, 65–67
 BCD options for, 1081
 bus driver, acting as, 624–625
 checked build version, 49
 defined, 37, 1161
 determining version running, 66
 device drivers, interaction with, 68
 DLLs for, 37, 65
 driver verifier option, 604
 dynamic processor support, 441
 Hal.dll, 1075
 I/O system component, as a, 539
 IRQ mapping IRQLs, 96
 kernel dependencies, viewing, 66
 lazy IRQL, 93
 NSIS miniport driver functions for, 1004
 portability role of, 38–39
 purpose of, 65
 system space for, 737, 752
 undocumented interfaces, viewing, 72–73
 handles
 !devhandles command, 155–156
 !handle command, 154
 !htrace command, 156
 access masks of, 153
 accessing objects without, 150
 advantages of, 150
 counters for, 159–161
 creation of, 149
 debugging mechanisms, 156
 defined, 149, 1161
 duplication of, 149
 flags of, 153
 granted access rights, 157
 Handle count object header field, 139
 handle information object subheader, 139–140
 Handle tool, 151
 Handle Tracing Database, 156
 Handle Viewer, 133
 inheritance of, 149, 153
 kernel handle tables, 154–156
 leaks, 156
 names of objects with, 163
 object, 1169
 object manager rights to, 150
 Object Reference Tracing, 156
 object retention, role in, 159–161
 okay to close object method, 148
 open file handles, searching for, 155–156
 per process maximum, 151–153
 processes, lists for, 5, 13
 protect from close flag, 153
 session namespaces, viewing, 168–169
 structure of, 153
 subhandle tables, 152
 tables for, 150–152, 1161
 Testlimit tool for, 152–153
 user mode process requirement for, 149
 viewing open, 150–151
 hard disks. *See* disks
 hard links, 923–924
 hardware
 crashes following new installations of, 1124–1125
 disk drives. *See* disks

- hardware, *continued*
- hardware IDs, 634
- HKEY_LOCAL_MACHINE\
 - HARDWARE subkey, 258
- interrupt processing. *See* interrupt dispatching
- kernel support role, 64–65
- memory. *See* physical memory processors. *See* CPUs
- stop codes for, 1123
- Windows hardware error architecture, 61
- hardware abstraction layer. *See* HAL (hardware abstraction layer)
- hardware device drivers, 68, 1161.
 - See also* device drivers
- Hardware Installation Wizard, 632
- hashes, 1161
- headers, object. *See* object headers
- heap manager
 - affinity manager of LFH, 733
 - allocation management, 730–731
 - architecture of, 730
 - C runtime use of, 730
 - core functionality, 731
 - debugging features, 734–735
 - defined, 1161–1162
 - front end layer, 731
 - functions for, 730
 - granularity of allocations by, 729–730
 - LFHs (Low Fragmentation Heaps), 732–733
 - locking heaps, 732
 - pageheap, debugging with, 735
 - purpose of, 729–730
 - security features, 733–734
 - structure of, 731
 - synchronization of heaps, 732
- heaps
 - address ranges, viewing, 758
 - allocation management, 730–731
 - block header dumps, 734
 - buckets, 732–733
 - creation by processes, 730
 - default process heaps, 730
 - defined, 1161
 - destruction by processes, 730
 - functions for, 730
 - HeapWalk block enumeration function, 732
 - IDs, displaying, 759
 - image loader initialization of, 220, 222
 - kernel mode. *See* pools, memory
 - LFHs (Low Fragmentation Heaps), 732–733
 - locking, 732
 - manager for. *See* heap manager
 - memory-mapped, issues with, 730–731
 - multiprocessor issues, 732
 - pageheap, debugging with, 735
 - randomization by ASLR, 760
 - security issues of, 733–734
 - synchronization, 732
 - system. *See* pools, memory
- H-HDDs, 844–845
- hibernation, 840, 1078
- hierarchy prioritization strategy for I/O, 599
- high-IRQL synchronization
 - ExInterlocked functions, 176–177
 - fast mutexes, 189–190
 - Interlocked functions, 172–173
 - kernel critical sections, 172
 - motivation for, 171–172
 - overview of, 171–172
 - queued spinlocks, 175–176
 - spinlocks. *See* spinlocks
- hives, registry
 - !reg hivelist, 274
 - ASCII vs. Unicode storage, 281
 - base blocks, 269
 - bins, 269–272
 - blocks, 269
 - cell indexes, 271
 - cells, 269–272
 - defined, 1162
 - dirty sector arrays, 279
 - handles, viewing, 268
 - key cell type, 270
 - lazy write operations, 279
 - loading with Regedit, 267
 - log hives, 278–279
 - memory mapping, 272–276
 - memory usage, viewing, 274
 - paged pool usage, viewing with !reg dumptool, 273
 - root keys of, 266
 - security-descriptor cell type, 270
 - size limits of, 268
 - structure of, 266, 269–272
 - subkey cell type, 270
 - symbolic links, 268
 - syncs, 279
 - system hive corruption, 1112
 - table of registry and file paths, 267
 - value cell type, 270
 - value-list cell type, 270
 - version numbers, 269
 - volatile, 266
- HKEY_CLASSES_ROOT (HKCR), 255
- HKEY_CURRENT_CONFIGURATION (HKCC), 259
- HKEY_CURRENT_USER (HKCU), 253
- HKEY_LOCAL_MACHINE (HKLM)
 - BCD subkey, 255–257
 - COMPONENTS subkey, 257–258
 - CurrentControlSet subkey, 308–309
 - defined, 23, 255
 - HARDWARE subkey, 258
 - hive file paths for, table of, 267
 - SAM subkey, 258
 - SECURITY subkey, 258
 - Services subkey, 282–287
 - SOFTWARE subkey, 258
 - SYSTEM subkey, 259, 268
- HKEY_PERFORMANCE_DATA (HKPD), 259
- HKEY_USERS (HKU)
 - \DEFAULT, 253
 - hive file paths for, table of, 267
 - purpose of, 253
 - user profiles, 254
- hooks to keystrokes, 516
- host-based virtualization, 228
- hotfixes, pending file moves for, 1096
- hotpatches, 242–244
- HTTP (Hypertext Transport Protocol)
 - 1.0, Wininet API for, 1019
 - cache functions, 1020–1021
 - HTTP Server API, 1019–1021
 - Http.sys, 1019–1020
 - RPC transport interface for, 1016
 - servicing. *See* Web servers
 - WebDAV miniport driver, 897
 - WinHTTP, 1019
- hung systems, troubleshooting, 1147–1150
- HungAppTimeout registry values, 1115–1116
- hypercritical worker threads, 199
- hyperspace, 737, 1162
- hyperthreading
 - defined, 40
 - ideal processor numbers with, 440
 - idling processors, thread preferences while, 443
 - pause assembly instructions, 174
 - viewing information for, 436
- Hyper-V
 - address spaces for, 238–239
 - AMD vs. Intel implementations, 232
 - API library for, 232
 - architecture of, 230–231
 - child partitions, 230, 232–234
 - emulated devices, 235
 - enlightened storage I/O, 236
 - enlightenments, 229, 233–234
 - GPA (guest physical address) space, 238–239
 - Guest Virtual Address Spaces, 232–233
 - hardware device support, 234–237

- hardware virtualization, 228
 - hypervisor architecture, 228
 - hypervisor component, 232
 - hypervisor stack, 230–231
 - I/O architecture, 235–236
 - integration components, 235–236
 - intercept mechanism, 239
 - memory, 238–239
 - partitions overview, 230
 - patch issues, 245
 - purpose of, 229
 - root partitions, 230–232
 - scheduler, 237
 - slowness of installations, 235
 - SPA (system physical address)
 - space, 238–239
 - synthetic devices, 235–237
 - VDevs (virtual devices), 234–235
 - VID (VM infrastructure driver), 232
 - virtual machine (VM) service, 231
 - virtual processors, 237–238
 - virtualization overview, 228
 - VMBus, 236–237
 - VSCs (virtualization service clients), 233, 236–237
 - VSPs (virtualization service providers), 232, 236–237
 - Windows Server Core, 231
 - worker processes, 231–232, 234–235
 - hypervisors. *See also* Hyper-V
 - BCD options for, 1081
 - defined, 1162
 - library, 60
- I**
- I/O completion ports
 - advantages of, 592–593
 - completion status tests, 563
 - concurrency values, 593–595
 - creating, 595
 - defined, 1156
 - IoCompletion objects, 593
 - lock contention issues, 594
 - mechanics of, 593–597
 - notification modes, 597–598
 - queue basis of, 595–597
 - scalability from, 43
 - thread agnostic I/O, 587, 595
 - worker factory use of, 388
 - I/O completion routines, 1162
 - I/O manager
 - advantages of, 540
 - boots, initializations during, 1091–1093
 - cancellations, I/O, 587–592
 - defined, 538
 - device driver dispatch routines, 548
 - device load order role, 626
 - device objects, 550–555
 - driver abstraction, 540
 - driver objects, 550–555
 - driver verifier option, 604
 - drivers, layering of, 543–544
 - error logging routines of device drivers, 550
 - fast I/O with, 565, 873–875
 - file object security, 460
 - file system driver interactions, 896
 - initialization routines, device driver, 548
 - IRPs, 539–540, 570–571
 - MDLs (memory descriptor lists), 570–571
 - neither I/O, 571
 - NTFS file system driver
 - interactions with, 934–935
 - opening files, 557–561
 - packet-driven nature of, 539
 - requests to layered drivers, 578–587
 - requests to single-layered drivers, 572–578
 - shutdowns, 1117–1118
 - start I/O routines, 548
 - Start value for, 623–624
 - UMDF management, 617
 - volume mounting role, 670–671
 - Windows executive, as
 - component of, 59
 - I/O priorities
 - hierarchy prioritization strategy, 599
 - idle prioritization strategy, 599
 - purpose of, 598
 - storage port driver strategy, 600
 - system storage driver strategy, 600–601
 - table of, 598–599
 - viewing, 601–602
 - I/O processing
 - asynchronous I/O, 563
 - bandwidth reservation, 603–604
 - buffered I/O, 570–571
 - caching files, 565–566
 - cancellations, I/O, 587–592
 - completion ports. *See* I/O completion ports
 - direct I/O, 570–571
 - fast I/O, 564–565
 - I/O completion, 575–577
 - in-paging I/O, 778–779
 - interrupts, mechanics of, 573–574
 - IoCompletion objects, 593. *See also* I/O completion ports
 - IRPs in, 566–571
 - locked user buffers, 587
 - mapped file I/O, 565–566
 - neither I/O, 571
 - priorities, 598–603
 - requests to layered drivers, 578–587
 - requests to single-layered drivers, 572–578
 - scatter/gather I/O, 566
 - scheduled file I/O, 603–604
 - status blocks, 575
 - synchronization issues, 577–578
 - synchronous I/O, 563, 572–578
 - thread agnostic I/O, 587
 - thread termination cancellations, 589–592
 - types of I/O, 563–571
 - unkillable processes, debugging, 590–592
 - I/O subsystem API, 1162
 - I/O system. *See also* I/O processing
 - components of, 537–539
 - defined, 1162
 - design goals of, 537–538
 - device drivers. *See* device drivers
 - file abstraction, 540–541
 - file objects, 555–562
 - I/O completion routines, 549
 - INF files, 538–539
 - KMDF. *See* KMDF (Kernel-Mode Driver Framework)
 - manager. *See* I/O manager
 - miniport drivers, 543
 - packet-driven nature of, 539
 - Plug and Play. *See* Plug and Play manager
 - port drivers, 543
 - power management with. *See* power manager
 - typical processing, 540–541
 - UMDF. *See* UMDF (User-Mode Driver Framework)
 - I/O, explicit file, 902–906
 - IA64 CPUs
 - address translation, 772
 - page sizes for, 705
 - versions of Windows for, 44–45
 - virtual address space layouts, 746
 - Wow64 restrictions, 215
 - IDE, Ataport.sys, 647
 - ideal processors, 440–441, 1162
 - IDL (Interface Definition Language), 1016
 - idle prioritization strategy for I/O, 599
 - idle process, 75
 - idle summary masks, 434
 - idle system polling behavior, 909

1200 idle threads

- idle threads, 418–419
- IDTs (interrupt dispatch tables)
 - defined, 1163
 - entry numbers, 89
 - indexes for, 88
 - interrupt objects, 101
 - one per processor rule, 89
 - viewing, 88–89
- IEEE 1394 (FireWire), UMDf with, 616
- IIS (Internet Information Server), 1019
- IKE (Internet Key Exchange), 1051–1052
- image files
 - accessed as data, then as executable, 795–796
 - address ranges for, displaying, 758
 - DLLs. *See* DLLs (dynamic-link libraries)
 - image bias computation, 759
 - large address space awareness, 699, 739, 746
 - loading. *See* image loader
 - randomization of, 759–760
- image global flags, 200–202
- image loader
 - behavior of, 220
 - bound import tables, 226
 - defined, 1162
 - dependencies of DLLs, loading, 222–227
 - flags, loader data table entry, 225
 - forwarder entries, 226
 - import parsing steps, 226–227
 - initialization steps of, 222
 - Ldr functions, 58
 - LDR_DATA_TABLE_ENTRY structures, 223–225
 - linked DLLs, loading, 222, 226–227
 - loaded module database, 223–225
 - loader data table entries, 223–225
 - loader data table entry fields, table of, 223
 - loader snaps, 221
 - mapped files, 710
 - Ntdll.dll, residence in, 220
 - PEB_LDR_DATA, 223–225
 - post-process initialization steps, 227–228
 - purpose of, 220
 - relocation, 226
 - tasks of, 220
 - viewing debug output from, 221
- impersonation
 - advantages of, 480–481
 - client/server model with, 480–483
 - defined, 458, 1162
 - disadvantages of, 481
 - mechanisms available for, 481
 - modifiers for settings, 482
 - named pipes, for, 1022
 - privilege for, 506
 - RPC, for, 1017
 - servers, limitations on use of, 482–483
 - spoofing issues, 482–483
 - SQOS levels, 482
 - system global data location okay for, 522
 - Winsock functions for, 1009
- indexing NTFS files, 923, 960–961
- INF files, 538–539, 632–634, 1064–1065
- init once. *See* run once initialization
- initialization routines, device driver, 548, 607, 1162–1163
- Initialized thread execution state, 401
- in-page I/O operations, 778–779, 1163
- input/output. *See* I/O system
- instack queued spinlocks, 176
- InstallShield, 64-bit Windows issues, 215
- instance IDs, PnP, 630
- instancing namespaces, 1163
- integrity levels
 - access of processes based on, 492
 - AccessCheck tool, displaying with, 158
 - default policies, 492
 - defined, 1163
 - explicit object integrity levels, 471–472
 - implicit object integrity levels, 471
 - launching programs in Low, 480
 - mandatory integrity checks, 492
 - object, viewing, 472–473
 - objects, of, 471–473
 - PMIE example, 466–470
 - process propagation of, 466
 - purpose of, 459
 - SIDs for, 464–465
 - spoofing prevention, 482–483
 - UIPI with, 493–494
 - viewing for processes, 465–466
- integrity of data, IPSec for, 1050
- intelligent read-ahead
 - defined, 1163
 - mechanics of, 875–876
 - preventing, 876
 - private cache maps, 1171
 - viewing with Process Monitor, 878–883
- Interactive Services Detection service, 299–300
- intercepts, hypervisor management of, 239
- Interface Definition Language (IDL), 1016
- Interlocked functions, 172–173
- Internet APIs. *See also* Web access APIs
 - FTP, 1019
 - HTTP APIs. *See* HTTP (Hypertext Transport Protocol)
- Internet Explorer
 - memory address ranges, viewing for, 758
 - Protected Mode integrity levels, 466–470
- Internet Key Exchange (IKE), 1051–1052
- interrupt controllers
 - APICs, 89–91
 - IA64 systems, 90
 - IRQ queries, 88
 - IRQ to IDT mapping, 89
 - MP Specification, 89–90
 - PICs, 89–91
 - purpose of, 88
 - routing algorithms, 89–90
 - slaves, 89–90
 - state storage by KPCR, 62–64
 - x64 systems, 89–90
 - x86 systems, 89–90
- interrupt dispatching
 - APC interrupts, 112–114
 - APICs, 89–91
 - connecting an interrupt object, 105–106
 - disconnecting an interrupt object, 105–106
 - dispatch code, 101
 - dispatchers defined, 1163
 - DPC interrupts, 106–111
 - dt nt!_kinterrupt command, 102–106
 - floating point state, 101–102
 - hardware interrupt overview, 88–89
 - I/O device generated of interrupts, 87
 - interrupt objects, 101–106
 - IRQ queries, 88
 - IRQLs. *See* IRQLs (interrupt request levels)
 - IRQs. *See* IRQs (interrupt requests)
 - ISRs, handing control to, 87
 - KiInterruptDispatch, 101
 - KiInterruptTemplate, 101
 - masked interrupts, 93–94
 - message-based interrupts, 106
 - overview of trap dispatching, 85–87
 - PICs, 89–91
 - Plug and Play manager with, 105

- real-time processing issues, 104–106
 - service routines. *See* ISRs (interrupt service routines)
 - software generated interrupts, 87
 - software interrupt overview, 106
 - tables. *See* IDTs (interrupt dispatch tables)
 - trap handler installation, 87
 - x64 systems, 89–90
 - x86 systems, 89–90
 - interrupt objects, 101–106, 1163
 - interrupt request levels. *See* IRQs (interrupt request levels)
 - interrupt service routines. *See* ISRs (interrupt service routines)
 - interrupts
 - APC interrupts, 112–114
 - controllers. *See* interrupt controllers
 - defined, 86, 1163
 - device-based, mechanics of, 573–574
 - disabling by kernel, 87
 - dispatching. *See* interrupt dispatching
 - dt nt!_kinterrupt command, 102–106
 - idle thread role for, 419
 - IoConnectInterruptEx API, 106
 - KINTERRUPT objects, 102, 174–175
 - masked, 93–94, 172
 - message-based interrupts, 106
 - request levels. *See* IRQs (interrupt request levels)
 - requests. *See* IRQs (interrupt requests)
 - routing algorithms, 89–90
 - service routines. *See* ISRs (interrupt service routines)
 - viewing objects associated with, 102–106
 - intrinsic functions, 172–173
 - IoCompletion objects, 137, 593
 - IoConnectInterruptEx API, 106
 - IoQueueWork functions, 198
 - IP filtering, 1049–1050
 - IPs (interprocessor interrupts), 96, 1163
 - IPSec (Internet Protocol Security), 1050–1052, 1054
 - IPv4, 1044–1045
 - IPv6
 - HTTP Server API support for, 1019
 - Next Generation TCP/IP Stack for, 1044–1045
 - Peer Name Resolution Protocol (PNRP), 1039–1041
 - SMB (Server Message Block) protocol with, 1025
 - Winsoc support for, 1007
 - WSK (Winsoc Kernel) support for, 1004
 - IrDA, Winsoc Helper library for, 1011
 - IRPs (I/O request packets)
 - !lirp debugger command, 570
 - associated, 584–585
 - buffer management, 570–571
 - cancel I/O routines, 549
 - components of, 568
 - crash dump analysis of, 1146, 1150
 - defined, 1162
 - device driver dispatch routines, 548
 - device trees, flow through, 627
 - examining, 581–582
 - file system driver handling of, 903–906
 - function codes, 568
 - headers of, 568
 - I/O completion, 575
 - I/O manager creation of, 539–540
 - IRP stack locations, 579
 - KMDF model with, 612–614
 - lists of, 569
 - mechanics of, 566–567
 - monitoring with Driver Verifier, 606
 - stack locations, 568–570
 - start I/O routines, 548
 - thread termination cancellations, 589–592
 - uncompleted, 581–582
 - IRQs (interrupt request levels)
 - !lirq command, 95
 - APC level, 97
 - clock level, 97, 109
 - correctable machine check levels, 97
 - crashes due to, 101
 - defined, 92, 1163–1164
 - device levels, 97
 - DPC/dispatch level, 97, 100–101, 106–107, 171–172
 - fast mutex raising of, 189
 - forcing IRQs checking with Driver Verifier, 802
 - high level, 96
 - high, crashing systems with, 1135–1137
 - high, synchronization. *See* high-IRQs synchronization
 - interprocessor interrupt level, 96
 - interrupt objects, 101–106
 - IPs, 96
 - kernel-mode thread effects, 94
 - lazy IRQs, 93, 1165
 - levels, list of predefined, 96–97
 - levels, purposes of specific, 96
 - low, synchronization. *See* low-IRQs synchronization
 - mapping IRQs to, 96
 - masked interrupts, 93–94, 172
 - multiprocessor system issues, 434–435
 - numbers for, 92
 - page fault stop codes, 1121
 - passive level, 97
 - PCR, values saved to, 95
 - power fail level, 96
 - processor IRQs settings, 93
 - profile level, 97
 - servicing order, 92–93
 - synchronization dependence on, 171–172
 - system worker threads for lowering, 198
 - thread scheduling, raising for, 405
 - threads, of, 399
 - user-mode code level, 96
 - IRQs (interrupt requests)
 - defined, 1163
 - mapping IRQs to, 96
 - mapping to IDTs, 89
 - queries for, 88
 - real-time processing issues, 104–106
 - verifying with Device Manager, 103–104
 - iSCSI drivers, 648
 - isolation of services, 294–300
 - isolation of transactions by TxR, 966–968
 - ISRs (interrupt service routines)
 - connecting an interrupt object, 105–106
 - control, transfers to, 87
 - defined, 1164
 - device driver use of, 548–549
 - device interrupts, mechanics of, 573–574
 - interrupt objects, 101–106
 - Plug and Play manager with, 105
 - purpose of, 87
 - Itanium processors. *See* IA64 CPUs
- ## J
- jobs
 - !job command, 447
 - binding during process creation, 359
 - default working sets, 447
 - defined, 13
 - job objects defined, 445, 1164

1202 Joliet disk format

jobs, *continued*
 maximum active processes limit, 446
 object type for, 136
 per-process user-mode CPU time limits, 446
 process priority class, 447
 processor affinity setting, 447
 scheduling class, 446
 security limits, 447
 user-interface limits, 447
 user-mode CPU time limits, 446
 viewing names and processes of, 447–449
 virtual memory limits, 447
 Windows API functions for, table of, 446
Joliet disk format, 890–891
journaling, 1164
junctions, 926

K

Kd.exe, 28, 1134
Ke spinlock functions, 174–175
KeAcquireStackQueuedSpinLock, 176
KeBugCheckEx function, 87, 1120–1121
KeInitThread, 360
Kenrate tool, 97–100
Kerberos authentication
 domains, user logons to, 517
 IPSec support for, 1051
 mechanics of logons, 518
 purpose of, 513
kernel debugger commands
 !analyze command, 1149
 !lpic command, 91
 !dbgprint command, 49
 !defwrites command, 886
 !devhandles command, 155–156
 !devnode command, 627
 !devstack command, 579
 !drvobj command, 671
 !lexqueue command, 200
 !gflag command, 202
 !handle command, 133, 154
 !heap, 734
 !htrace, 156
 !idt command, 88–89, 102
 !ioapic command, 91
 !lirp command, 1145–1147
 !lirq command, 95
 !ljob command, 447
 !locks command, 191
 !memusage command, 807
 !miniports, 1055
 !numa command, 437
 !object command, 144, 489, 552–553
 !obtrace, 156
 !pcr command, 62–64, 95
 !pic command, 91
 !popolicy command, 642
 !process command, 144, 345, 447–449, 765, 1144
 !qlocks command, 176
 !ready command, 391
 !reg dumppool, 273
 !reg findkcb, 278
 !reg hivelist, 274
 !reg kcb, 278
 !reg openkeys, 278
 !teb command, 378
 !thread command, 184, 376, 569
 !timer command, 109
 !token command, 478
 !vm command, 703, 1145
 !wdfkd.wdfldr, 608
 !wsle, 830
 defined, 29
 lm kv, 547
kernel debugging
 ! commands. *See* kernel debugger commands
 attaching Debugging Tools to user-mode processes, 27
 booting into debugging mode, 1148–1149
 crash dump files, 28
 crashes, viewing with host debuggers, 1151
 debug events, table of, 216–217
 debugger extension commands, 29
 defined, 26
 dt command, 29–30
 handle tables, viewing, 154
 help file for, 29–30
 KPCR, displaying, 62–64
 LiveKd tool for, 30–31
 local, 28
 remote connection method, 28
 security descriptors, viewing, 488–490
 substructures, viewing, 30
 symbol files, 26–27
 tools for, 26
 Windows executive library, 60
kernel dispatcher objects. *See* dispatcher objects
kernel event tracing, 207–210
kernel handle tables, 1164
kernel logger. *See* KT Kernel Logger
Kernel Memory counters, 702
kernel memory dumps, 1126–1129, 1164

kernel mode
 access dangers, 16
 address space of, 16
 architecture, role in system, 34–35
 cache copying functions, 869
 code signing (KMCS), 17
 components of Windows, 37
 defined, 16, 1164
 device drivers. *See* kernel-mode device drivers
 executive, Windows, 37
 HAL (hardware abstraction layer), 37
 handle tables for, 154–156
 heaps. *See* pools, memory
 KMDF. *See* KMDF (Kernel-Mode Driver Framework)
 monitoring in Task Manager, 18
 Ntdll.dll system support library, calls from, 57–58
 performance counters for, 17–19
 switching with user mode, 17
 system service dispatching, 127–129
 system threads. *See* system threads
 time in, determining, 97–100
 windowing and graphics system, 37
Kernel-Mode Driver Framework (KMDF). *See* KMDF (Kernel Mode Driver Framework)
kernel objects, 135, 1164–1165
Kernel Patch Protection (KPP), 244–246
Kernel Profiler tool, 97–100
kernel profiling, 97
kernel queue worker factory support, 388
Kernel Security Device Driver. *See* KSecDD (Kernel Security Device Driver)
kernel stacks, 786–787, 816
kernel streaming filter drivers, 68, 1165
kernel support functions, 4
kernel transaction manager. *See* KTM (Kernel Transaction Manager)
kernel trap handlers. *See* trap dispatching
kernel variables, 342–343, 379
kernel, Windows
 boot process initialization of subsystems, 1088–1093
 compatibility between Windows versions, 45
 context switching, 64–65
 control objects, 62

- debuggers in. *See* kernel debugging
- defined, 37, 1164
- dispatcher objects. *See* dispatcher objects
- executive objects, relationship to, 61
- handle tables for, 154–156
- hardware support role, 64–65
- kernel-mode components of Windows, 37
- KPCR. *See* KPCR (kernel processor control region)
- KPRCB. *See* KPRCB (kernel processor control block)
- microkernel-based operating systems, 34–35
- mode for processes. *See* kernel mode
- MS-DOS support, 64
- Ntoskrnl.exe binaries links, 67
- objects, 61–62
- processor control region, 62–64
- protected process support, 347
- structure of, 61
- transaction manager. *See* KTM (Kernel Transaction Manager)
- unified for multiple CPUs, 41
- user-written code for, using device drivers, 69
- variables. *See* kernel variables
- Kernel32.dll
 - CreateThread function, 380–381
 - mailslots implementation by, 1024–1026
 - named pipes implementation by, 1024–1026
 - process initializing operations, 360–362
- kernel-mode device drivers
 - bus drivers, 542–543
 - C programming language for, 550
 - categories of, 542
 - class drivers, 543
 - defined, 1165
 - file objects, 555–562
 - file system. *See* file system drivers (FSDs)
 - filter drivers, 543
 - function drivers, 543
 - I/O requests to single-layered drivers, 572–578
 - layering of, 543–544
 - listing, 546
 - !m kv command, listing with, 1144
 - miniport drivers, 543
 - Non-Plug and Play drivers, 542, 550
 - Plug and Play drivers, 542
 - port drivers, 543
 - structures of, 547–550
 - WDM. *See* WDM (Windows Driver Model) drivers
- kernel-mode graphics drivers, 1165
- kernel-mode system threads. *See* system threads
- KeServiceDescriptorTable, 130–133
- KeSynchronizeExecution, 578
- key objects, 137, 276, 1165
- keyed events, 186–189, 1165
- keys, encryption, 991
- keys, registry
 - cell indexes, 271
 - control blocks, 1165
 - defined, 251, 1165
 - HKEY_CLASSES_ROOT, 255
 - HKEY_CURRENT_CONFIGURATION (HKCC), 259
 - HKEY_CURRENT_USER, 253
 - HKEY_LOCAL_MACHINE. *See* HKEY_LOCAL_MACHINE (HKLM)
 - HKEY_PERFORMANCE_DATA (HKPD), 259
 - HKEY_USERS (HKU), 253–254
 - key cell type, 270
 - key control blocks, 276–278, 281
 - key object type, 276–277
 - naming convention, 251
 - new, created by applications, 277
 - object namespace form of key names, 276
 - root keys. *See* root keys, registry subkeys, 251
 - symbolic links, 268
 - value types, 251–252
- KINTERRUPT, 102, 174–175
- KiThreadStartup, 363
- KiWaitTest function, 179
- KMCS (kernel-mode code signing), 17, 246–247
- KMDF (Kernel-Mode Driver Framework)
 - add-device routines, 607
 - attributes, table of, 612
 - callback routines for events, 607–608, 614
 - data model for, 608–612
 - defined, 70
 - dispatch methods for queues, 614
 - displaying drivers, 608
 - events, 607–608
 - Evtlo callback routines, 607
 - handler tasks, 614
 - handlers for IRPs, 612–613
 - hierarchy of, 609–611
 - I/O model, 612–614
 - initialization routines, 607
 - IRPs with, 612–614
 - object contexts, 611–612
 - object management, 608–609
 - object types, table of, 609–610
 - power management states, 614
 - queues for, 613–614
 - structure of drivers for, 607
 - suitability of, 607
 - WDM, relation to, 607, 612
- KOBJECTS enumeration, 185
- KPCR (kernel processor control region), 62–64
- KPP (Kernel Patch Protection), 244–246
- KPRCB (kernel processor control block)
 - cache mapping and pinning fields, 871
 - cache MDL activity fields, 872
 - cache read activity fields, 869–870
 - fast I/O fields, 874–875
 - flush operation fields, 883
 - lazy write fields, 877
 - mechanism of, 62–64
 - thread-scheduling fields, 405
- KPROCESS blocks
 - creation of, 356
 - KeInitializeProcess, 356
 - page directories addresses, 764
 - structure of, 338–340
- KSecDD (Kernel Security Device Driver), 455
- KT Kernel Logger, 207–210
- KTHREAD blocks
 - defined, 372
 - displaying, 375
 - fields, table of, 372–373
 - initialization during process creation, 360
 - system service pointer field, 372
- KTM (Kernel Transaction Manager)
 - APIs for, 260
 - CommitTransaction function, 261
 - CreateTransaction function, 260
 - deferred deletion of objects, 160
 - defined, 60, 1165
 - DTC, enabling of, 240
 - Enlistment objects, 241
 - Ktmutil.exe tool for viewing, 241
 - logging operations, 260–261
 - NTFS use of, 240
 - purpose of, 240
 - resource manager (RM), 261–262
 - Resource Manager (RM) objects, 241
 - Tm object types, 137
 - Transaction Manager (TM) objects, 241
 - Transaction objects, 241
 - TxF with, 240, 973
 - TxR with, 240

L

- languages, single worldwide binary for, 24
- LANMan, 897–899
- large address space aware images, 699, 739, 746
- large pages, 705–706
- last known good control sets
 - BCD option for booting, 1082
 - defined, 1165
 - HKLM\SYSTEM subkey, 259
 - mechanics of, 1101
 - SCM management of, 308–309
 - screen crashes during boots, for correction of, 1113
- last processor numbers, 440
- latency, striped volumes for reducing, 663–664
- layered design of Windows, 38
- lazy evaluation, 719, 787
- lazy IRQ, 93, 1165
- lazy writing, 877–883, 906, 1165
- LCNs (logical cluster numbers)
 - data runs with, 950–951
 - defined, 1166
 - physical addresses, conversion to, 938
 - VCN-to-LCN mapping, 939–940
- LDAP (Lightweight Directory Access Protocol), 1011, 1067
- LDM (Logical Disk Manager)
 - partitioning
 - advantages of, 656
 - database for, 656–657
 - GPT or MBR with, 660
 - LDMDump, 658
 - viewing databases, 658
 - volume manager for, 661
- Ldr. *See* image loader
- LdrplInitializeProcess, 369
- LDTs (local descriptor tables), 64
- legacy drivers, 1166
- LFHs (Low Fragmentation Heaps), 732–733
- LFS (log file service), 976–978, 981
- licensing, determining features enabled by, 46–47
- lightweight filter drivers (LWDs), 1057
- line-based interrupts, 106
- Link-Layer Topology Discovery (LLTD), 1043
- link-local clouds, P2P, 1040
- Link-Local Multicast Name Resolution (LLMNR), 1039
- links, hard, 923–924
- lists, ExInterlocked functions for, 176–177
- LiveKd tool, 30–31, 1129
- LKG. *See* last known good control sets
- LLMNR (Link-Local Multicast Name Resolution), 1039
- LLTD (Link-Layer Topology Discovery), 1043
- lm kv command, 1144
- load and unload device drivers privilege, 510
- load balancing, network, 1068–1069
- loaded module database, 223–225
- loader data table entries, 223–225
- loader snaps, 221
- loader, image. *See* image loader
- local FDSs, 896–897, 1166
- Local group, SID for, 462
- local kernel debugging, 28, 1166
- local procedure calls
 - advanced. *See* ALPCs (Advanced Local Procedure Calls)
 - fast. *See* fast LPC
 - LPCs replaced by ALPCs, 203
- local replacement policies, 828
- local RPC, 1017–1018
- local security authentication server process. *See* LSASS (local security authentication server process)
- Local Security Authority Server (Lsasrv), 992–998
- Local Security Authority subsystem. *See* Lsass (Local Security Authority subsystem)
- local security policy, 511
- Local Security Policy Editor, 501, 511, 533–535
- local service account, 288, 291
- local system accounts
 - characteristics of, 289
 - defined, 288, 1166
 - group memberships of, 289
 - privileges of, 289–290
 - service applications running in, 288–290
 - Windows subsystem services in, 297
- locally unique identifiers. *See* LUIDs (locally unique identifiers)
- LocalService default service groupings, 314
- location awareness. *See* NLA (Network Location Awareness)
- locking pages in physical memory, 707–708
- log hives, 278–279, 1166
- log sequence numbers (LSNs), 914–915
- log start LSNs, 914
- logging
 - base log files, 913–914
 - boot logging in safe mode, 1114
 - cache manager recovery support, 853–854
 - change logging, 927–928
 - CLFS. *See* CLFS (Common Log File System)
 - common logging file system driver (Clfs.sys), 67
 - containers, 913–914
 - dedicated logs, 911–913
 - defined, 910, 1166
 - Event Log logging. *See* Event Log; Event Logger
 - event tracing for, 207–210
 - journaling, 1164
 - kernel logger, enabling, 208–210
 - log record structures, 910
 - LSNs (log sequence numbers), 914–915, 917
 - multiplexed logs, 911–913
 - NTFS log file writes, 940
 - security event log management privilege, 511
 - torn writes, 915
 - transaction logs. *See* CLFS (Common Log File System)
 - TxF logging, 973
- Logical Disk Manager (LDM)
 - partitioning, 656–660
- logical drives, 653–654
- logical network identity and interfaces, 1043
- logical prefetching
 - DLL loading by, 369
 - mechanics of, 823–827
 - prefetcher kernel component, 1166
 - process initialization, during, 363
- logical processors, 435
- logical sequence numbers (LSNs), 853
- logoffs, tracing with Process Monitor, 265–266
- logons. *See also* Winlogon
 - access check step, 518
 - account rights retrieval for, 502–503
 - authentication calls, 514
 - authentication of. *See* authentication components of, 513
 - credential providers for, 514
 - Ctrl+Alt+Delete logon attention, 453, 515–516
 - interface, displaying, 514. *See also* LogonUI
 - listing active sessions, 519–520
 - logon process, 1167. *See also* Winlogon manager for. *See* Winlogon

- overview of, 513–515
 - registry, reading of, 250
 - SCM, notification of success to, 308–309
 - secure logon facility rating, 452
 - single sign-ons, 515
 - tokens, created for, 473, 518
 - tracing with Process Monitor, 265–266
 - trusted path functionality, 453
 - user interface for. *See* LogonUI
 - user logon steps, 516–518
 - user profiles, loading, 520
 - LogonUI
 - credential provider use of, 79
 - defined, 455
 - network secondary authentication capabilities, 515
 - purpose of, 514
 - user logon steps, 516–518
 - LogonUser function, 481–483
 - look-aside lists, 728–729, 1167
 - low-IRQL synchronization
 - condition variables, 195–196
 - dispatcher headers, 182–185
 - event setting mechanism, 179
 - executive resources, 190–191
 - guarded mutexes, 189–190
 - kernel dispatcher objects. *See* dispatcher objects
 - keyed events, 186–187
 - KiWaitTest function, 179
 - mechanisms for, lists of, 178
 - purpose of, 177–178
 - pushlocks, 192–193
 - SRW locks, 196
 - wait blocks, 183–186
 - low-memory conditions
 - keyed events, 187–188
 - simulating to test drivers, 802
 - trimming working sets to free up, 829
 - LPCs (local procedure calls)
 - advanced. *See* ALPCs (Advanced Local Procedure Calls)
 - fast. *See* fast LPC
 - KSecDD functions for, 455
 - LRU (least recently used) algorithm, 827–828
 - LSA (Local Security Authority)
 - account right retrieval, 502–503
 - audit record messages to, 512
 - policy database, 1166
 - server process, 1166
 - Lsasrv (Local Security Authority Server), 992–999
 - LSASS (local security authentication server process)
 - access token generation, 80
 - defined, 80
 - logging on services, 305
 - mechanics of, 80
 - SCM calls to, 301–302
 - services contained by, 313
 - Lsass (Local Security Authority subsystem)
 - act as part of operating system privilege, 508, 510
 - audit responsibilities, 511
 - defined, 454, 1166
 - EFS management, 992
 - initialization, 457
 - KSecDD functions for, 455
 - local security policy, 511
 - policy database, 454
 - SAM service, 454
 - token creation by, 473
 - user logon steps, 516–518
 - Winlogon authentication calls to, 514
 - LSNs (log sequence numbers)
 - base LSNs, 973
 - defined, 1167
 - mechanics of, 914–915
 - restart LSNs, 914, 973–974
 - translating virtual of physical, 917
 - LSNs (logical sequence numbers), 853–854
 - LUIDs (locally unique identifiers), 476–477, 520
 - LWDs (lightweight filter drivers), 1057
- ## M
- magic packages, 1054
 - mailslots
 - defined, 1021
 - file system drivers for, 1025
 - implementation of, 1024–1026
 - operation of, 1023
 - UNC names for, 1021
 - Managed Object Format (MOF)
 - language, 319–323
 - management applications, 318–319
 - mandatory integrity checks, 492
 - mandatory integrity control, 22
 - mandatory labels, 471
 - mandatory policies, object, 473
 - mandatory policies, token, 476
 - manifests, assembly, 364–365
 - manual-reset events, 195
 - MAPI support, 1067
 - mapped files
 - ACLs for, 713
 - address ranges, viewing, 758
 - defined, 710
 - I/O of, 565–566, 1167
 - image loader with, 710
 - MapView functions, 710
 - prototype PTEs, 776–777
 - viewing with Process Explorer, 711
 - mapped page writer, 700, 812–813
 - MapUserPhysicalPages functions, 720
 - MapView functions, 710
 - marshalling areas, 1167
 - masks, access, 496–497
 - masks, active processor, 434
 - masks, affinity. *See* affinity masks
 - masks, IRQL, 93–94, 1167
 - Master Boot Record. *See* MBR (Master Boot Record)
 - master file tables, NTFS, 938–941
 - MBR (Master Boot Record)
 - bootable partition searches by, 1076
 - GPT disks, on, 654
 - partition tables, 1170
 - partitioning based on, 653–654, 660
 - transfer of control to, 1076
 - troubleshooting corruption issues, 1109
 - MBR partitioning, 653–654, 660
 - MCMs (miniport call managers), 1057–1058
 - MDLs (memory descriptor lists)
 - Direct I/O memory descriptions, 570–571
 - DMA caching use of, 872
 - Driver Verifier checks of, 803
 - dummy pages, 780
 - support routines, 762
 - Media Foundation API, 346–348
 - MemInfo tool
 - paging list dumps with, 806
 - PFN entries, 817
 - physical memory layout, 821
 - standby paging list sizes, 810–812
 - MemLimit utility, 755
 - memory
 - !vm command, 703
 - 64-bit address spaces, 15
 - address spaces for. *See* address spaces
 - amounts supported, 818–819
 - bad physical pages list, BCD, 1079
 - commit charge notification, 834
 - committed. *See* committed pages
 - diagnostics tool, 332
 - execution prevention. *See* DEP (data execution prevention)
 - file mapping objects, 12
 - flash, ReadyBoost for, 844–845
 - Hyper-V management of, 238–239
 - look-aside lists, 728–729
 - low, simulating to test drivers, 802
 - low, trimming to increase. *See* working set manager

- memory, *continued*
 - manager. *See* memory manager
 - mapped files. *See* mapped files
 - mapping virtual to physical. *See* address translation
 - mapping with PAE. *See* PAE (Physical Address Extension)
 - Memory bar histogram, 702
 - nonpaged, 570–571
 - notification events, 833–835
 - paging. *See* paging virtual memory to disks
 - paging defined, 1170
 - physical. *See* physical memory
 - Physical Memory counters, 702
 - pools. *See* pools, memory
 - priority numbers, 809–812
 - private address spaces for processes, 712, 758
 - proactive management of. *See* SuperFetch service
 - process page file quotas, 707
 - process virtual address spaces, 12
 - protecting, 711–713
 - quotas for types of processes, 756
 - RAM. *See* physical memory
 - registry hives, mapping of, 272–276
 - shared memory sections, 12
 - SuperFetch. *See* SuperFetch service
 - versions of Windows, amount supported by, 44–45
 - virtual implementation for processes. *See* virtual memory system
 - Write Copies/sec counter, 719
 - zeroing, 701, 808
- Memory and Process performance counters. *See* performance counters
- Memory Diagnostic Tool, Windows, 1106
- memory dumps
 - analysis. *See* crash dump analysis
 - kernel. *See* kernel memory dumps
- memory manager
 - boot initializations by, 1089, 1091
 - cache manager use of, 849–850
 - cache size issues, 855–859
 - cluster prefetches, 779–780
 - clustering, 1156
 - commitment, 1156
 - component overview, 700–701
 - copy-on-write page protection, 718–719
 - core services overview, 699–700
 - defined, 1167
 - demand paging, 787, 823
 - dereference segment threads, 701
 - DLL loading, 760
 - driver verifier option, 604
 - exception handler of, 700
 - execution prevention. *See* DEP (data execution prevention)
 - executive, as component of, 59, 700
 - flushing of cache pages, 877
 - granularity of allocations, 708
 - Http.sys with, 1020–1021
 - internal synchronization of, 701
 - kernel-mode system thread
 - components, 700–701
 - logical prefetching, 823–827
 - look-aside lists, 728–729
 - mapped file I/O, 1167
 - mapped page writer, 700
 - mapping virtual to physical memory, 699
 - Mm functions of, 704
 - modified and mapped page writer, 812–813, 906
 - modified page writer, 700
 - multiprocessor system capabilities, 701
 - notification events, 833–835
 - NUMA support, 791–792
 - page directories, 764–766
 - page fault handler, 906–907
 - page fault handling. *See* page faults
 - page file management. *See* page files
 - page priority handling, 809–812
 - page table creation, 762, 787–788
 - page units, 705–706
 - paging function of, 699
 - permissions for sections, 713
 - placement policies, 827–828
 - pools. *See* pools, memory
 - primary tasks of, 699
 - process handle permissions, 704
 - process/stack swapper
 - component, 700
 - prototype PTEs, 776–777
 - PTEs, setting bits in, 767
 - pushlocks in, 193
 - quotas, 756
 - replacement policies, 827–828
 - section objects. *See* section objects
 - services overview, 704
 - shared memory sections, 709–711
 - stop codes generated by, 1122
 - system thread use, 76
 - transparency of, 14
 - virtual address spaces. *See* virtual address space layouts
 - Windows API functions exposing services of, 704
 - working set manager, 700
 - zero page threads, 701
 - zeroing memory, 807–809
- memory-mapped executable issues, 1096
- memory-mapped file functions, 704
- Memtest.exe, 1075
- Message Queuing API, 1032
- message-based interrupts, 106
- metadata
 - base log file structure, 913–914
 - defined, 890, 1167
 - extensions metadata directory, NTFS, 941
 - logging by NTFS recovery support system, 976–980
 - NTFS metadata, 938, 947–948
 - update steps ensuring recovery, 853
- methods, object. *See* object methods
- MFTs (master file tables), NTFS
 - defined, 1167
 - mechanics of, 938–941
 - mirrors, 1167
 - TxF data stored in, 971
 - viewing information about, 941
- microkernel-based operating systems, 34–35
- Microsoft Developer Network (MSDN), 1168
- Microsoft Interface Definition Language (MIDL) compiler, 1016
- Microsoft Systems Center 2007, 121
- Microsoft Transaction Server (MTS), 1032
- MIDL (Microsoft Interface Definition Language) compiler, 1016
- MilInitialize functions, 751
- MiModifiedPageWriter, 812–813, 906
- minidumps, 1127–1128
- miniport drivers
 - defined, 543, 1167
 - Flags field, 1056
 - LANMan Redirector, 897
 - listing loaded, 1055
 - MCMs (miniport call managers), 1057–1058
 - NDIS, 1004, 1053
 - partitioning information, 651–652
 - place in disk driver architecture, 647
 - Remote NDIS, 1060–1062
 - WebDAV, 897
 - Windows-supplied, 648
- mirror sets, 1167
- mirrored volumes, 664–666, 675, 1167

MiSystemVaType array, 753
 MiSystemVaTypeCountLimit array, 755
 mklink utility, 923–926
 Mm functions, 704, 708
 MMCSS (MultiMedia Class Scheduler Service), 420, 430–434
 modes
 kernel. *See* kernel mode
 performance counters for, 17–19
 purpose of, 16
 switching between by applications, 17
 user. *See* user mode
 viewing in Task Manager, 18
 modified and mapped page writer, 700, 812–813, 906, 1167–1168
 Modified no-write page state, 804
 Modified page state, 804
 modified page writer, 700, 812–813, 906
 Mount Manager, 667–674
 mount points, 669–670, 1168
 mounting devices
 defined, 1168
 dismount operations, 897
 Plug and Play with, 668
 registry storage of data for, 667–668
 removable media, 674
 reparse points, 669
 types of devices mounted, 667–668
 volume mounting, 670–671, 939
 VPBs (volume parameter blocks), 670–673
 MoveFileEx API, 1096
 MP Specification, 89–90
 MPIO (Multipath I/O) drivers, 648–649
 MPR (Multiple Provider Router), 303, 1034–1036
 MS DTC (Microsoft Distributed Transaction Coordinator), 1032
 Msconfig utility, 1100
 MSDN (Microsoft Developer Network), 1168
 MS-DOS
 file names, 945–947
 granularity of memory allocations, 708
 kernel support for, 64
 support images for process creation, 352–353
 symbolic link creation by Smss, 1094
 Msinfo32, 71, 546
 MSV1_0, 513, 517–518
 MTS (Microsoft Transaction Server), 1032

multimedia playback priority boosts, 420
 multipartition volumes, 645, 661–667, 1168
 Multipath I/O (MPIO), 648–649
 multipathing, 1168
 multiple core system thread scheduling, 435
 Multiple Provider Router. *See* MPR (Multiple Provider Router)
 multiple redirector support, 1033–1038
 multiple sessions, Terminal Services for, 19–20
 Multiple UNC Provider. *See* MUP (Multiple UNC Provider)
 multiprocessor systems. *See also* SMP (symmetric multiprocessing)
 boot initialization of extra processors, 1091
 fast mutexes, advantages of, 189
 heap scaling issues, 732
 idle threads, 418–419
 IPLs (interprocessor interrupts), 96
 memory manager capabilities for, 701
 number of supported processors, 40
 scalability, 43
 thread scheduling. *See* multiprocessor thread scheduling
 unified kernel, 41
 versions of Windows, amount supported by, 44–45
 multiprocessor thread scheduling
 active processor masks, 434
 affinity masks, 438–440
 compared to uniprocessor systems, 434
 dispatcher database considerations, 434–435
 dynamic processor issues, 441–442
 ideal processor numbers, 440–441
 idle processors, scheduling while there are, 443
 idle summary masks, 434
 IRQL issues, 434–435
 last processor numbers, 440
 logical processors, 435
 no idle processors, choosing when, 443–444
 NUMA systems, 436–438
 per-CPU deferred ready state thread list, 435
 specific CPUs, selecting threads for, 444

multitasking. *See also* SMP (symmetric multiprocessing)
 defined, 39
 preemption scheduling scenario, 414
 voluntary switch scenario, 413
 multithreading
 atomic execution problem, 197
 ideal processor numbers with, 440
 multiprocessor support for, 43
 MUP (Multiple UNC Provider)
 defined, 1034
 MPR use of, 1036
 operation of, 1037–1038
 mutants. *See* mutexes
 mutexes
 critical sections, inside, 194–195
 defined, 1168
 executive mutexes. *See* fast mutexes
 fast mutexes, 189–190
 guarded mutexes, 189–190
 Mutex object type, 137
 OS/2 history of, 137
 purpose of, 182
 mutual exclusion, 170–171, 1168.
 See also synchronization

N

name resolution
 defined, 1039
 DNS. *See* DNS (Domain Name System)
 LLMNR (Link-Local Multicast Name Resolution), 1039
 Peer Name Resolution Protocol (PNRP), 1039–1041
 WINS, 1039
 Winsock name-resolution functions, 1009
 name retention, 159, 1168
 named pipes
 defined, 1021
 driver for, protected prefix for, 1094
 file system drivers for, 1025
 format for names, 1021
 functions for, 1021–1022, 1024
 impersonation for, 481, 1022
 implementation of, 1024–1026
 mode options, 1021–1022
 operation of, 1021–1022
 PipeList tool, 1025–1026
 reading and writing functions, 1022
 RPC with, 1016–1018
 servers, 1021
 transactions with, 1022
 UNC names for, 1021
 viewing, 1025–1026

- names of objects
 - base object names, listing, 165
 - case sensitivity of, 163
 - directories, 163
 - global visibility of, 162
 - handles, relation to, 163
 - kernel objects, 163–164
 - management of, 162–166
 - network visibility of, 164
 - private namespaces, 164
 - session namespaces, 167–169
 - single-instancing with named objects, 166
 - squatting attacks, 164
- namespaces
 - BaseNamedObjects, 167
 - DosDevices, 167–168
 - global, 167
 - instancing, 167, 1163
 - Key object type, 137
 - local, 167
 - private namespaces, 164–167
 - registry, 276
 - session namespaces, 167–169
 - virtualization, UAC, 521–528
 - volume, 667
 - Winsock, adding service providers to, 1009
 - WMI, 324–325
- NAP health certificates, 1051
- NAT (network address translation), 1049–1050
- National Computer Security Center (NCSC), 451
- native applications, 1168
- native system services, 4, 51, 130–133
- Nbtstat command, 1028
- NCSC (National Computer Security Center), 451
- NDIS (Network Driver Interface Specification)
 - activity awareness of drivers, 1054
 - call managers, 1057–1058
 - components of, 1053–1055
 - connection-oriented (CoNDIS), 1055, 1057–1060
 - drivers, 1053–1054
 - header-data split capability, 1054
 - interfaces for drivers, 1055
 - intermediate drivers, 1057
 - IPSec with, 1054
 - library (Ndis.sys), 1004, 1053–1054
 - lightweight filter drivers (LWDs), 1057
 - listing loaded miniports, 1055
 - load balancing drivers, 1068–1069
 - MCMs (miniport call managers), 1057–1058
 - miniport drivers, 1004, 1053, 1056
 - Network Monitor, 1059–1060
 - protocol drivers, 1003–1004, 1053–1054
 - purpose of, 1053
 - receive scaling ability, 1054
 - Remote, 1060–1062
 - run-time configuration capability of, 1054
 - TCP/IP offloading, 1054
 - wake-on-LAN, 1054
- neither I/O, 571
- NetBIOS (network basic I/O system)
 - API implementation, 1029
 - disadvantages of, 1027
 - LAN adapter (LANA) numbers, 1027–1028
 - name system, 1027
 - Next Generation TCP/IP Stack with, 1045
 - operation of, 1028
 - routing scheme for, 1028
 - sessions, 1028
 - TCP/IP, over, 1029
 - viewing names with Nbtstat, 1028
 - WINS name resolution for, 1039
 - Winsock Helper library for, 1011
- Netlogon, 455, 518
- Netsh.exe, 1010, 1052
- network adapters
 - header-data split capability, 1054
 - logical network interfaces for, 1043
 - NDIS miniport drivers, 1004
 - wake-on-LAN, 1054
- Network and Sharing Center, 1034
- network drives, SCM notification role, 303
- network file system, 872, 1168
- Network Load Balancing, 1068–1069
- Network Location Awareness (NLA), 1011, 1042–1043
- network logon service, 1168
- Network Map functionality, 1043
- Network Monitor, 1059–1060
- network protocol drivers. *See* protocol drivers
- network provider interfaces, 1035
- network redirectors and servers, 68, 852, 885–886, 1168
- network service account, 288, 290–291
- networking
 - adapters. *See* network adapters
 - APIs for. *See* networking APIs
 - architecture for, overview of, 1001
 - binding, 1064–1065
 - BITS (Background Intelligent Transfer Service), 1030
 - components for, list of, 1003–1005
 - datagrams, 1008, 1013
 - diagnostics for, 332
 - file system drivers, network, 1168
 - filtering for. *See* WFP (Windows Filtering Platform)
 - goal for, 1001
 - Helper libraries for protocols with Winsock, 1011
 - IP filtering, 1049–1050
 - IPSec (Internet Protocol Security), 1050–1052
 - layers, OSI, 1002–1003
 - LLTD (Link-Layer Topology Discovery), 1043
 - location awareness. *See* NLA (Network Location Awareness)
 - mailslots for. *See* mailslots
 - Message Queuing API, 1032
 - MUP for. *See* MUP (Multiple UNC Provider)
 - named pipes for. *See* named pipes
 - NAT, 1049–1050
 - NetBIOS API. *See* NetBIOS (Network Basic I/O System)
 - network layer, OSI, 1002
 - Network Monitor, 1059–1060
 - NetworkService default service groupings, 314
 - OSI Reference Model, 1001–1003
 - Peer-to-Peer Infrastructure APIs, 1031
 - protocol drivers. *See* protocol drivers
 - QoS (Quality of Service), 1062–1064
 - redirectors. *See* network redirectors and servers
 - remote access, 1066
 - remote procedure call API. *See* RPC (remote procedure call)
 - request-reply model, 1001
 - restriction rules, table of, 297
 - software types in stacks, 1001
 - stacks for. *See* stacks, network
 - TCP/IP protocol. *See* TCP/IP
 - TDI clients. *See* TDI (Transport Driver Interface) clients
 - topology discovery. *See* LLTD (Link-Layer Topology Discovery)
 - transport service providers, adding, 1009
 - UPnP (Universal Plug and Play), 1032–1033
 - Web access. *See* Web access APIs
 - well-known addresses for Winsock, 1009
 - Winsock API for. *See* Winsock
 - WNet API, 1033–1036
- networking APIs
 - kernel. *See* WSK (Winsock Kernel)
 - list of, 1006

- mailslots. *See* mailslots
- named pipes. *See* named pipes
- NetBIOS. *See* NetBIOS (network basic I/O system)
- purpose of, 1003
- RPC. *See* RPC (remote procedure call)
- sockets. *See* Winsock
- Web access. *See* Web access APIs
- Next Generation TCP/IP Stack, 1012–1013, 1044–1045, 1053–1054
- NLA (Network Location Awareness), 1011, 1042–1043
- NLS for internationalization, 222
- NMR (Network Module Registrar), 1012–1013
- no execute page protection, 713–718. *See also* DEP (data execution prevention)
- Nobody group, 462
- nodes, NUMA, 436–438
- nonpaged memory, 570–571
- nonpaged pools
 - corruption of, 1140–1142
 - defined, 721, 1168
 - initial sizes of, 722
 - location of, 721
 - monitoring usage of, 724–728
 - notification events, 834
 - number of, 721–722
 - quotas for, 756
 - reservation of addresses for, 751
 - sizes of, 722–724
 - system, 737
 - system space addresses for, 752
- Non-Plug and Play drivers
 - defined, 542
 - device object creation, 550–551
 - legacy drivers, 620–621
 - Process Monitor device driver mechanics, 262
 - Start values, 623–624
- non signaled state, 179
- nonuniform memory access
 - architecture. *See* NUMA (Non Uniform Memory Architecture)
- No-Read-Up object policy, 473, 492
- notification events, 195, 833–835
- Notmyfault tool
 - Buffer Overflow bug, creating, 1141–1142
 - code corruption from drivers, 1143–1144
 - driver information commands, 1138–1139
 - Hang option, 1149–1150
 - High IRQL Fault option, 1135–1137
 - launching, 1134
 - pool leak detections, 727–728
 - Stack Crash option, 1145–1147
 - stack traces of executing threads, 1138
 - stop codes shown by, 1137
 - unkillable processes, debugging, 590–592
 - verbose analysis with, 1137–1139
- NtCreateFile system service, 902
- NtCreateUserProcess, 349, 351–352, 360
- Ntddk.h ASSERT macro, 48
- Ntdll.dll
 - defined, 37, 1168
 - functions in, 57–58
 - heap manager functions, 730
 - hotpatching, 242
 - image loader, 220
 - NtCreateFile function, 557
 - purpose of, 57
 - system services functions, 58
 - undocumented interfaces, viewing, 72–73
 - user-mode debugging role, 217–218
- NTFS (NT File System)
 - \$DATA attribute, 943–944
 - advanced feature list, 920
 - advanced feature overview, 895
 - atomic transactions, 918–919
 - attribute lists, 1154–1155
 - attributes, file, 940–944, 948–951
 - bad clusters, 940–941, 985–988
 - base file records, 938
 - bitmap file of allocation state, 940, 1155
 - boot file, 940
 - cache manager recoverable file system support, 853–854
 - cache manager with file system driver, 935
 - change journal files, 941, 956–959
 - change logging, 927–928
 - clusters for, 895, 937–938
 - compression support, 927, 951–956
 - data attribute, 943
 - default resource manager directory, 941
 - defined, 895
 - defragmentation support, 931–932
 - design goals of, 918–920
 - dynamic bad-cluster remapping, 923
 - dynamic partitioning, 932–933
 - encryption support. *See* EFS (Encrypting File System)
 - extensions metadata directory, 941
 - FCBs (file control blocks), 936
 - file names, 945–948
 - file objects and handles, 935–936
 - file records, 938, 942–944
 - file reference numbers, 942
 - file system driver of, 934–936
 - Filename attribute, 943
 - hard links, 923–924
 - I/O manager interactions with file system driver, 934–935
 - indexing facility of, 923, 943, 960–961
 - KTM update support for, 240
 - LCNs (logical cluster numbers), 938–940, 950–951
 - local file system driver for, 896–897
 - log file writes, 940
 - metadata files, 938
 - MFTs (master file tables), 938–941, 971
 - mirror, MTF, 939
 - mount points with, 669–670
 - mounting volumes, 939
 - MS-DOS file names, 945–947
 - nonresident attributes, 948–951
 - Ntfs.sys, 896
 - object IDs, 929–930, 941, 944, 961
 - OLE linking, 929–930
 - per-user volume quotas, 928
 - physical disk address calculations, 938
 - POSIX support, 931
 - quota files, 941, 962–963
 - recovery support. *See* NTFS recovery support
 - redundancy support, 919–920
 - reparse points. *See* reparse points
 - resident attributes, 948–951
 - rollback operations, 918–919
 - root directory records, 940
 - SCBs (stream control blocks), 936
 - sector relationship to clusters, 938
 - security descriptor attribute. *See* security descriptors
 - security file, 940
 - security model of, 919
 - self-healing feature, 895, 989
 - shortcuts, shell, tracking support for, 929–930
 - shrinking engine, 932–933
 - spanned volumes, 662–663
 - sparse data compression, 952–954
 - sparse files, 927, 956
 - stop code for fatal errors, 1123
 - stream-based caching, 852
 - streams, 920–922
 - symbolic links, 924–926
 - tunneling file metadata, 947–948
 - TxF. *See* TxF (Transactional NTFS)

1210 NTFS recovery support

NTFS (NT File System), *continued*
Unicode-based names, 922–923
uppercase file, 940–941
VCNs (virtual cluster numbers),
938–940, 950–951
viewing volume information with
Fsutil.exe, 941
volume file, 940–941
volumes, 937

NTFS recovery support
analysis passes, 981–982
atomic transaction basis of,
918–919
bad-cluster recoveries, 985–988
batching, 975
cache flushing operations, 975,
977
checkpoint records, 978–980
committed transactions, 978–979,
984
dirty page tables, 981–985
errors, file system, 984–985
file system corruption, recovery
from, 986–988
guaranteed consistent states, 984
importance of, 895
journaling, 975
lazy writes, 975
LFS (log file service), 976–978
LFS calls for recovery, 981
limits without TxF, 974
log file resets, 980
log file size issues, 980
log record types, 978–980
mechanics of recoveries, 981–985
metadata logging by, 976–980
overhead from, 975
overview of, 974
recoverable file system design,
975
redo entries, 978–979
redo passes, 982–983
reliability strategies of, 975
rollbacks, 978–979
scenarios, table of, 988
self-healing feature, 989
transaction tables, 981–985
TxF for. *See* TxF (Transactional
NTFS)
types of transactions tracked, 979
undo entries, 978–979
undo passes, 983–984
update records, 978–981
write-through operations, 975

NtGlobalFlag variable, 200–202
Ntkrnlmp.exe, 1169
Ntkrnlpa.exe, 37, 770
Ntoskrnl.exe
binaries linked to, 67
boot phase mechanics, 1088–1093

defined, 37, 1075, 1169
executive functions in, 58
executive of, 1160
HAL dependencies, 66–67
heap manager functions, 730
memory manager, 700
undocumented interfaces,
viewing, 72–73
version, checking which is
running, 41–42

NtReadFile function, 1143

NUMA (Non Uniform Memory
Architecture)
defined, 40
functions for application node
choice, 791
ideal nodes selection, 791–792
ideal processor numbers with, 440
mechanics of, 791–792
memory manager support for,
791–792
node graph construction, 791
page pool dependence on,
721–722
thread synchronization
mechanisms, 436–438

O

object attributes
defined, 21, 1169
flags, table of, 140–142
ObjectAttributes parameter, 21
type-specific, 143–147

Object Browser, WMI, 325–326

object directories, 1169

object headers, 138–140, 144, 162

object IDs, NTFS
application use of, 961
attribute for, 944
defined, 929–930
storage of, 941

object manager
access checks, 459–461
attribute flags, table of, 140–142
boot initializations, 1090
C2 security compliance, 134
close object method, 148
default object type, 146–147
defined, 21, 60, 1169
delete object method, 148
executive. *See* executive objects
filtering, 170
flags, table of, 140–142
generic services, 142–143
goals of, 134–135
granted access rights, 157
handles, object, 149–157. *See also*
handles
header fields, 138–140

headers, accessing, 142
methods, object, 147–149
names management, 162–166
namespaces, 133
naming scheme, 134
network I/O calls, 1036
object retention, 159–161
object security. *See* object security
object-tracking debug flag, 143
okay to close object method, 149
open object method, 148
Openfiles /query command
handle display, 133–134
parse object method, 148–149
purpose of, 133
pushlocks in, 193
resource accounting, 161–162
resource management role, 134
retention of objects goal, 135
section object management, 793
security descriptor management,
488
security object method, 149
security role of, 458
services, non-generic, 142
session namespaces, 167–169
session object isolation goal, 135
subheaders, 139–140
subheaders, accessing, 142
symbolic links, 167
synchronization, 146–147, 179
tools for viewing, 133
type creation issues, 147
type initializer fields, 146
type objects, 143–147
types with, 142
WinObj tool, 133

object methods
close object method, 148
defined, 21, 147, 1169
delete object method, 148
extensibility, 147
object manager with, 147–149
okay to close object method, 149
open object method, 148
parse object method, 148–149
security object method, 149
type creation issues, 147

Object Reference Tracing, 156

object retention, 159–161

object security
access checks for. *See* access
checks
access determination mechanics,
492–501
access masks, 496–497
ACLs, assignment to new objects,
490–491
ACLs, displaying for objects,
157–158

- AuthZ API, 500–501
 - denial of service attacks, 164
 - desired access rights, 157
 - Ex versions of API vs. open API, 159
 - granted access rights, 157
 - integrity levels, 471–473
 - mandatory policies, 473
 - object manager role in, 458
 - overview of, 458–459
 - reuse protection, 452
 - security descriptors for. *See* security descriptors
 - security reference monitor, 157
 - squatting attacks, 164
 - tokens, 473–480
 - UIPI for user processes, 493–494
 - object types
 - advantages of, 142
 - defined, 21, 1169
 - type data structures, viewing, 145
 - type object header field, 139
 - Object Viewer, 794
 - object-oriented kernel code, 35
 - objects
 - accessing without handles, 150
 - attributes. *See* object attributes
 - bodies of, 142
 - control objects, 62, 1157
 - creator information subheader, 139–140
 - data structures, compared to, 21
 - default object type, 146–147
 - deferred deletion of, 160
 - defined, 21, 1169
 - deletion of, 160
 - executive, 135
 - filtering, 170
 - Flags type initializer field, 146
 - flags, table of, 140–142
 - GDI/User objects, 135
 - global object create privilege, 506
 - handle information subheader, 139–140
 - handles for, 1169. *See also* handles
 - headers. *See* object headers
 - integrity levels of, 471–473
 - kernel, 61–62. *See also* kernel objects
 - labels, modification privilege, 507
 - manager. *See* object manager
 - mandatory labels, 471
 - methods. *See* object methods
 - name information subheader, 139–140
 - names management, 162–166
 - ObjectAttributes parameter, 21
 - ObpCreateHandle function, 459
 - okay to close object method, 148–149
 - opening by name, 459
 - owner rights of, 495–497
 - permanent, 159
 - pointers to. *See* pointers, object
 - quota charges, 162
 - quota information subheader, 139–140
 - reference counts, 160
 - retention of, 159–161
 - reuse protection, 1169
 - security. *See* object security
 - security descriptors of. *See* security descriptors
 - signaled state effects by type, 180
 - structure overview, 138
 - subheaders, 139–140
 - tasks performed by, 21
 - temporary, 159–161
 - type data structures, viewing, 145
 - type objects, 143–147
 - types. *See* object types
 - ObpCreateHandle function, 459
 - ObRegisterCallback API, 170
 - OCA (Online Crash Analysis), 1133–1134
 - OCI (open cryptographic interface), 1017
 - okay to close object method, 148–149
 - OLE (object linking and embedding), 929–930
 - one-time initialization. *See* run once initialization
 - Online Crash Analysis (OCA), 1133–1134
 - open cryptographic interface (OCI), 1017
 - open object method, 148
 - Open Systems Interconnection (OSI) reference model, 1001–1003
 - OpenEvent for access checks, 159
 - Openfiles /query command handle display, 133–134
 - operating system model, 34–35
 - oplock protocol, 898–899
 - Orange Book, 451
 - OSI Reference Model
 - correlation with Windows APIs, 1004–1005
 - defined, 1001–1002
 - layers in, 1002–1003
 - protocol stacks, 1003
 - OTS (over-the-shoulder) elevation, 528–533
 - output. *See* I/O system
 - over-the-shoulder (OTS) elevation, 528–533
 - owner rights of objects, 495–497, 508, 510
- ## P
- P2P IDs, 1040
 - packet filtering, IPSec for, 1050
 - PAE (Physical Address Extension)
 - address translation example, 770–771
 - BCD options for, 1082–1083
 - defined, 37, 1170
 - DEP requirements, 714
 - device driver issues, 819–820
 - internal system file names, table of, 41
 - kernel differences when present, 41
 - mechanics of, 769–771
 - Ntkrnlpa.exe, 770
 - Ntoskrnl.exe, checking version of, 41–42
 - page directory pointer tables, 769–771
 - page tables for, 763
 - purpose of, 769
 - page directories
 - address translation mechanics, 764–766
 - CPU register for, 764–765
 - defined, 764, 1169
 - indexes, 763
 - locating for the current process, 764
 - locating PDEs (page directory entries), 764
 - structure of, 765
 - viewing address of for running processes, 765
 - virtual addresses of, viewing, 765
 - widths for, 763
 - x64, 773
 - page directory pointer tables, PAE, 769–771
 - page faults
 - clustered, 779–780
 - collided page faults, 779
 - defined, 774, 1169
 - demand paging triggered by, 823
 - demand zero faults, 775, 807–809
 - in-page I/O operations from, 778–779
 - invalid PTE basis of, 774
 - invalid PTE types, 775–776
 - page file access faults, 775
 - prototype PTEs, 776–777
 - PTE unknown faults, 776
 - reasons for, table of, 774–775
 - replacement policies, 827–828
 - signaling page table creation with, 787–788
 - transition faults, 775

1212 page files

- page files. *See also* paging virtual memory to disks
 - adding new, 782
 - clearing at shutdown, 781
 - commit limit, viewing, 783
 - committed memory, relation to, 710
 - criteria for sizing, 782–783
 - default sizes, table of, 781
 - defragmenting, 781
 - locking with AWE, 721
 - maximum sizes of, 781
 - performance counter meaning, 780–781
 - performance counters for, 782
 - potential page file usage, viewing, 783
 - Process Explorer data on, 784
 - purpose of, 780–781
 - quotas for, 756, 781, 1169
 - registry tracking of, 781
 - Session Manager reads during boots, 781
 - system running on virtual memory errors, 783
 - viewing system page files, 782
 - viewing usage with Task Manager, 783
- page frame numbers. *See* PFNs (page frame numbers)
- page tables
 - creation by memory manager, 762
 - defined, 1169
 - entries. *See* PTEs (page table entries)
 - indexes for locating PTEs, 763, 766
 - kernel-mode-only marking of, 736
 - lazy evaluation for construction of, 787–788
 - locating PTEs in indexes, 764
 - locating with page directory indexes, 763
 - maximum number of, 765
 - mechanics of translations, 766–767
 - number required to cover address space, 766
 - page directories of. *See* page directories
 - private address space maps, 765–766
 - structure of, 763
 - x64, 773
- paged pools
 - corruption of, 1140–1142
 - defined, 721, 1169–1170
 - limiting size of, 755
 - location of, 721
 - monitoring usage of, 724–728
 - notification events, 834
 - number of, 721–722
 - quotas for, 756
 - sizes of, 722–724
 - system paged pool, 737
 - system space addresses for, 752
- Pagedefrag tool, 781
- page-file-backed sections, 710, 1169
- pageheap feature, 735
- pages
 - bad, notification of, 834
 - committing, 706–708
 - copy-on-write page protection, 718–719
 - creation for initial process address space, 355–356
 - defined, 705
 - disk writes of, 707. *See also* paging virtual memory to disks
 - execution prevention. *See* DEP (data execution prevention)
 - invalid, references to. *See* page faults
 - large page advantages, 705–706
 - large page security issue, 706
 - locked memory page tracking, 801
 - locking in physical memory, 707–708
 - memory manager modified and mapped page writer, 906
 - memory protection options, 712–713
 - owner pages, CLFS, 915–916
 - page fault handler, 906–907
 - page-related structure allocation, 753
 - pool corruption of, 1140–1142
 - prefetched, 779–780
 - priority numbers, 809–812
 - private, 706–707
 - reserving, 706–708
 - shared, 776–777
 - sizes of, 705–706
 - states, table of, 804
 - synchronization objects in bug, 802
 - tables of. *See* page tables
- paging. *See* paging virtual memory to disks
- paging files. *See* page files
- paging virtual memory to disks
 - !vm command, 703, 1145
 - address translation, 705–706
 - Bootmgr enabling of, 1077
 - cache management for, 701
 - defined, 14, 1170
 - dereference segment threads, 701
 - files on disk. *See* page files
 - free page requests, 700
 - freeing virtual memory pages, 707
 - in-page I/O operations, 778–779
 - Kernel Memory counters, 702
 - mapped page writer, 700
 - memory manager role in, 699–700
 - modified page writer, 700
 - purpose of, 14
 - System Page File counter, 702
 - writing pages to, 707
 - zero page threads, 701
- parse object method, 148–149
- partition manager
 - multipartition volume management, 661–667
 - PnP coordination, 655
 - responsibilities of, 651–652
 - SAN policy, 670–671
- partition tables, 1170
- partitions, disk
 - active partitions, 653–654
 - boot sectors of, 653–654, 1073–1077
 - defined, 645, 1170
 - dynamic partitioning, 932–933
 - EFI (Extensible Firmware Interface), 654
 - EFI System partitions, 1087
 - extended, 653–654
 - GPT partitioning, 654
 - hard versus soft, 660
 - hidden, 652
 - IDs of, 652
 - LDM partitioning. *See* LDM (Logical Disk Manager) partitioning
 - manager for. *See* partition manager
 - manager
 - MBR partitioning, 653–654
 - mirror sets, 1167
 - partition tables, 653
 - primary partitions, 653
 - spanned volumes, 662–663
 - stored key conversion to volume drive letters, 1095
 - striped volumes, 663–664
 - volume equivalence for basic disks, 655
 - volume possession mechanics, 652
 - Windows installations, partitioning during, 660
- partitions, hypervisor
 - child partitions, 230, 232–234
 - overview of, 230
 - root partitions, 230–232
- passwords
 - credential providers, 79
 - user logon request mechanics, 516–517

- patches, 242–246
- PatchGuard, 244–246
- pause assembly instructions,
 - spinlocks with, 174
- PCA (Program Compatibility Assistant), 333
- PCR (processor control region)
 - defined, 1171
 - IRQL values saved to, 95
 - viewing contents of, 95
- PCRs (platform configuration registers), 681–683
- PDEs (page directory entries)
 - defined, 1169
 - locating, 764
 - PAE address translation example, 770–771
 - PAE width for, 769–771
 - PFNs in, 764
 - structure of, 765
- PDOs (physical device objects), 628, 1160–1161, 1170
- PEB (process environment block)
 - address space for, 335
 - CreateProcess setup of, 357–358
 - dumping, 342
 - initial value fields, table of, 357–358
 - purpose of, 340–341
 - structure of, 341
 - Windows replacements for initial value fields, table of, 358
- Peer Name Resolution Protocol (PNRP), 1039–1041
- Peer-to-Peer Infrastructure APIs
 - defined, 1170
 - mechanics of, 1031
- per processor flags, 175–176
- performance
 - counters. *See* performance counters
 - diagnostics, 332
 - I/O caching. *See* cache manager
 - quantums, controlling, 408–410
- performance counters
 - cache mapping and pinning, 871
 - cache MDL activity, 872
 - cache read activity fields, 869–870
 - Data Maps/sec, 884
 - fast I/O, 874–875
 - flush operation fields, 883
 - HKEY_PERFORMANCE_DATA (HKPD), 259
 - Kernel Memory counters, 702
 - lazy write counters, 877, 884
 - Memory: Free System Page Table Entries, 744
 - Memory: Write Copies/sec counter, 719
 - memory-related, table of, 702–703
 - mode-related, 17–19
 - page file, table of, 782
 - Performance Data Helper (PDH)
 - functions, 259
 - Physical Memory counters, 702
 - pool size counters, 724
 - Process: Page File Bytes counter, 780–781
 - process related, table of, 343–344
 - Read Aheads/sec, 875–876
 - System Page File counter, 702
 - thread related, table of, 379–380
 - working set size counters, 830
- Performance Monitor
 - counter descriptions, viewing, 25
 - CPU starvation boosts, watching, 428–430
 - disk monitoring with, 665–666
 - mode related counters, 17–19
 - priority boosts, viewing, 423–425
 - starting as Reliability and Performance Monitor, 25
 - system threads, finding running, 76
 - thread execution states, viewing, 402–404
 - working set size counters, 830
- performance options, applying to, 355
- permanent objects, 159
- permissions. *See* access control
- PFNs (page frame numbers)
 - !memusage command, 807
 - Active state, 804
 - Bad state, 804
 - collided page faults processing, 779
 - Color field, 815
 - database structure of, 804
 - defined, 1169
 - field in PTEs, 766
 - fields of, variance of, 814–817
 - flags, 815–816
 - Free state, 804
 - I/Os in progress, 816
 - kernel stack field, 816
 - list link field, 816
 - lists of pages by state, 805
 - Modified no-write state, 804
 - Modified state, 804
 - Original PTE contents field, 815
 - page list mechanics, 807–809
 - page location step, 764
 - page priority handling, 809–812, 815
 - page states, table of, 804
 - PDEs, inside of, 764
 - PFN of PTE field, 815
- physical memory limits, as reason for limiting, 818–819
- PTE address field, 814
- PTE changes from, 809
- purpose of, 803
- reference count field, 814
- Rom state, 804
- share count field, 816
- Standby state, 804, 809–812
- synchronized access to database of, 701
- system space addresses for, 752
- Transition state, 804
- Valid state, 804
- viewing the database, 806–807
- viewing the entries in, 817
- working set index field, 816
- zero page threads, 808
- zero pages, 804, 807–808
- PGM (Pragmatic General Multicast) Winsock Helper library, 1011
- physical device objects. *See* PDOs (physical device objects)
- physical disks. *See* disks
- physical layer, OSI, 1003
- physical memory
 - !filecache debugger command, 857
 - !vm command, 703
 - 32-bit client limits to useful, 820–821
 - amounts supported, 818–822
 - availability, viewing with Task Manager, 821
 - AWE functions, 719–721
 - cache size in, 858–859
 - committed virtual pages for, 706–708
 - copy-on-write page protection, 718–719
 - DMA caching, 872
 - free, notification events, 834
 - freed memory checks, 802–803
 - granularity of allocations, 708
 - layout, viewing, 821
 - limitations, 818–819
 - locking pages in, 707–708
 - low, trimming to increase. *See* working set manager
 - mapping virtual memory system to. *See* address translation
 - Memory bar histogram, 702
 - nonpaged pools, 721
 - optimization software, 846–847
 - paging defined, 1170
 - PFN databases of. *See* PFNs (page frame numbers)
 - Physical Memory counters, 702
 - priority numbers, 809–812
 - replacement policies, 827–828

- physical memory, *continued*
 - reserved, viewing, 821–822
 - shared memory sections, 709–711
 - SuperFetch. *See* SuperFetch service
 - system variables describing, 818
 - system working sets in, 856–857
 - versions of Windows, amount supported by, 44–45
 - virtual address space in. *See* working sets
 - working sets of. *See* working sets
- PICs (Programmable Interrupt Controllers), 89–91
- PIDs (product IDs), 1158
- PipeList tool, 1025–1026
- placement policies, 827–828
- Plug and Play
 - drivers. *See* Plug and Play drivers manager
 - manager. *See* Plug and Play manager
 - PnP-X (Plug and Play Extensions), 1032–1033
 - UPnP (Universal Plug and Play), 1032–1033
- Plug and Play drivers
 - boot initialization, 1090
 - defined, 542
 - device objects, 550–551
 - load order, 631–632
 - loading responsibility, 619
 - manager, support for, 621–622
 - signing policies, 634–636
 - UMDF reflectors for, 617
- Plug and Play manager
 - capabilities provided by, 619–620
 - commands to drivers, 621–622
 - compatible IDs, 634
 - defined, 538, 1170
 - device identifiers, 630
 - device trees, 625–628
 - devnodes of trees. *See* devnodes
 - digital signing policy, 247
 - DIIDs (device instance IDs), 630
 - docking station removal privilege, 508
 - driver loading responsibility, 619
 - driver signing policies, 634–636
 - driver support for, 621–622
 - enumeration of devices, 624–628
 - filter driver support for, 621
 - hardware IDs, 634
 - Hardware Installation Wizard calls, 632
 - inf files, 632–634
 - installation of drivers, 632–636
 - instance IDs, 630
 - interrupt chaining, 105
 - levels of support for non-PnP devices, 620–621
 - load order of drivers, 626–627
 - Mount Manager registration with, 668
 - network device support, 620
 - notifications, 621
 - protected driver lists, 636
 - purpose of, 619
 - query-remove commands, 621
 - query-stop commands, 622
 - remove commands, 621
 - resource arbitration, 619, 621
 - Root virtual bus driver, 624–625
 - start device commands, 621
 - Start values, 623–624
 - state transitions, 622
 - stop commands, 622
 - surprise-remove commands, 622
 - volume manager, coordination with, 655
 - Windows executive, as component of, 59
- PMIE (Protected Mode Internet Explorer), 466–470
- PnP. *See* Plug and Play
- PnP (Plug and Play) manager. *See* Plug and Play manager
- PnP-X (Plug and Play Extensions), 1032–1033
- PNRP (Peer Name Resolution Protocol), 1039–1041
- pointers, object
 - Pointer count object header field, 139
 - pointer encoding, 717–718
 - retention, role in, 160
- policies
 - log management, 918
 - lsass database, 454
 - Software Restriction Policies, 533–535
- pools, memory
 - !poolused command, 726
 - !vm command, 1145
 - allocation routines for, 721
 - corruption of, 1140–1142
 - defined, 4
 - device drivers, viewing usage by, 725–726
 - Driver Verifier special pool option, 1141–1142
 - Driver Verifier tracking of, 726
 - executive allocation routines for, 60
 - Gflags, enabling special, 1141
 - hives usage of, viewing with !reg dumptool, 273
 - Kernel Memory counters, 702
 - leaks, detecting, 727–728
 - location of, 721
 - look-aside lists, 728–729
 - monitoring usage of, 724–728
 - nonpaged pools, 721
 - NUMA node dependence, 721–722
 - number of, 721–722
 - paged pools, 721
 - pool tags, 157, 725–726
 - Pool Tracking for drivers, 801
 - Poolmon tool, 724–726
 - sizes of, 722–724
 - Special Pool verification, 799–801
- pools, thread. *See* worker factories
- port drivers
 - C-LOOK algorithm, 647
 - defined, 1170
 - LANMan Redirector, 897
 - MPIO-aware, 649
 - place in disk driver architecture, 647
 - Scsiport.sys, 647
 - undocumented interfaces of, 543
 - Windows-supplied, 647
- port objects, 1170
- portability, 38–39, 68
- POSIX
 - configuring, 57
 - DLL for, 52
 - enhanced version availability, 50–51
 - exception dispatching, 117
 - name space support for, 945
 - NTFS support for, 931
 - SUA. *See* SUA (Subsystem for Unix-based Applications)
 - subsystem functions, 56–57
 - support images for process creation, 352
 - user-mode processes for, 36
 - Windows subsystem, calls to, 53
- power manager
 - !popolicy command, 642
 - ACPI compliance, 636–638
 - commands, sending of, 639–640
 - control of device power by drivers and applications, 643–644
 - defined, 1170
 - device power states, 638
 - driver power operation, 639–643
 - hardware latency characteristic of states, 636
 - option configuration, 641–642
 - PnP manager queries to devices, 639
 - policies, displaying, 642
 - power consumption characteristic of states, 636
 - query commands, 640
 - sleeping states of, 637–638
 - software resumption characteristic of states, 636

- stop codes for crashes, 1121
- system power capabilities,
 - viewing, 641
- system state definitions, table of, 637
- system to device state mappings, 639–640
- time out values, 643
- transition decisions, 638–639
- viewing device status, 640
- Windows executive, as component of, 59
- PPP support, 1057
- PPs (Protection Profiles), 453
- PRCB (processor control block)
 - IRQL values saved to, 95
 - multiprocessor thread scheduling, 434–435
 - queued spinlock pointers in, 175
 - ready queues, 404
- preemptive multitasking, 413–414
- prefetching. *See* logical prefetching
- prefixes, function name, table of, 72–73
- presentation layer, OSI, 1002
- Previous Versions, 693–695, 698
- primary partitions, 653
- principal names, RPC, 1017
- printer drivers
 - 64-bit, from 32-bit processes, 215
 - defined, 1170–1171
 - port drivers with, 542
- priority classes
 - job processes, setting for, 447
 - organization of threads by, 393
 - specifying, 350–351, 396
 - viewing, 397
- priority levels, thread
 - balance set manager use of, 427–430
 - booster function, 395
 - boosts during executive resource waits, 422–423
 - boosts to, list of cases for, 419
 - CPU starvation, boosts for, 427–430
 - determination of, 393–395
 - foreground process boosts after waits, 423–425
 - functions for getting/setting, 395
 - GUI thread wake-up boosts, 425–427
 - I/O completion boosts to, 420–423
 - MMCSS boosts, 430–434
 - multimedia playback boost, 420
 - preemption scheduling scenario, 414
 - privilege for increasing, 506
 - quantum end scenario, 415–417
 - ready queues for, 404
 - real-time priorities, 399
 - tools for viewing, 396–398
 - voluntary switch scenario, 413
 - waits for events and semaphores, boosts after, 421–422
- private address space page tables, 765–766
- private cache maps, 1171
- private committed pages, 706–707
- private memory allocations, 712, 758
- private namespaces, 164–167
- private pages, 706–707
- privilege levels of CPUs, 16
- privileged access control, 22
- privileges, account
 - APIs for checking, 503
 - assignment tools, 501
 - component enforcement of, 503
 - defined, 501
 - displaying, 501
 - enabling of, 503–504
 - security auditing, 511
 - super privileges, 509–510
 - table of, 505–508
- procedure calls. *See* local procedure calls
- process and thread manager. *See* process manager
- process blocks. *See* EPROCESS blocks
- Process Explorer
 - access rights, displaying for objects, 158
 - ASLR support status of processes, 761
 - color coding of processes, 11
 - CPU utilization, viewing, 9
 - DEP settings, viewing, 716–717
 - device drivers, listing, 546
 - Difference Highlight Duration option, 11
 - DPCs, display of time used by, 75
 - DPCs, monitoring, 110–111
 - evading single-instancing, 166
 - full paths for images, 11
 - handles, viewing open, 150–151
 - hive handles, viewing, 268
 - I/O priorities, 602–603
 - information available from, 8–9
 - integrity levels, viewing, 465–466
 - interrupts, 75
 - job objects, viewing, 447–449
 - layout of, 10
 - mapped files, viewing, 711
 - mapping system threads, 77
- page file data, 784
- page priority, viewing, 810
- priority, viewing, 397
- privileges, viewing, 503–504
- processes requesting memory
 - resource notification, 835
- section objects, viewing, 794
- security descriptors, viewing, 488
- service processes, 82, 315
- session namespaces, 168–169
- SIDs, viewing, 463–464
- System Idle Process, 418
- thread start addresses, viewing, 117–119
- threads, viewing, 381–386
- tree view, 11
- viewing process details with, 10–11
- worker factories, viewing, 388–390
- process IDs, 5, 1171
- process manager
 - boot initializations, 1090, 1093
 - CPU rate limit enforcement, 445
 - defined, 1171
 - quotas for address spaces, 756
 - thread creation, 380–381
 - Windows executive, as component of, 59
- Process Monitor
 - access-denied errors, 909
 - application settings, locating, 263
 - basic vs. advanced modes, 908
 - BUFFER OVERFLOW trace results, 264–265
 - cache file system activity, viewing, 878–883
 - device driver mechanics, 262
 - file system filter drivers, as example of, 907–908
 - file system traces, 909–910
 - friendly names for I/O, 908
 - I/O priorities, viewing, 601–602
 - idle system polling behavior, 909
 - integrity levels, PMIE, 466–470
 - logon traces, 265–266
 - prefetcher activity, viewing, 826
 - privileges required for, 908
 - registry monitoring overview, 262
 - registry troubleshooting techniques, 264–265
 - registry, viewing idle system activity, 262
 - repetitive polling of registry, viewing, 262
 - traces with, 264–265
 - troubleshooting file systems, 908–910
 - unprivileged user sessions, tracing, 265

1216 Process object type

- Process Monitor, *continued*
 - viewing process startup process, 366–370
 - viewing system processes tree, 74
- Process object type, 136
- Process: Page File Bytes counter, 780–781
- process page file quotas, 707
- process VADs, 788–789
- process working sets, 822, 1171
- process/stack swapper, 700
- processes
 - access tokens of, 5, 13
 - address space limitations on, 14–15
 - ASLR support status of, 761
 - components of, 5
 - CPU utilization, viewing, 9
 - creating. *See* CreateProcess functions
 - data structures of, 335–342
 - debug ports of, 216
 - defined, 1171
 - elevation of, 528–533
 - EPROCESS blocks, 335–342, 345
 - executable programs of, 5
 - Explorer. *See* Process Explorer functions related to, table of, 344–345
 - handles for objects. *See* handles handles of, 5, 13. *See also* handles idle, swapping out of memory, 832
 - IDs of. *See* process IDs
 - image loader initialization of, 222
 - integrity levels of, 465–466
 - jobs, relationship to, 13, 445
 - kernel global variables for, table of, 342–343
 - KPROCESS blocks, 338–340
 - object basis of, 21
 - page directories of, 764–766
 - page file quotas, 707
 - page priority, 809–810
 - parent process IDs, viewing, 5–6
 - PEB (process environment block), 335–342
 - performance counters for, table of, 343–344
 - priority of, 394–395
 - private address spaces of, 712, 736
 - Process Explorer for viewing, 8–9
 - programs, distinguished from, 5
 - protected processes, 346–348, 384–386
 - quantum reset values, 407
 - security structure for, 512
 - server processes, 1174
 - shared memory sections, 12, 709–711
 - signaled states for
 - synchronization, 179–180
 - start-of-thread function, 117–118
 - system. *See* system processes system addresses for, 752
 - Task Manager, viewing
 - information with, 6–8
 - thread scheduling indifference to, 392–393
 - threads as components of, 5
 - threads of. *See* threads
 - unkillable I/O, debugging, 590–592
 - virtual address descriptors, 13
 - virtual address spaces of, 5, 12
 - virtual size limits, 699
- Processor: % Privileged Time counter, 17–19
- Processor: % User Time counter, 17–19
- processor affinity, 1171
- processor masks, active. *See* active processor masks
- processors. *See* CPUs
- profile system performance
 - privilege, 508
- Program Compatibility Assistant (PCA), 333
- programs
 - application programs. *See* applications
 - defined, 1171
 - processes, distinguished from, 5
 - protected driver lists, 636
 - protected mode, 1077, 1171
 - protected processes
 - access masks for, table of, 347
 - defined, 1171
 - mechanics of, 346–348
 - purpose of, 346
 - threads of, viewing, 384–386
 - protecting memory, 711–713
 - Protection Profiles (PPs), 453
 - protocol device classes, 616
 - protocol drivers
 - defined, 68, 1171
 - NDIS for device independence.
 - See* NDIS (Network Driver Interface Specification)
 - purpose of, 1044
 - TDI. *See* TDI (Transport Driver Interface) transports
 - protocol stacks, 1003. *See also* stacks, network
 - prototype PTEs, 776–777, 794, 1171
 - providers
 - ETW, 207
 - network provider interfaces, 1035
 - ordering of, 1035
 - WNET, 1034–1035
- PsCreateSystemThread function, 76
- Psexec tool
 - Blue Screen tool, running, 1152
 - launching programs at Low integrity level, 480
- PsExec, viewing security descriptors with, 456
- PsGetSid, 463
- Pshed.dll, 67
- PspAllocateProcess, 354–359
- PspAllocateThread, 359–360
- PspCreateThread, 359–360, 381
- PspCreateThreadNotifyRoutine variable, 379
- PspInsertProcess, 359
- PspInsertThread, 359–360
- PspUserThreadStartup, 363–364
- Psxss startup, 1096
- PTEs (page table entries)
 - Accessed bits, 767
 - byte index fields, 768
 - byte indexes, 763–764
 - defined, 762, 1169
 - Dirty bits, 767, 775
 - fields of, 766
 - flags in, 766
 - invalid. *See* page faults
 - invalid types of, 775–776
 - limitations of, 744
 - locating in page table indexes, 764, 766
 - locating with page table indexes, 763
 - mechanics of translations, 766–767
 - PAE address translation example, 770–771
 - PAE width for, 769–771
 - page location step, 764
 - performance counter for, 744
 - PFN field, 766
 - PFNs, pointers from, 809
 - prefetched pages in, 779
 - prototype PTEs, 776–777
 - status and protection bits, table of, 767
 - structure of, 763
 - system, 737, 1176
 - system space, 744–745
 - system space addresses for, 752
 - tracking and viewing, 745
 - transition, 1178
 - translation look-aside buffers, 767–769
 - widths, 763
 - Write bits, 767
- pushlocks, 178, 192–193

Q

QoS (Quality of Service), 1043, 1062–1064, 1171

Quality Windows Audio/Video Experience (qWAVE), 1063–1064

quantums, thread

- accounting for, 407–408
- boosting, 410
- clock intervals for, 406–408
- controlling, 408–410
- defined, 392, 1171
- end processing, triggering of, 407
- initial value set by CreateProcess, 356
- job scheduling class, 446
- jobs, values for, 409
- partial quantum decay, 407
- process reset values, 407
- Programs vs. Background Services settings, 409–410
- quantum end scenario, 415–417
- quantum targets, 406–407
- registry values for, 410–412
- reset values, 407
- Short vs. Long settings, 410–412
- targets, 1171
- units, 1171
- Variable vs. Fixed settings, 410–412
- windows in foreground, effect of, 410

query name object method, 148

queued spinlocks, 175–176, 1172

queues

- defined, 1172
- dispatcher ready threads, 404
- I/O completion ports based on, 595–597
- WDFQUEUE for KMDF, 613–614
- work items, 1181

quotas

- charges, 162, 1172
- management system, 444–445
- NTFS tracking of, 962–963
- per user for disk space, 928
- system address space, 756

qWAVE (Quality Windows Audio/Video Experience), 1063–1064

R

race conditions, use after free bugs, 1140

RAID level 0, 663–664

RAID level 1, 664–666

RAID level 5, 666–667, 1172

RAM. *See* physical memory

RAM optimization software, 846–847

ramdisks, BCD options for, 1083

Raw FSD, 896

read I/O operations. *See* I/O processing

read-ahead, intelligent. *See* intelligent read-ahead

read-committed isolation, 966

ReadFile function

- file system driver operations after calling, 902–906
- mailslots with, 1024–1025
- named pipes with, 1024–1025
- synchronous I/O, 563

ReadFileEx, 113, 577

ReadProcessMemory function, 706–707

ready queues, dispatcher, 404, 435

ready summary bit masks, 1172

Ready thread execution state, 400

ReadyBoost, 844–845, 1099

ReadyBoot, 1099

ReadyDrive, 845–846

real mode, 1077, 1172

real-time priorities, 399

real-time processing, 104–106, 399

receive scaling, 1054

Receive Window Auto Tuning, 1044

recovery

- cache manager recoverable file system support, 853–854
- CLFS support for. *See* CLFS (Common Log File System)
- FAT file systems, 985–988
- NTFS bad-cluster recoveries, 985–988
- NTFS design goal for, 918–919
- NTFS support for. *See* NTFS recovery support
- recoverability defined, 1172
- TxF implementation of, 973–974

recursive faults, 1151

redirectors, 897–899, 1033–1038, 1065

redundancy, NTFS support for, 919–920

reference counts, 160, 1172

referencing objects, access checks for, 460

Reg.exe, 249–250

Regedit, 249–250, 267, 1112–1114

Regini.exe, 249–250

registries

- altitude, 280
- application settings, locating, 263
- application startups reading of, 250
- ASCII vs. Unicode storage, 281
- bins, 269–272
- boot loader reading of, 250
- cell indexes, 271

- cells. *See* cells, registry
- configuration management subsystem. *See* configuration manager
- drive letter storage in, 667–668
- editing, overview of tools for, 249–250
- enumeration of devices in, 628, 630
- filtering, 280
- hives of. *See* hives, registry
- HKEY_CLASSES_ROOT, 255
- HKEY_CURRENT_CONFIGURATIION (HKCC), 259
- HKEY_CURRENT_USER, 253
- HKEY_LOCAL_MACHINE (HKLM), 255–259
- HKEY_PERFORMANCE_DATA (HKPD), 259
- HKEY_USERS, 253–254
- initialization during boots, 1094–1095
- kernel reading of, 250
- key control blocks, 276–278, 281
- key object type, 137, 276
- keys. *See* keys, registry
- KTM support for updates to, 240
- logon process reading of, 250
- memory mapping of, 272–276
- memory usage, viewing, 274
- modification scenarios, common, 250
- monitoring overview, 262
- namespaces of, 276
- object namespace form of key names, 276
- operation flow control, 277–278
- overview of, 23
- paged pool usage, viewing with !reg dumppool, 273
- performance optimizations of, 280–281
- polling vs. notification of changes in, 250
- quantums, settings for, 410–412
- readings of, principal times for, 250
- REG_ value types, table of, 251–252
- REG_LINK type, 251–252
- Regedit.exe. *See* Regedit
- RegNotifyChangeKey function, 262
- repetitive polling, viewing, 262
- resource manager (RM), 261–262
- root keys. *See* root keys, registry
- security descriptors, 280–281
- security-descriptor cell type, 270
- Services subkey values, table of, 282–287

- registries, *continued*
 - session manager configuration information, 78
 - structure of, 252
 - subkeys, 251. *See also* keys, registry
 - System.sav, 279
 - transaction operations on. *See* TxR (Transactional Registry)
 - troubleshooting techniques, Process Monitor, 264–265
 - value types, 251–252
 - virtualization of, 526–528
 - Wow64 structure for, 213–214
 - Registry Editor, 256–257
 - Reliability and Performance Monitor. *See* Performance Monitor
 - remote access, 1066
 - remote connections, Terminal Services for, 19–20
 - remote FSDs (file system drivers), 897–899
 - Remote NDIS, 1060–1062
 - Remote Registry Service, 291
 - reparse data, 1172
 - reparse points
 - defined, 1172
 - junctions, 926
 - mount points with, 669
 - NTFS reparse point attribute, 944, 965
 - NTFS reparse point files, 941
 - symbolic links, 925–926
 - reparse tags, 1172
 - replacement policies, 827–828
 - replay protection, IPsec for, 1050
 - request packets. *See* IRPs (I/O request packets)
 - reserving pages, 706–708
 - resident attribute, 1172
 - resource accounting, 161–162
 - resource arbitration, Plug and Play manager, 619, 621, 1172
 - Resource Manager (RM)
 - group for, 462
 - objects, KTM, 241
 - TxF use of, 969–971
 - Resource Monitor, 25–26
 - resources
 - exhaustion prevention, 332
 - file objects, 555–562
 - low, simulation by Driver Verifier, 802
 - mutual exclusion rule, 170–171
 - TxF resource managers, 969–971
 - restart LSNs, 914, 973–974
 - restore files and directories
 - privilege, 510
 - restricted tokens, 483, 1172–1173
 - resuming after hibernation, 1078
 - rings, 1173
 - RM. *See* Resource Manager (RM)
 - Rom page state, 804
 - root keys, registry
 - defined, 251
 - hive root keys, relation to, 266
 - HKEY_CLASSES_ROOT, 255
 - HKEY_CURRENT_CONFIGURATIION (HKCC), 259
 - HKEY_CURRENT_USER, 253
 - HKEY_LOCAL_MACHINE (HKLM), 255–259
 - HKEY_PERFORMANCE_DATA (HKPD), 259
 - HKEY_USERS, 253–254
 - tables of, 252
 - root partitions, hypervisor, 230–232
 - Root virtual bus driver, 624–625
 - rotate VADs, 790
 - routines, kernel support. *See* kernel support functions
 - Routing and Remote Access service, 1066
 - RPC (remote procedure call)
 - advantages for programmers, 1014–1015
 - ALPC support for, 202–204
 - asynchronous version of, 1015–1016
 - availability of networking provided by, 1016
 - defined, 1014
 - IDL (Interface Definition Language), 1016
 - impersonation for, 481, 1017
 - implementation, 1017–1018
 - kernel-mode support for, 1018
 - local execution, appearance of, 1014
 - local RPC, 1017–1018
 - marshalling for remote execution, 1015
 - MIDL compiler for, 1016
 - named pipes with, 1017–1018
 - notification mechanism of, 1015–1016
 - operation mechanics, 1014–1016
 - principal names, 1017
 - procedural model of, 1014
 - purpose of, 1014
 - run-time DLL for, 1017–1018
 - security issues, 1017
 - server name publishing, 1016
 - stub procedures libraries, 1015
 - subsystem, 1018
 - transport provider interface of, 1016
 - unmarshalling procedures, 1015
 - Winsock with, 1017–1018
 - Rtl functions
 - Ntdll as container of, 58
 - RtlUserThreadStart, 364
 - Run As Administrator command, 531
 - run as functionality, 528–533
 - run once initialization, 197–198
 - run time library. *See* Rtl functions
 - Running thread execution state, 400
- ## S
- SACLs (system access control lists)
 - ACE types in, 488
 - assignment to new objects, 491
 - defined, 1176
 - security descriptors, in, 485–487
 - system audit ACEs, 1176
 - safe mode
 - boot logging in, 1104–1105
 - defined, 1173
 - Directory Services Restore mode, 1102
 - driver loading in, 1102–1103
 - F8 command for, 1101
 - options of, 1101–1102
 - purpose of, 1101
 - services loaded in, 1104
 - user program awareness of, 1104
 - Userinit in, 1104
 - safe structured exception handling, 717–718
 - SAM (Security Accounts Manager)
 - service
 - APIs, 1067
 - database for, 1173
 - defined, 454, 1173
 - viewing descriptors for, 456
 - SANs (storage area networks)
 - iSCSI drivers, 648
 - policy for volumes, 670–671
 - SAS (secure attention sequence)
 - beginning user logon, 516–517
 - B-level security provided by, 453
 - defined, 1173
 - logon process role of, 79
 - mechanics of, 516
 - notifications to Winlogon, 515
 - scalability
 - architectural support for, 43
 - dispatcher databases for, 404
 - worker factory, 387
 - scatter/gather I/O, 566, 1173
 - ScAutoStartServices, 303–304
 - SCBs (stream control blocks), 936
 - ScGenerateServiceDB, 300–301
 - SChannel (Secure Channel), 1017
 - scheduled file I/O, 603–604
 - scheduling threads. *See* thread scheduling

- SCM (service control manager)
 - application registration by, 282–283
 - auto started services, 303–307
 - auto starts completed event, 307
 - binding dependencies, specifying for, 1064–1065
 - boot steps for, 1097
 - command processing for
 - applications, 287–288
 - database generation, 300–301
 - delayed auto starts, 306–307
 - dependencies, 304
 - device driver loading, 302, 306
 - failed service starts, 306
 - failures of services, 310–311
 - groups, organization of, 300–301
 - ImagePath values, 305
 - last known good boots, 308–309
 - logging on services, 305
 - looping through services stage, 306
 - LSASS calls, 301–302
 - network drive notifications, 303
 - preshtutdown notifications, 312–313
 - privilege creation by, 291–292
 - recovery options, 310–311
 - RPC pipes of, 302
 - SCP dialog function, 300
 - SCP interaction with, 317
 - service processes, launching, 306
 - service startup process, 287, 303–307
 - service tags, 316
 - services database, 301
 - shared services processes, 313–316
 - shutdown mechanism, 311–313
 - shutdown preparations, 302
 - startup errors, 307–308
 - startup of, 300–302
 - SvcCtrlMain, 300–301
 - synchronization event of, 300
 - system architecture role, 80–82
 - time out value, 311
 - SCPs (service control programs)
 - defined, 317
 - policy layering, 317
 - SCM dialog function for, 300
 - Services MMC snap-in as example, 317
 - screen crashes during boots, 1112–1114
 - SCSI, 647–648
 - SDK. *See* Windows Software Development Kit (SDK)
 - Search service change journals, 956
 - secondary resource manager, TxR, 969–971
 - section objects
 - access rights, 710
 - ACLs for, 713
 - Based attribute, 793
 - committed virtual pages, 706–708
 - control areas, 794, 796–798
 - copy-on-write page protection, 718–719
 - creation of, 351–352, 710
 - data structures for mapped sections, 794
 - defined, 1173
 - execution prevention. *See* DEP (data execution prevention)
 - executive use of, 792
 - mapped file I/O, 565–566, 792
 - mapped files, 710–711, 792
 - Maximum size attribute, 793
 - object manager allocation control of, 793
 - page file backed sections, 1169
 - Page protection attribute, 793
 - Paging file/Mapped file attribute, 793
 - pointer structure, 794
 - pointers, 1173
 - prototype PTEs, 776–777, 794
 - Section object type, 136
 - segment structures, 794
 - sharing memory with, 709–711
 - structure of, 793
 - subsection structures, 794
 - subsections, 1175
 - viewing, 794
 - viewing mapped files, 711
 - views of partial sections, 710, 1179
 - sectors
 - bad-cluster recoveries, 985–988
 - boot sectors, use of, 896
 - clusters, relationship to, 937–938
 - defined, 645, 889, 1173
 - secure attention sequence (SAS), 79, 453
 - Secure Channel (SChannel), 1017
 - secure logon facilities, 1173
 - Secure Sockets Layer. *See* SSL (Secure Sockets Layer)
 - security
 - access. *See* access control
 - auditing. *See* auditing, security
 - authentication for. *See* authentication
 - AuthZ API, 500–501
 - B-level security, 453
 - C2 security ratings, 451–452
 - Common Criteria, 453
 - components, system, 454–457
 - contexts. *See* security contexts
 - descriptors. *See* security descriptors
 - directory services. *See* Active Directory
 - heap management, 733–734
 - HKLM\SECURITY subkey, 258
 - impersonating threads. *See* impersonation
 - integrity mechanism. *See* integrity levels
 - IPSec (Internet Protocol Security), 1050–1052, 1054
 - KSecDD, 455
 - locking paging files, 721
 - Lsass. *See* Lsass (Local Security Authority subsystem)
 - LSASS. *See* LSASS (local security authentication server process)
 - object access checks. *See* access checks
 - object method, 148–149
 - object protection. *See* object security
 - object reuse protection, 452
 - overview of, 22–23
 - protected process issues. *See* protected processes
 - rating standards for, 451–453
 - reference monitor. *See* SRM (security reference monitor)
 - routine, default object method for, 147
 - RPC issues, 1017
 - security-descriptor cell type, 270
 - STs (Security Targets), 453
 - Trusted Computer System Evaluation Criteria, 451–453
 - trusted facility management, 453
 - trusted path functionality, 453
 - Windows subsystem, 22–23
- security contexts
 - access tokens of processes, 5
 - defined, 473
 - threads, of, 12
 - threads, when differing from their processes. *See* impersonation
 - tokens for identifying. *See* tokens
- security descriptors
 - access determination with, 492–501
 - ACEs with, 485–488
 - attributes, list of, 485
 - DACL assignments to objects, 490–491
 - defined, 484–485, 1174
 - flags, table of, 485–486
 - NTFS attribute for, 943, 963–965
 - NTFS security file database of, 940
 - NTFS use of, 919
 - null DACL access rights, 487
 - object header fields, 139, 489
 - object manager with, 488

1220 security quality of service

- security descriptors, *continued*
 - privilege for accessing, 507
 - viewing, 488–490
- security quality of service (SQOS)
 - levels, 482
- security reference monitor. *See* SRM (security reference monitor)
- segment structure prototype PTEs, 776–777
- self-healing file systems, 899
- semaphores
 - critical sections, inside, 194–195
 - defined, 1174
 - priority level boosts after waits for, 421–422
 - Semaphore object type, 137
- Server service, 897
- server versions of Windows
 - architecture compared to client versions, 43–47
 - list of, 43–44
 - optimization compared to clients, 46
 - physical memory limits for, 818–819
- servers
 - communications ports, ALPC, 205
 - connection ports, ALPC, 205
 - name publishing, 1016
 - remote file system drivers for, 897–899
 - Server Message Block (SMB) protocol, 898, 1025
 - server processes, defined, 1174
 - Web. *See* Web servers
- service applications
 - accounts for, 288
 - alternative accounts, configuring, 291
 - auto starting of, 303–307
 - control handlers for, 287
 - CreateService function, 282–283
 - defined, 282
 - dependencies, 304
 - error control registry values, 284
 - installation and registration of, 282–283
 - isolation issues, 294–300
 - least privilege, running with, 291–294
 - local service account with, 291
 - local system account with, 288–290
 - multiple services per process, 286–287
 - network service account with, 290–291
 - registry key creation, 282–283
 - SCM command processing, 287–288
 - SCMs, relation to, 282
 - SCPs, control by, 282
 - security contexts for, 288
 - security descriptors, 286
 - Session 0 Isolation, 297–300
 - settings, table of, 283–287
 - SID registry values, 286
 - SIDs for, 295–300
 - Start registry values, 284
 - StartService function, 282
 - StartServiceCtrlDispatcher, 287
 - startup process, 303–307
 - status messages of, 287
 - Type registry values, 284, 286–287
 - user accounts, running under, 297
 - user interaction, configuring for, 298–299
 - Windows Services MMC snap-in, 291
- service control manager. *See* SCM (service control manager)
- service descriptor tables, 130–133
- service packs, pending file moves for, 1096
- service processes, 36
- services
 - applications. *See* service applications
 - components implemented as, 81
 - components of, 282
 - CurrentControlSet subkey, 308–309
 - defined, 4, 281, 1180
 - dialog box display prohibition, 297
 - failures and recoveries, 310–311
 - generic object manager, 142–143
 - groupings, table of, 314
 - ImagePath values, 305
 - Interactive Services Detection, 299–300
 - isolation issues, 294–300
 - listing, 81–82
 - mapping, 81
 - names of, 81
 - native system. *See* native system services
 - privileges, viewing, 293–294
 - Process Explorer, viewing in, 82
 - recoveries, 310–311
 - Sc.exe tool, 293–294
 - SCM. *See* SCM (service control manager)
 - SCPs. *See* SCPs (service control programs)
 - server processes, 1174
 - service control manager (SCM), 80–82
 - Service Host (SvcHost), 313
 - Services registry key, 308–309
 - Services.exe, 81
 - shared processes, 313–316
 - shutdown mechanics, 1115–1118
 - shutdown mechanism, 311–313
 - SIDs for, 295–300
 - startup errors, 307–308
 - startup process, 303–307
 - types of, 4
 - user mode processes, 36
 - viewing services running inside processes, 315–316
- Services MMC snap-in, 291, 310–311
- Services.exe. *See* SCM (service control manager)
- Session 0 Isolation, 297–300
- session layer, OSI, 1002
- Session Manager. *See* Ssmss.exe (Session Manager)
- session namespaces, 167–169
- session spaces
 - defined, 736, 1174
 - global space addresses, 752
 - memory utilization, viewing, 743
 - system addresses for, 752
 - x86 layout of, 740–743
- session working sets, 823
- sessions
 - defined, 1174
 - multiple, 19–20
 - number of, 20
 - process membership in, displaying, 742
 - version differences for, 20
 - viewing MM_SESSION_SPACE, 742
- Setup, Windows
 - BCD preparation, 1077
 - boot sector creation by, 1076–1077
 - boot sector format dependence, 1077
 - EFI firmware setup, 1086
 - system partition formatting, 1076–1077
- Sfc.exe (System File Checker), 1111–1112
- shadow copies. *See* VSS (Volume Shadow Copy Service)
- shared assemblies, 364–365
- shared cache maps, 862–867, 1174
- shared memory sections. *See also* section objects
 - defined, 709, 1174
 - prototype PTEs, 776–777
 - thread vs. process addressable space, 12
- shatter attacks, 494
- shims, WFP, 1047
- shortcuts, shell, 929–930
- shrinking engine, 1174

- shrinking partitions, 932–933
- shutdowns
 - mechanics of, 1115–1118
 - privileges to, 507–508
 - services shutdown mechanism, 311–313
 - system-shutdown notification routines, device driver, 550
- side-by-side assemblies. *See* SxS (Side-by-Side Assembly)/Fusion
- SideShow, UMDF with, 616
- SIDs (Security IDs)
 - access checks, for, 461–462
 - assignment of, 461–462
 - built-in, 462
 - defined, 461, 1174
 - enabled vs. disabled, 496
 - format of, 461
 - group, in tokens, 475–476
 - integrity levels with, 464–465
 - Owner Rights SID, 495–496
 - relative identifiers (RIDs), 461
 - restricted tokens, in, 483
 - security descriptors, in, 485–487
 - services, for, 295–300
 - user logons, generation for, 516–517
 - user, file tagging with, 928
 - viewing, 463–464
 - well-known
 - Winlogon session SIDs, 462
- Sigcheck utility, 533
- signaled state, 179–180, 1174
- signed drivers, 634–636
- simple volumes, 645, 1174
- single sign-ons, 515, 1175
- single-instancing with named objects, 166
- singly linked lists, 176–177, 749–751
- slim reader writer locks. *See* SRW (slim reader writer) locks
- SIPolicy tool, 46–47
- small memory dumps, 1127–1128, 1175
- smartcard credential providers, 79
- SMB (Server Message Block) protocol
 - CIFS version of, 898
 - named pipe and mailslot reliance on, 1025
 - WNet provider for, 1034–1035
- SMP (symmetric multiprocessing)
 - ASMP compared to, 39
 - defined, 39, 1176
 - identical processor check during boot, 1093
 - mutual exclusion rule, 171
 - scalability support, 43
 - threading synchronization. *See* multiprocessor thread scheduling
- Sms.exe (Session Manager)
 - concurrency level, setting, 1094
 - configuration manager call during boot, 1094
 - crashes on Csrss and Winlogon termination, 79
 - critical process setting, 1094
 - Csrss, starting, 1095–1096
 - defined, 1075
 - known DLLs, opening, 1095
 - MS-DOS device symbolic link creation, 1094
 - page files, reads during boots, 781
 - processes launched by, 78
 - protected prefix creation by, 1094
 - purpose of, 78–79
 - registry information for, 78
 - Smpinit duties, 1094
 - special characteristics of, 1094
 - start of process during boots, 1093–1096
 - subsystems, initializing lists to run during boots, 1095
 - Terminal Services sessions, 79
 - user session creation, 78–79
 - Wininit/Winlogon launches, 1096
- snapshots. *See* VSS (Volume Shadow Copy Service)
- sockets. *See* Winsock
- soft links. *See* symbolic links
- software DEP (data execution prevention), 717–718
- Software Development Kit. *See* Windows Software Development Kit (SDK)
- software interrupts. *See* interrupt dispatching
- Software Restriction Policies, 533–535
- spanned volumes, 662–663, 1175
- sparse data compression, 952–954
- sparse files, 927, 956, 1175
- spinlocks
 - availability to executive, 174–175
 - defined, 173, 1175
 - device drivers requiring, 174–175
 - disadvantages of, 177
 - ExInterlocked functions, 176–177
 - global queued, 175
 - hardware support for, 173
 - instack queued spinlocks, 176
 - IRQLs of, 174
 - Ke functions with, 174–175
 - KeAcquireStackQueuedSpinLock, 176
 - lock operation, 173
 - mechanism of, 173–175
 - pause assembly instructions, 174
 - queued, 175–176
- splash-screen crashes or hangs, troubleshooting, 1112–1114
- SQL Server shadow copies, enabling, 688–689
- SQOS (security quality of service) levels, 482, 1174
- squatting attacks, 164
- SRM (security reference monitor)
 - access checks, 459–461
 - auditing role, 511
 - boot initializations, 1090
 - calls to, 157
 - defined, 1174
 - discretionary access checks, 497
 - impersonation with. *See* impersonation
 - inputs to model, 461
 - integrity levels with, 464
 - mechanics of, 457
 - privilege to act as part of the operating system, 508, 510
 - responsibilities of, 454
 - SeAccessCheck function, 460
 - SID inputs to, 461–464
 - tokens with. *See* tokens
 - Windows executive, as component of, 59
- SRW (slim reader writer) locks, 196
- SSL (Secure Sockets Layer)
 - HTTP Server API support for, 1019–1021
 - IPSec certificate support, 1051
 - SSP implementation of, 1017
 - WinHTTP support for, 1019
- SSPIs (Security Support Provider Interfaces)
 - built-in, list of, 1017
 - impersonation for, 481
 - RPC with, 1017
- SSTP (Secure Socket Transmission Protocol), 1066
- stack cookies, 717–718
- stack crashing, 124–125, 1145–1147
- stacks, memory
 - address ranges for, 758
 - DPC stacks, 787
 - frames of, 1175
 - kernel stacks, 786–787
 - process/stack swapper, 700
 - randomization by ASLR, 760
 - threads, reservations for, 707
 - two stacks per thread model, 784
 - user stacks, 785
 - viewing for threads, 382–383
- stacks, network
 - layers of software types supporting, 1001
 - Next Generation TCP/IP Stack, 1012–1013, 1044–1045
 - OSI Reference Model, 1001–1003
 - protocol stacks, 1003
- stacks, storage. *See* storage stacks
- Standby page state, 804, 809–812

1222 Standby thread execution state

- Standby thread execution state, 400
- standby, SuperFetch service
 - scenario for, 840
- start I/O routines, 548
- start-of-thread function, 117–118
- Startup And Recovery settings, 1125
- startups
 - automatic program executions
 - during, 1100
 - boots. *See* boot process
 - last known good. *See* last known good control sets
 - startup repair tool, 332
 - Startup Repair tool, 1106–1108
 - troubleshooting with WinRE, 1106–1108
- stop codes
 - access violation-based, 1122
 - blue screens, displayed in,
 - 1120–1121
 - consistency check based,
 - 1122–1123
 - critical object termination, 1123
 - defined, 1175
 - DEP-based, 713–714
 - display-based, 1122
 - exception-based, 1122, 1145
 - file system, 1123
 - hardware error based, 1123
 - IRQL issues, 1121
 - KeBugCheckEx function, as inputs to, 1120–1121
 - kernel pool manager generated, 1122
 - memory management based,
 - 1122
 - Notmyfault, shown by, 1137
 - page fault, 1121
 - parameter information with,
 - 1120–1121
 - power management, 1121
 - reference information for, 1121
 - trap-based, 1122
 - USB error based, 1123
- storage
 - basic disk based. *See* basic disks
 - BitLocker for security. *See* BitLocker
 - class drivers. *See* storage class drivers
 - defragmentation, 931–932
 - disks for. *See* disks
 - drivers for. *See* disk storage drivers
 - dynamic. *See* dynamic disks
 - importance of, 645
 - latency, striped volumes for
 - reducing, 663–664
 - per user volume quotas, 928
 - port drivers, 647
 - self-healing, 989
 - stacks. *See* storage stacks
 - symbolic links for disk objects,
 - 650–651
 - terminology, 645
 - Winload support for, 646
- storage class drivers
 - Disk driver, 647
 - disk object representation,
 - 650–651
 - place in disk driver architecture, 647
- storage stacks
 - driver architecture of, 646
 - MPIO, 649
 - system storage driver strategy,
 - 600–601
- Storport.sys, 647
- streams
 - caching based on, 852
 - defined, 1175
 - multiple data streams feature of NTFS, 920–922
 - SCBs (stream control blocks), 936
- string parameters, 23–24
- Strings utility, viewing prefetch files with, 825
- striped volumes, 663–664, 674, 1175
- structured exception handling, 114, 1175
- STs (Security Targets), 453
- SUA (Subsystem for Unix-based Applications)
 - advantages over POSIX, 56
 - as enhancement to POSIX, 50–51
 - boot startup of, 1096
 - configuring, 57
 - defined, 1175
 - launching, 57
- subheaders, object
 - conditions required for presence of, 140
 - defined, 139
 - Subheader offsets field of headers, 139
 - table of, 139–140
- subkeys, registry, 251
- subsections, 1175
- subsystem DLLs, 1175
- Subsystem for Unix-based Applications. *See* SUA (Subsystem for Unix-based Applications)
- subsystems
 - binding of executable images to, 51
 - DLLs, applications calls to, 53
 - DLLs, purpose of, 36–37, 52
 - environment, role of, 51
 - I/O subsystem API, 1162
 - initializing lists to run during boots, 1095
- Ntdll.dll system support library, 57–58
- POSIX, 56–57
- RPC, 1018
- specifying binding in Visual C++, 51
- Unix-based applications, for. *See* SUA (Subsystem for Unix-based Applications)
- viewing type for images, 52
- Windows. *See* Windows subsystem
- super privileges, 509–510
- super users, 509
- SuperFetch service
 - advantages of, 842
 - agents, 837
 - application launch agent, 842
 - architecture of, 837
 - defined, 1175–1176
 - FileInfo driver with, 839
 - logical prefetches, role in, 823–827
 - memory manager, use of, 837
 - overview of, 836
 - page access tracking, 839
 - page prioritization, 840–841
 - ReadyDrive with, 846
 - rebalancer, 837, 841–842
 - robustness component, 843–844
 - Scenario manager, 837, 840
 - Trace Collector and Processor, 836–837
 - Tracer mechanism, 836, 838–839
- support images, 352
- SvcCtrlMain, 300–301
- swapper, working set, 832
- SxS (Side-by-Side Assembly)/Fusion
 - allocation by image loader, 222
 - mechanism of, 364
 - setup during process creation, 361
 - shared assembly capability, 365
 - symbol files for kernel debugging, 26–27
- symbolic links
 - creating, 924–926
 - default evaluation policy, 925
 - defined, 1176
 - device objects, of, 551
 - disk objects, 650–651
 - explicit I/O operations, use in, 903
 - file object, 556, 561–562
 - keys for, registry, 268
 - mechanics of, 924–925
 - MS-DOS device, creation by Smss, 1094
 - network provider creation of, 1036

- NTFS feature for, 924–926
- object manager implementation of, 167
- privilege for creating, 506, 925
- reparse point basis of, 925–926
- symmetric encryption algorithms, 1176
- symmetric multiprocessing. *See* SMP (symmetric multiprocessing)
- synchronization
 - condition variables, 195–196
 - critical code sections for, 194–195
 - deadlocks, 194
 - defined, 146, 1176
 - dispatcher headers, 182–185
 - DPC/dispatch level, dependence on, 171–172
 - events, condition variables to replace, 195
 - executive resources, 190–191
 - executive, visibility to programmers, 178–179
 - ExInterlocked functions, 176–177
 - gates, 189
 - guarded mutexes, 189–190
 - heaps, of, 732
 - high-IRQL. *See* high-IRQL synchronization
 - Interlocked functions, 172–173
 - IRQL checking with Driver Verifier, 802
 - kernel dispatcher objects. *See* dispatcher objects
 - kernel mechanisms, table of, 178
 - keyed events, 186–187
 - KiWaitTest function, 179
 - KTHREAD data for. *See* KTHREAD blocks
 - mutexes. *See* mutexes
 - mutual exclusion, 170–171
 - nonsignaled state, 179
 - object support for, 146–147
 - overview of, 170–172
 - pushlocks, 192–193
 - queued spinlocks, 175–176
 - run once initialization, 197–198
 - signaled state, 179–180, 1174
 - spinlocks. *See* spinlocks
 - user-mode code primitives for, list of, 178
 - user-visible objects, 178–179
 - wait blocks, 183–186
- synchronous I/O
 - cancellations, 588
 - defined, 563, 1176
 - requests to single-layered drivers, 572–578
- Sysinternals tools, 32
- system address spaces
 - cache pages in, 869
 - code, 736
 - components of, 736–737
 - crash dump information space, 737
 - defined, 736
 - dynamic management of, 751–755
 - HAL space, 737
 - hyperspace, 737
 - initialization functions, 751
 - kernel virtual allocator mechanism, 751–752
 - limiting range size by type, 754–755
 - mapped views, 736
 - MemLimit utility, 755
 - nonpaged pools, 737
 - paged pools, 737
 - page-related structure allocation, 753
 - PTEs, 737, 744–745
 - quotas, 756
 - reclaiming memory, 752–754
 - requests and releases, dynamic, 751
 - types of addresses, determining, 753
 - types of addresses, table of, 752
 - types of addresses, viewing usage, 754
 - working set list, 737
 - x86 system address space layouts, 740
- system architecture. *See* architecture
- system audit ACEs, 488, 1176
- system cache. *See also* cache manager
 - defined, 737, 1176
 - prefetched pages, 779–780
 - system space addresses for, 752
 - working set, 737
- System Calls/Sec counter, 133
- system code write protection, 1143–1144, 1176
- System File Checker, 1111–1112
- system files, 41, 1110–1111
- system functions
 - accessing internal, 69
 - DPCs for execution of, 107
 - KeBugCheckEx function, 87
 - services, 58
 - Windows. *See* Windows subsystem
- system global flags, 200–202
- system global location namespace
 - virtualization, 521–528
- system hive corruption, 1112
- system idle process, 75, 418
- System Information viewer, 900–901
- system memory pools. *See* pools, memory
- System Page File counter, 702
- system partitions
 - boot process role of, 1076
 - defined, 1176
 - formatting by Windows Setup, 1076–1077
- System process
 - mapping system threads, 76–77
 - purpose of, 75
 - threads. *See* system threads
- system processes
 - DPCs (deferred procedure calls), 75
 - idle process, 75
 - interrupts, 75
 - list of common, 74
 - LSASS. *See* LSASS (local security authentication server process)
 - service control manager, 80–82
 - session manager. *See* Ssmss.exe (Session Manager)
 - System process, 75–77
 - System process threads. *See* system threads
 - tree of, 74
 - Userinit, 80
 - Winlogon. *See* Winlogon
- system PTEs (page table entries), 737, 1176
- System Restore
 - purpose of, 1106
 - rolling back option, 1114
 - shadow copies for, 694
- system service dispatching
 - 32-bit, 125–127
 - 64-bit, 127
 - defined, 125
 - dispatch tables, 127, 129, 1176
 - drivers with, 128–129
 - epc for IA64, 127
 - handler registration, 126
 - iretd instruction, 125
 - kernel mode, 127–129
 - KeServiceDescriptorTable, 130–133
 - KiSystemService, 126, 128–129
 - mapping call numbers to functions, 130–133
 - monitoring activity of, 133
 - previous mode values, 128
 - processor detection, 127
 - service descriptor tables, 130–133
 - syscall for 32-bit AMD processors, 126
 - syscall for x64, 127
 - sysenter instruction, 125, 127
 - sysexit instruction, 125
 - trampolines, 129
 - Zw calls, 128–129
- system services, 58, 1176–1177

system support processes, 36, 1177

system threads

- balance set manager, 76
- components calling, 76
- defined, 1177
- device driver creation of, 76
- mapping to device drivers, 76–77
- mechanics of, 75–76
- memory for, 75
- PspCreateSystemThread function, 76
- tracing to calling routines, 76–77

system timers, 109

system variables

- physical memory, for, 818
- system working set, 833

system volumes, 660, 1076

system worker threads

- cache manager use of, 886–887
- defined, 198, 1177
- device drivers with, 198–199
- dynamic worker threads, 199–200
- lazy writes with, 877
- listing, 200
- mechanics of, 198–199
- number created, 199
- requesting, 198
- types of, 199

system working sets. *See also* working sets

- components included in, 856
- defined, 823, 1177
- examining, 833
- list data structures of, 737
- size computations, 833
- system cache working set misnomer, 737
- types of pages in, 832
- viewing, 856–857

system-shutdown notification routines, 550

systemwide cookies, 363

systemwide data structures, protection of, 711

T

take ownership privilege, 510

Task Manager

- Applications tab, 7–8
- kernel mode, monitoring, 18
- Memory Usage bar, 783
- page file usage, viewing, 783
- physical memory availability, 821
- process information, viewing with, 6–8
- process priority, viewing, 397
- System Idle Process, 418
- virtualization status, changing, 525

- virtualization status, viewing, 522

TCBs (thread control blocks). *See* KTHREAD blocks

TCP/IP

- binding to, 1065
- Compound TCP (CTCP), 1044
- ECN (Explicit Congestion Notification), 1044
- filtering. *See* WFP (Windows Filtering Platform)
- high loss throughput improvements for, 1045
- logging activity with kernel logger, 208–210
- name resolution protocols for, 1039
- NDIS offloading of, 1054
- NetBIOS over, 1029
- Next Generation TCP/IP Stack, 1012–1013, 1044–1045, 1053–1054
- preferred Windows protocol, reasons for, 1044
- QoS implementation, 1062–1063
- Receive Window Auto Tuning, 1044
- RPC transport interface for, 1016
- UMDF with, 616
- viewing device objects of, 1045
- Winsock Helper library for, 1011

TCSEC (Trusted Computer System Evaluation Criteria), 451–453

TDI (Transport Driver Interface) clients

- defined, 1003
- replacement API for. *See* WSK (Winsock Kernel)

TDI (Transport Driver Interface) transports

- defined, 1003–1004
- NDIS functions for, 1004
- TDI interface availability in WSK, 1012

TDX, 1045

TEBs (thread environment blocks)

- !teb command, 378
- allocation by PspAllocateThread, 360
- CreateThread setup of, 381
- granularity of memory allocations, 708
- KTHREAD pointer to, 373
- storage of, 377
- viewing, 378

temporary objects, 159–161

Terminal Server registry hives

- memory issues, 268

Terminal Services

- API for, 20
- interactive window stations, 167

- purpose of, 19–20

session manager session creation for, 79

- session namespaces, 167–168
- version differences for, 20

Terminated thread execution state, 401

termination of threads, 417

TestLimit tool

- handle creation limits, 152–153
- memory leak data, 740
- reserving address space with, 746

thread agnostic I/O

- file object attribute data for, 556
- I/O completion ports for, 587, 595. *See also* I/O completion ports
- locking user buffers mechanism, 587
- mechanisms for, 587
- user-initiated cancellation, 588

thread IDs, 12, 759

Thread object type, 136

thread pools. *See* worker factories

thread scheduling

- !ready command, 391
- API functions for, 395–396
- balance set manager, 427–430
- base priority determination, 393–395
- code locations for, 392
- context switching, 392
- CPU rate limits, 444–445
- current priority determination, 394
- dispatcher databases, 404–405
- dispatchers, 392
- execution states for, 400–404
- increase scheduling priority privilege, 399
- IRQLs with, 399, 405
- KPRCB thread-scheduling fields, 405
- KTHREAD field for data, 373
- multiprocessor. *See* multiprocessor thread scheduling
- overview of, 391
- preemption scheduling scenario, 414
- priority levels, 393–395
- priority-driven nature of, 391
- process indifference of, 392–393
- quantum end scenario, 415–417
- quantums. *See* quantums, thread real-time priorities, 399
- SetThreadIdealProcessor function, 395
- sleep functions, 396
- SwitchToThread function, 396
- termination of threads, 417

- tools for viewing priorities, 396–398
- viewing ready threads, 391
- voluntary switch scenario, 413
- wait functions, 396
- Thread: % Privileged Time counter, 17–19
- Thread: % User Time counter, 17–19
- threaded DPCs, 110
- threads
 - access rights denied for protected processes, table of, 384–386
 - access tokens of, 13
 - activation stacks, 381
 - alertable wait states, 1153–1154
 - APC control of, 112
 - completion ports for. *See* I/O completion ports
 - components of, 12
 - context switching, 418
 - contexts of, 12, 1177
 - CPU register contents, 12
 - CreateProcess creation of, 359–360
 - creation steps, 380–381
 - creation, DLLs supporting, 54
 - critical worker threads, 199
 - data structures of, 370–378
 - Deferred Ready state, 1157
 - defined, 1177
 - delayed worker threads, 199
 - dispatcher databases, 404–405
 - dispatching, 392. *See also* thread scheduling
 - dynamic worker threads, 199–200
 - environment blocks. *See* TEBs (thread environment blocks)
 - ETHREAD blocks, 360, 370–378
 - examining activity of, 381–386
 - execution states for, 400–404
 - fibers compared to, 12
 - functions for creating and manipulating, table of, 380
 - hypercritical worker threads, 199
 - I/O processing cancellations, 589–592
 - idle threads, 418–419
 - IDs of. *See* thread IDs
 - impersonation by. *See* impersonation
 - KelInitThread, 360
 - kernel variables for, 379
 - killing in Process Explorer, 396–398
 - KiThreadStartup, 363
 - KTHREAD data. *See* KTHREAD blocks
 - message states, viewing in Task Manager, 7–8
 - modes, switching between, 17
 - multiprocessor support for multiple, 43
 - multiprocessor, scheduling. *See* multiprocessor thread scheduling
 - mutual exclusion principle, 170–171
 - NtCreateThreadEx, 381
 - number per client requests, importance of, 592–593
 - page priority, 809–810
 - performance counters for, 379–380
 - pools of. *See* worker factories
 - priority levels, 393–395
 - processes, as components of, 5
 - processes, use of resources of, 12
 - protected process threads, 384–386
 - PspAllocateThread, 359–360
 - PspCreateThread, 359–360, 381
 - PspInsertThread, 359–360
 - quantums. *See* quantums, thread reserving memory for user-mode stacks, 707
 - scheduling. *See* thread scheduling
 - security contexts pf, 12
 - security structure for, 512
 - seed initialization during process creation, 356
 - shared memory sections, 12
 - signaled state, 179–180
 - spinlock acquisition, 174
 - stack crashing, 124–125
 - stack creation, 360
 - stack randomization, 760
 - stacks of, 12
 - stacks, viewing, 382–383
 - start addresses, viewing, 117–119, 382–383
 - storage area of, 12
 - system. *See* system threads
 - system worker. *See* system worker threads
 - TEBs. *See* TEBs (thread environment blocks)
 - termination of, 417
 - timers, 1177
 - TLS (thread-local storage), 12
 - TpWorkerFactory object type, 137
 - trap frames, 86–87
 - viewing information for, 376–377
 - wait queue reordering by APCs, 114
 - waiting, swapping memory of, 832
 - Windows subsystem structures for, 371
- thunk context setup, 364
- time
 - change privileges, 508
 - system, 109
- Timer object type, 137
- timers, thread, 1177
- TLBs (translation look-aside buffers), 767–769, 1178
- Tlist.exe tool
 - process tree, viewing with, 5–6
 - thread information, 377
- TLS (thread-local storage), 12
- Tm object types, 137
- tokens
 - !token debugger command, 478
 - create a token object privilege, 506, 510
 - DACLs of, 476
 - defined, 473, 1153
 - discretionary access checks using, 496–497
 - dt ntl!_TOKEN debugger command, 477
 - expiration time field, 477
 - fields of, 476–477
 - filtered admin tokens, 474, 483–484
 - flags of, 476
 - group SIDs in, 475–476
 - information contained in, 474
 - inheritance of, 474
 - LogonUser function creation of, 474
 - LSASS generation of, 80
 - LUIDs of, 476–477
 - mandatory integrity policies, 475–476
 - mandatory policies of, 476
 - primary compared to impersonation, 476. *See also* impersonation
 - privilege arrays, 476, 483
 - processes, of, 5, 13
 - restricted, 483
 - SIDs, restrictions with, 483
 - threads, of, 13
 - Token object type, 137
 - user logons, created for, 473, 518
 - viewing, 477–479
- tools used in book, list of, 24–25
- topology discovery. *See* LLTD (Link-Layer Topology Discovery)
- torn writes, 915
- TPM (Trusted Platform Module)
 - defined, 677–678
 - Services, 687
 - TBS, 681–683
- TpWorkerFactory object type, 137
- tracing
 - consumers, ETW, 207
 - controllers, ETW, 207
 - ETW (Event Tracing for Windows), 207–210

- tracing, *continued*
 - kernel events, 207–210
 - kernel trace classes for, 207–208
 - NtTraceControl system call, 207
 - providers, ETW, 207
 - Transactional Registry. *See* TxR (Transactional Registry)
 - transactions
 - atomic transactions, 918–919
 - boots, manager initializations during, 1091–1093
 - CLFS support for. *See* CLFS (Common Log File System)
 - defined, 1177
 - Fsutil.exe transaction queries, 968
 - isolation of, 966–968
 - KTM for. *See* KTM (Kernel Transaction Manager)
 - Ktmutil.exe tool for viewing, 241
 - listing current, 968
 - named pipes, for, 1022
 - nontransacted writers and readers, 966–967
 - rollback operations, 918–919, 967
 - tables, transaction, 1177
 - TM (Transaction Manager) objects, 241
 - Tm object types, 137
 - Transact-demo.exe tool, 967–968
 - transacted writers and readers, 966
 - TxF. *See* TxF (Transactional NTFS)
 - TxR. *See* TxR (Transactional Registry)
 - transfer jobs, 1030
 - Transition page state, 804
 - transition PTE, 1178
 - Transition thread execution state, 401
 - translating virtual to physical addresses. *See* address translation
 - translation look-aside buffers, 767–769
 - TransmitFile function, 1008–1009
 - TransmitPackets function, 1009
 - Transport Driver Interface clients. *See* TDI (Transport Driver Interface) clients
 - transport drivers, NDIS intermediate drivers, 1057
 - transport layer, OSI, 1002, 1004–1005
 - transport protocol drivers. *See* protocol drivers
 - transport provider interface, RPC, 1016
 - transport service providers
 - adding to Winsock, 1009
 - Mswsock.dll, 1011
 - NMR with WSK for, 1012
 - TDI. *See* TDI (Transport Driver Interface) transports
 - transports. *See* TDI (Transport Driver Interface) transports
 - trap dispatching
 - activators of, 85–86
 - exceptions. *See* exception dispatching
 - handlers. *See* trap handlers
 - interrupts. *See* interrupt dispatching
 - KeBugCheckEx function, 87
 - machine state capture, 86–87
 - page fault dispatching, 774
 - system service traps. *See* system service dispatching
 - trap frames, 86–87, 1178
 - trap handlers
 - control transfer mechanism, 87
 - defined, 85, 1178
 - interrupt handler installation, 87
 - KeBugCheckEx function, 87
 - traps, 85, 1178
 - triage dumps, 1127
 - trimming, working set, 1155
 - triple faults, 1151
 - Triple-DES (3DES), 990–991
 - troubleshooting file systems, 908–910
 - Trusted Computer System Evaluation Criteria (TCSEC), 451–453
 - trusted facility management, 453, 1178
 - trusted path functionality, 453, 1178
 - Trusted Platform Module. *See* TPM (Trusted Platform Module)
 - TSCs (time stamp counters), 759–760
 - TxF (Transactional NTFS)
 - \$TXT_DATA attribute, 971–973
 - APIs for, 968–969
 - architecture of, 965
 - CLRs (compensating log records), 973
 - defined, 1177
 - directories for, 941
 - Fsutil.exe transaction queries, 968
 - isolation of transactions, 966–967
 - KTM with, 240, 973
 - listing current transactions, 968
 - log files, 970
 - logging implementation, 973
 - LSN storage by, 972
 - LSNs, base and restart, 973
 - MFT, permanent data stores for TxF, 971–973
 - recovery implementation of, 973–974
 - redo passes, 973–974
 - redo records, 973
 - resource managers of, 969–971
 - rollback criteria, 967
 - TOPS files, 970–971, 973–974
 - Transact-demo.exe tool, 967–968
 - transaction tables, 974
 - TxIDs, 972
 - undo passes, 974
 - undo records, 973
 - undo-next LSNs, 973
 - USN records, 972–973
 - TxR (Transactional Registry)
 - APIs for, 260
 - CommitTransaction function, 261
 - CreateTransaction function, 260–261
 - defined, 1177–1178
 - internal log files, 261
 - isolation levels, 261
 - KTM with, 240
 - logging operations, 260–261
 - resource manager (RM), 261–262
 - subkeys, deletions of, 260
 - type initializer fields, 146
 - type objects
 - defined, 143, 1178
 - initializer fields, table of, 146
 - uniform formats of objects with, 142
 - viewing, 144–145
- ## U
- UAC (User Account Control)
 - application-compatibility shims, 522–523
 - compatibility flags, using to run applications, 533
 - compatible manifests with, 522
 - defined, 1178
 - digital signatures, effects of, 529–530
 - domain-based elevation, 529
 - elevation dialog boxes, 529–530
 - elevations, 528–533
 - file virtualization, 523–526
 - File Virtualization Filter Driver, 524
 - filtered admin tokens with, 483–484
 - HKEY_CLASSES_ROOT, 255
 - integrity levels of objects, impact of, 471–472
 - issues and solutions of, 520–521
 - LSASS access token generation, 80
 - namespace virtualization, 521–528
 - purpose of, 520–521
 - registry virtualization, 526–528
 - requesting administrative rights, 531–533
 - super privileges with, 509

- trustinfo elements in manifests, 532
 - UIPI bypasses, 532
 - viewing virtualization status, 522
 - virtualization status, changing, 525
 - UDF (Universal Disk Format), 891
 - UDP, RPC transport interface for, 1016
 - UI Access flag, 494
 - UIPI (User Interface Privilege Isolation)
 - integrity levels for messaging, 493–494
 - UAC bypasses, 532
 - UMDF (User-Mode Driver Framework)
 - architecture of, 616–619
 - defined, 70, 542
 - driver host processes, 616
 - driver manager, 618
 - host processes, 618
 - I/O manager with, 617
 - kernel-mode drivers for, 618
 - KMDF compared to, 616
 - protocol device classes, 616
 - reflectors, 617
 - stacks of drivers, 616–617
 - UNC (Universal Naming Convention)
 - defined, 1021
 - MUP. *See* MUP (Multiple UNC Provider)
 - undocumented interfaces, 72–73
 - Unicode
 - defined, 1178
 - NTFS names based on, 922–923
 - Windows internal use of, 23–24
 - uniprocessor flag, 439
 - Universal Disk Format (UDF), 891
 - Universal Naming Convention. *See* UNC (Universal Naming Convention)
 - Universal Plug and Play (UPnP), 1032–1033
 - UNIX. *See* POSIX; SUA (Subsystem for Unix-based Applications)
 - unkillable processes, debugging, 590–592
 - unload routines, 550
 - unresponsive systems, 1147–1150
 - update records, 1178
 - updates (hotpatches), 242–244
 - UPnP (Universal Plug and Play), 1032–1033
 - USB
 - PIDs (product IDs), 630
 - Remote NDIS for, 1060–1062
 - stop code for, 1123
 - UMDF with, 616, 618–619
 - user-mode drivers, 70
 - VIDs (vendor IDs), 630
 - use after free bugs, 1140
 - User Account Control. *See* UAC (User Account Control)
 - user address spaces
 - ASLR for, 757–761
 - DLL image files in, 757
 - dynamic management of, 757
 - heap randomization, 760
 - image randomization, 759–760
 - layout of, 757–758
 - quotas for, 756
 - stack randomization, 760
 - types for allocations, 758
 - viewing with Vmmap utility, 758–759
 - user applications, types of, 36
 - user data, system global storage issue, 521–522
 - USER functions
 - purpose of, 54
 - windowing and graphics system, 37
 - User Interface Privilege Isolation. *See* UIPI (User Interface Privilege Isolation)
 - user mode
 - address space of, 16
 - APCs, 113
 - attaching Debugging Tools to processes, 27
 - AuthZ API security, 500–501
 - components of system architecture, 36
 - critical sections, staying in, 194
 - debugging framework of executive, 60
 - defined, 16, 1178
 - DLL function calls remaining in, 53
 - environment subsystem server processes, 36
 - image loader initialization of processes, 222
 - performance counters for, 17–19
 - services processes, 36
 - switching with kernel mode, 17
 - system support processes, 36
 - Windows subsystem use of, 56
 - user names
 - user logon request mechanics, 516–517
 - Winlogon processing of, 79–80
 - user process address space BCD option, 1081
 - user profiles
 - HKEY_USERS, 253–254
 - storage locations for, 254
 - User Profiles management dialog box, 254
 - USER service, service descriptor tables for, 130–133
 - user stacks, 785
 - Userinit
 - LSASS invoking of, 80
 - purpose of, 80
 - safe mode awareness, 1104
 - user-mode debugging
 - boot initialization, 1090
 - consumers, 217
 - create events, 217
 - DbgUi APIs, 216–218
 - debug events, table of, 216–217
 - debug object, 216
 - debug ports, 216
 - defined, 1178
 - handles management, 219
 - kernel support for, 216–217
 - Kernel32.dll support for, 219
 - load dll events, 217
 - mechanics of debugger requests, 217
 - modules supporting, 216
 - native support for, 217–219
 - Ntdll.dll, 217–218
 - producers, 217
 - viewing debugger objects, 218–219
 - Windows subsystem support for, 219
 - User-Mode Driver Framework (UMDF), 70
 - user-mode drivers, 70
 - users, super, 509
 - Usndump.exe change journal dumps, 947–959
- ## V
- VACBs (virtual address control blocks)
 - array structures, 860–862
 - defined, 1178–1179
 - index arrays, 863–865, 1178
 - priority mapping levels, 860
 - views, cache, 854–855
 - VADs (virtual address descriptors)
 - address translation role, 765
 - defined, 13, 1179
 - demand paging, 787
 - process VADs, 788–789
 - prototype PTEs with, 776–777
 - purpose of, 787–788
 - rotate VADs, 790
 - structure of, 788
 - viewing, 789
 - Valid page state, 804

- VCNs (virtual cluster numbers)
 - arrangement of, 938
 - defined, 1179
 - mappings to LCNs, 939–940, 950–951
- VDDs (virtual device drivers), 542, 1179
- VDM (virtual DOS machine)
 - processes, 54, 353–354
- VDS (Virtual Disk Service), 675–677
- vectored exception handling, 116
- versions of Microsoft Windows
 - checked build version, 47–49
 - client. *See* client versions of Windows
 - determining edition booted, 45–46
 - key differences between, 44
 - list of, 43–44
 - optimization differences between, 46
 - physical memory limits for, 818–822
 - server. *See* server versions of Windows
 - table of, 1
- video
 - drivers, GDI direct access to, 55
 - GDI. *See* GDI (Graphics Device Interface)
 - mapped memory for, 821–822
 - MMCSS with, 430–434
 - rotate VADs for drivers, 790
 - VGA mode to default resolution change during boots, 1096
- VIDs (vendor IDs), 1158
- views, section object, 710, 1179
- views, VACB, 854–855, 862–863
- virtual address control blocks (VACBs). *See* VACBs (virtual address control blocks)
- virtual address descriptors. *See* VADs (virtual address descriptors)
- virtual address space layouts
 - 64-bit, 745–746
 - 64-bit addressing limitations, 749–751
 - escaping 2GB process limits in x86, 738–739
 - high addresses, forcing allocation of, 739–740
 - IA64, 746
 - large address space awareness, 738–739
 - process address space overview, 736
 - session spaces, 736
 - shareable memory, 758
 - system space components, 736–737
 - types of data mapped into, 736
 - user address space layouts, 757–758
 - x86 address space layouts, 737–740
 - x86 session space layouts, 740–743
 - x86 system address space, 740
- virtual address spaces
 - ASLR for user address space, 757–761
 - AWE functions with, 719–721
 - committing pages, 706–708
 - components of addresses for translations, 763
 - defined, 1179
 - dynamic system management of, 751–755
 - freeing virtual memory pages, 707
 - layouts of. *See* virtual address space layouts
 - limiting range size by type, 754–755
 - mapping to physical pages. *See* address translation
 - nonpaged pool space. *See* nonpaged pools
 - nonpaged pools
 - paged pool space. *See* paged pools
 - pages, defined, 705. *See also* pages; paging virtual memory to disks
 - placement policies, 827–828
 - pools. *See* pools, memory
 - priority numbers, 809–812
 - process size limits, 699
 - quotas, 756
 - reclaiming memory, 752–754
 - reserving pages, 706–708
 - section objects for mapping. *See* section objects
 - sizes of pages, 705–706
 - types of addresses, determining, 753
 - types of addresses, viewing usage, 754
 - viewing with Vmmap utility, 758–759
 - working sets of. *See* working sets
- virtual address translation. *See* address translation
- virtual block caching, 852, 1179
- Virtual Disk Service. *See* VDS (Virtual Disk Service)
- virtual DOS machine (VDM)
 - processes, 54
- virtual memory system
 - !vm command, 1145
 - 64-bit address spaces, 15
 - address space of. *See* virtual address spaces
 - AWE extensions of, 15
 - cache manager with, 854–855
 - job limits for, 447
 - kernel mode address space, 16
 - mapping, 14
 - memory protection, 712
 - paging to disks, 14. *See also* paging virtual memory to disks
 - purpose of, 14
 - RAM optimization software effect on, 846–847
 - size of address spaces, 14–15
 - virtual memory, disk storage of. *See* page files; paging virtual memory to disks
 - virtual pages. *See* paging virtual memory to disks; virtual memory system
 - VirtualAlloc functions, 706–708, 720
 - virtualization, 362, 1179. *See also* Hyper-V
 - virtualization, UAC, 521–528
 - VirtualLock function, 707–708
 - virus scanner use of file system filter drivers, 907
 - Vista, Windows. *See* client versions of Windows
 - VMBus, 236–237
 - VMKs (volume master keys)
 - purpose of, 679–681
 - sealing, 681–682
 - unsealing, 684
 - Vmmap utility, 758–759
 - volume files, 1179
 - volume manager
 - application direct requests, 674
 - bad-cluster recoveries, 985–988
 - basic disk management, 655
 - DMDiskManager, 661
 - driver nature of, 655
 - drivers, layering with file system drivers, 544
 - dynamic disk, 661
 - I/O operations, 674–675
 - mirrored volumes, 664–667
 - mount operations, 655, 668
 - multipartition volume management, 661–667
 - names of device objects, 650–651
 - object creation, 655
 - partition possession mechanics, 652
 - PnP manager, coordination with, 655
 - spanned volumes, 662–663

- striped volumes, 663–664
- VolMgr, 661–662
- VolMgrX, 661–662
- volume master keys. *See* VMKs (volume master keys)
- volume sets, 662, 1179–1180. *See also* spanned volumes
- Volume Shadow Copy Service. *See* VSS (Volume Shadow Copy Service)
- volumes
 - basic disks, equivalence to partitions, 655
 - boot volume creation, 660
 - boot volumes defined, 1155
 - cache manager recoverable file system support, 853–854
 - compression, testing for, 951–952
 - defined, 1179
 - dismount operations, 897
 - FAT, 892–894
 - foreign volumes, BitLocker, 685
 - I/O operations of, 674–675
 - MFTs (master file tables), 938–941
 - mirrored volumes, 664–667
 - mount points for, 669–670
 - namespace mechanism, 667
 - NTFS, 937
 - privilege for task performance, 507
 - quotas, per-user, 928
 - RAID level 0, 663–664
 - RAID level 1, 664–666
 - RAID level 5, 666–667
 - shrinking, 933
 - simple compared to multipartition, 645
 - snapshots, scheduling for, 697–698
 - spanned volumes, 662–663
 - striped volumes, 663–664
 - system volumes, 660
 - types of multipartition volumes, 662
 - VPBs (volume parameter blocks), 670–673
- VPBs (volume parameter blocks)
 - file system driver interactions with, 896
 - mount mechanics with, 670–673
- VPNs (Virtual Private Networks), 1066
- VSS (Volume Shadow Copy Service)
 - architecture of, 688–689
 - backups, using for, 692–693
 - clone shadow copies, 688
 - copy-on-write shadow copies, 688
 - defined, 688
 - device objects of, 692–693
 - differential copies, 688

- operation of, 689–690
- Previous Versions feature use of, 693–695, 698
- scheduling snapshots for volumes, 697–698
- shadow copies defined, 1174
- Shadow Copy Provider, 690–692
- steps for copy creation, 689–690
- System Restore use of, 694
- transportable copies, 690
- types of copy mechanisms, 688
- Volsnap (Volume Shadow Copy Driver), 690–692
- Vssadmin utility for viewing copies, 696–697

W

- wait blocks
 - defined, 183, 1180
 - KTHREAD fields for, 373
 - relationship to threads and dispatcher objects, 183
 - viewing, 184, 186
- wait hints, 1180
- wait queues, 114, 413
- wait state
 - alertable wait states, 1153–1154
 - boosts during executive resource waits, 422–423
 - priority level boosts after events and semaphore waits, 421–422
 - voluntary switching from, 413
 - Waiting thread execution state, 401
- WaitForMultipleObjects function, 178–179
- WaitForSingleObject function, 178–179
- wake-on-LAN, 1054
- watermarked desktop option, 1080
- WBEM (Web-Based Enterprise Management)
 - basis for WMI, 318
 - CIM specification, 319–320
 - Wbetest tool, 323
- WDF (Windows Driver Foundation)
 - defined, 1180
 - KMDF compared to UMDF, 70
 - purpose of, 70
- Wdfkd.dll, 608
- WDI (Windows Diagnostic Infrastructure), 61, 329–333, 1180
- WDK (Windows Driver Kit)
 - ASSERT macro, 48
 - documentation, 31
 - downloading, 31
 - file system drivers, required to build, 895–896
- header files, 31–32
- Windows executive functions, 58
- WDM (Windows Driver Model)
 - drivers
 - bus drivers, 69, 542–543
 - categories of, 542–543
 - defined, 69, 1180
 - filter drivers, 69–70
 - function drivers, 69
 - interactions of, 543
 - types of drivers, 69
 - WMI routines. *See* WMI (Windows Management Instrumentation)
- Web access APIs
 - HTTP Server API, 1019–1021
 - purpose of, 1018
 - WinHTTP, 1019
 - Winlnet, 1019
- Web servers
 - cache functions, 1020–1021
 - configuration functions for, 1021
 - HTTP Server API, 1019–1021
 - Http.sys, 1019–1020
 - port range reservations, 1019
 - request queues, 1019–1020
 - URL groups, 1019–1020
 - well-known addresses for
 - Winsock, 1009
 - WinHTTP API for, 1019
 - Winsock extensions for, 1008–1009
- Web services, DPWS, 1033
- WebDAV, 897
- well-known addresses for Winsock, 1009
- well-known installers, 1111–1112
- WER (Windows Error Reporting)
 - ALPC error ports, 124
 - analysis at Microsoft, 1133–1134
 - configuration options, 122–124
 - configuring, 120
 - crashed unhandled exception filters, 122
 - debugger option, 121
 - defined, 120
 - error report options, 121
 - Microsoft Systems Center 2007 with, 121
 - OCA (Online Crash Analysis), 1133–1134
 - purpose of, 1131
 - sending of reports to Microsoft, 1133
 - settings screen for, 1131
 - silent process deaths, 122–125
 - solutions suggested to users, 1133–1134
 - unhandled exceptions triggering mechanism, 121
 - user input requests, 1133

- WER (Windows Error Reporting), *continued*
 - Wercon.exe, launching, 120
 - WerFault dump operations, 1132
 - WerFault execution during logons, 1133
 - WerFault pre-dump operations, 1130–1131
 - WerFault.exe, 121
- WFP (Windows File Protection)
 - callout drivers, 1180
 - differences with WRP, 1111
- WFP (Windows Filtering Platform)
 - BFE (base filtering engine), 1047, 1049–1050
 - callout drivers, 1004, 1047
 - components of, 1047
 - defined, 1004, 1180
 - filter engine, 1047
 - IPSec callout driver, 1052
 - NAT support, 1049
 - shims, 1047
 - Windows Firewall support, 1049–1050
- WHQL (Windows Hardware Quality Labs), 538–539, 635
- Win16, process creation for, 353–354
- Win32 API. *See also* Windows API
 - defined, 2
 - history of, 3–4
- Win32k.sys. *See also* Windows subsystem
 - components contained in, 54
 - defined, 37
 - GDI components, 54–55
 - kernel load during boots, 1096
 - system mapped views for, 736
 - threads, structures for, 371
- WinDbg
 - crash dump analysis with, 1134, 1137
 - kernel debugging with, 28
 - loader databases, viewing, 223–225
 - viewing user-mode debugger objects, 218–219
- window stations
 - defined, 1180
 - opening by SCM, 305
 - service access to, 297–299
 - Winlogon, creation of, 515
- windows
 - priority level boosts for, 425–427
 - quantums, effect of windows in foreground on, 410
 - windows and graphics system, 37
- Windows 3.1, 353–354
- Windows API
 - categories of callable functions, 2
 - defined, 2, 1180
 - documentation in SDK, 2
 - functions, 4
 - Heap interfaces, 730
 - history of, 3–4
 - memory manager functions, 704
 - memory protection options, 712–713
- Windows Boot Loader. *See* Winload (Windows Boot Loader)
- Windows Boot Manager. *See* Bootmgr (Windows Boot Manager)
- Windows Diagnostic Infrastructure. *See* WDI (Windows Diagnostic Infrastructure)
- Windows Driver Foundation. *See* WDF (Windows Driver Foundation)
- Windows Driver Kit. *See* WDK (Windows Driver Kit)
- Windows drivers, 1180. *See also* device drivers
- Windows Error Reporting (WER). *See* WER (Windows Error Reporting)
- Windows executive. *See* executive, Windows
- Windows Filtering Platform. *See* WFP (Windows Filtering Platform)
- Windows Firewall
 - connection security rules management, 1052
 - IPSec support for, 1051
 - network restriction rules, 296–297
 - WFP support for, 1049–1050
- Windows hardware error architecture, 61
- Windows Internet Name Service (WINS), 1039
- Windows Management Instrumentation. *See* WMI (Windows Management Instrumentation)
- Windows Media Player, 166, 430–434
- Windows Memory Diagnostic Tool, 1106
- Windows native API, 1094
- Windows Networking API, 1033–1036
- Windows Next Generation TCP/IP Stack. *See* Next Generation TCP/IP Stack
- Windows NT
 - legacy drivers, 1166
 - networking APIs, 1067
- Windows Recovery Environment (WinRE), 1106–1108
- Windows Resources Protection (WRP), 1111–1112
- Windows Server. *See* server versions of Windows
- Windows Server Core, 231
- Windows services. *See* services
- Windows Setup. *See* Setup, Windows
- Windows Sockets. *See* Winsock
- Windows Software Development Kit (SDK)
 - latest version of, obtaining, 31
 - Windows API documentation in, 2
- Windows subsystem. *See also* Win32k.sys
 - calls from other subsystems, 53
 - components of, 54
 - CreateProcess operations with, 360–362
 - DirectX wrappers, 54
 - DLLs loaded by Csrss.exe, 54
 - DLLs translating API functions to service calls, 54
 - DLLs, application calls to, 53
 - executive objects, 136
 - file specification of, 51
 - GDI, 54–56
 - importance of, 51
 - kernel load during boots, 1096
 - kernel-mode device driver, 54
 - launch by Smss during boot process, 1094
 - startup values, 51
 - threads, structures for, 371
 - undocumented interfaces, 72–73
 - user interface control calls, 54
 - user-mode functions, 56
 - window manager, 54
- Windows System Resource Manager (WSRM), 398–399
- Windows Task Manager
 - Performance tab, 702–703
- Windows Transport Driver Interface standard, 1003
- Windows Update
 - hotpatches, 242–244
 - TxR use by, 261
- Windows Vista. *See* client versions of Windows
- Windows, versions of. *See* versions of Microsoft Windows
- WindowStation object type
 - components represented by, 137
 - okay to close object method, 149
 - open object method, 148
- WinHTTP, 1019
- WinInet, 1019
- Wininit
 - boot steps of, 1097
 - critical nature of, 1096

- defined, 1075
- Smss launches of during boots, 1096
- Winlogon steps compared to, 516
- Winload (Windows Boot Loader)
 - autorecovery BCD option for, 1080
 - BCD options for, table of, 1080–1084
 - boot volume file load steps, 1085–1086
 - configuration queries by, 1085
 - defined, 1074
 - loading by Bootmgr, 1084
 - registry fixes by, 279
 - storage support in, 646
 - troubleshooting menu, BCD option for, 1080
 - WinRE, automatic launches of, 1107
- Winlogon
 - activity triggers for, 80
 - authentication, 79
 - authentication calls to Lsass, 514
 - boot steps of, 1097–1098
 - credential providers, 79, 514
 - Ctrl+Alt+Delete logon attention, 515–516
 - defined, 455
 - desktops created by, 515
 - locked nature of desktop, 516
 - LogonUI, 79
 - LSASS calls, 80
 - Lsass connection creation, 515
 - passwords, 79–80
 - request handling by, 79
 - responsibilities of, 513
 - SAS notifications, 515
 - SCM, notification of success to, 308–309
 - secure attention sequence, 79
 - session SIDs, 462
 - shutdown initiation, 311
 - shutdown mechanics, 1115–1117
 - Smss launches of during boots, 1096
 - system initialization steps, 515–516
 - user logon steps, 516–518
 - user names, 79–80
 - window station creation, 515
- WinObj tool
 - ACL display for objects, 157–158
 - ALPC port objects, viewing, 203
 - base object names, listing, 165
 - device object name display, 552
 - disk objects, displaying, 650–651
 - file systems, viewing registered, 900
 - memory condition events, 835
 - object manager namespace, viewing, 133
 - purpose of, 133
 - session namespaces, 168
 - type objects, listing, 144
- WinRE (Windows Recovery Environment), 1106–1108
- Winresume, 1074
- WINS (Windows Internet Name Service), 1039
- Winsock
 - accept operations, 1007–1008
 - AcceptEx function, 1008
 - AFD (Ancillary Function Driver), 1012
 - binds, 1007
 - BSD Sockets basis of, 1006
 - client operation, 1007
 - ConnectEx function, 1009
 - connection backlogs, 1007–1008
 - connectionless operation, 1008
 - connection-oriented operation, 1007–1008
 - datagrams, 1008
 - DisconnectEx function, 1009
 - DLLs for, 1011
 - extensibility by third parties, 1009–1011
 - extension APIs, 1008–1009
 - feature list, 1006–1007
 - file handles for sockets, 1012
 - Helper libraries, 1011
 - history of, 1006
 - impersonation functions, 1009
 - implementation of, 1011–1012
 - initialization, 1007
 - IPv6 support, 1007
 - kernel API. *See* WSK (Winsock Kernel)
 - listen operations, 1007–1008
 - Mswsock.dll, 1011
 - multipoint message support, 1007
 - name-resolution functions, 1009
 - namespace support, 1007, 1009
 - PNRP for IPv6, 1040
 - polling sockets, 1007–1008
 - protocol independence of, 1007
 - receive functions, 1007–1008
 - RPC run-time DLL use of, 1017–1018
 - select functions, 1007–1008
 - send functions, 1007–1008
 - server operation, 1007–1008
 - showing providers with Netsh, 1010
 - socket creation, 1007
 - SPI (service provider interface), 1009
 - TransmitFile function, 1008–1009
 - TransmitPackets function, 1009
 - transport service providers, adding, 1009
 - well-known addresses with, 1009
 - Ws2_32.dll, 1011
 - WSAPoll functions, 1007–1008
- Winsock Kernel. *See* WSK (Winsock Kernel)
- WinSta0, 167
- WMI (Windows Management Instrumentation)
 - APIs for, 319
 - architecture of, 318–319
 - association classes, 325–327
 - CIM with, 320–323
 - CIMOM, 319
 - COM API, 319
 - defined, 318
 - dynamic designator, 323
 - Event Log provider example, 320
 - infrastructure, 319
 - managed objects, 318–319
 - management applications, 318–319
 - MOF with, 320–323
 - namespace for, 324–325
 - Object Browser, 325–326
 - providers, 318–320
 - registry settings, 328
 - scripting support, 326
 - security issues, 329
 - service mechanics, 327–328
 - SQL support, 327
 - WDM routines, 1180
 - WMI utility, 328
- WNet API, 1033–1036
- work items, 1181
- worker factories
 - API for, 386
 - defined, 386
 - Factory object, 387
 - I/O completion ports, 388
 - kernel queue support for, 388
 - object manager type for, 387
 - responsibilities of, 387–388
 - scalability benefit of, 387
 - stop codes from bad references, 1122
 - viewing, 388–390
 - Vista compared to pre-Vista, 386–387
- worker threads. *See* system worker threads
- working set manager
 - aging pages, 1153
 - automatic trimming, 1155
 - calls from and to the balance set manager, 831–832
 - defined, 1181
 - purpose of, 700
 - trimming initialization by, 829

1232 working sets

working sets

- automatic trimming of, 829, 1155
 - balance set manager context of, 831–832
 - clock algorithm for, 1156
 - defined, 699, 1181
 - demand paging, 823
 - entries, viewing, 830
 - freeing a page during PFN operations, 809
 - lock implementation for, 701
 - logical prefetching, 823–827
 - manager. *See* working set manager
 - mechanics of, 829
 - PFN index field for, 816
 - placement policies, 827–828
 - process working sets, 822, 1171
 - quotas for, 756
 - replacement policies, 827–828
 - session working sets, 823
 - size of, viewing, 830
 - sizing of, 828–829
 - swapper mechanics, 832
 - system. *See* system working sets
 - system cache, 737
- Workstation service, 897
- Wow64
- 16-bit applications, 215
 - 16-bit installer applications, 215
 - address space for, 211, 746
 - architecture of, 211
 - defined, 211
 - device driver issues, 214–215
 - DLLs for, 211
 - exception dispatching, 212
 - file system redirection, 212–213
 - I/O control functions, 214–215
 - IA64 (Itanium) issues, 215
 - printing from 32-bit processes, 215
 - registry reflected keys, 214
 - registry splits, 213–214
 - restrictions for, 215
 - Synnative virtual directory, 213
 - system calls, 212
 - user callbacks, 212
- WPD (Windows Portable Device) Framework, 616
- WQL (WMI Query Language), 327
- write I/O operations. *See also* I/O processing
- fast I/O steps for, 873–874
 - lazy writing, 877–883
 - write throttling, 885–886
 - write-through caching, 883
- write throttling, 1181
- write-back caching, 877–883
- WriteFile function
- file system driver operations after calling, 902–906
 - mailslots with, 1024–1025
 - named pipes with, 1024–1025
 - synchronous I/O, 563
- WriteFileEx
- I/O completion, 577
 - user-mode APCs with, 113
- WriteProcessMemory function, 706–707
- writing, memory protection against, 712–713
- WRP (Windows Resources Protection), 1111–1112
- WS Discovery, 1033

- Ws2_32.dll, 1011. *See also* Winsock
- WSK (Winsock Kernel)
- advantages of, 1012
 - binding of applications, 1013
 - defined, 1004
 - events of, 1013
 - extension interfaces, 1014
 - implementation, 1012–1014
 - NMR (Network Module Registrar) with, 1012–1013
 - registration functions, 1013
 - socket categories, 1013
 - TDI SPI interface availability in, 1012
- WSRM (Windows System Resource Manager), 398–399

X

- x.509 certificates, 1051
 - x64 Windows. *See* 64-bit Windows versions (x64)
 - x86 systems
 - address space layouts, 737–740
 - session space layouts, 740–743
 - system address space layouts, 740
- XP, Windows. *See* client versions of Windows

Z

- zero page threads, 701, 808, 1181
- Zeroed page state, 804
- zone security system, 921

Mark Russinovich



Mark Russinovich is a technical fellow in the Windows Core Operating System Division. He is a member of the core team that provides architectural direction and oversight across Windows, with a focus on security and virtualization. Mark is currently working on the technical direction and architectural plan for Windows 8. He also continues developing tools for the Windows Sysinternals Web site, the most popular TechNet subsite, with 2 million downloads per month. Mark consistently delivers the top-rated sessions at major IT and developer conferences, including Microsoft TechEd, TechReady, and WinHEC.

Mark has written dozens of magazine articles and serves as contributing editor for both *Microsoft TechNet* and *Windows IT Pro* magazines. Mark joined Microsoft in 2006 when the 85-person company he cofounded 10 years earlier, Winternals Software, was acquired along with Sysinternals. At Winternals he was chief software architect, defining the business and technical direction for the company.

David Solomon



David Solomon, president of David Solomon Expert Seminars (www.solsem.com), has focused on explaining the internals of the Microsoft Windows NT operating system line since 1992. He has taught his world-renowned Windows internals classes to thousands of developers and IT professionals worldwide. His clients include all the major software and hardware companies, including Microsoft. He was nominated a Microsoft Most Valuable Professional in 1993 and from 2005–2008.

Prior to starting his own company, David worked for nine years as a project leader and developer in the VMS operating system development group at Digital Equipment Corporation. His first book was entitled *Windows NT for Open VMS Professionals* (Digital Press/Butterworth Heinemann, 1996). It explained Windows NT to VMS-knowledgeable programmers and system administrators. His second book, *Inside Windows NT, Second Edition* (Microsoft Press, 1998), covered the internals of Windows NT 4.0. Since the third edition (*Inside Windows 2000*) David has coauthored this book series with Mark Russinovich.

In addition to organizing and teaching seminars, David is a regular speaker at technical conferences such as Microsoft TechEd and Microsoft PDCs. He has also served as technical chair for several past Windows NT conferences. When he's not researching Windows, David enjoys sailing, reading, and watching *Star Trek*.

Alex Ionescu



Alex Ionescu is the founder of Winsider Seminars & Solutions Inc., specializing in low-level system software for administrators and developers. He also teaches Windows internals courses for David Solomon Expert Seminars, including at Microsoft. Alex was the lead kernel developer for ReactOS, an open source clone of Windows XP/Windows Server 2003 written from scratch, for which he wrote most of the Windows NT-based kernel.

Alex is also very active in the security research community, discovering and reporting several vulnerabilities related to the Windows kernel and presenting talks at conferences such as Blackhat and Recon. Alex's experience in operating system design and kernel coding dates back to his early adolescence, when he first played with John Fine's educational operating system and kernel and boot loader code. Since then he has been active in the area of NT kernel development, offering help and advice for driver developers, as well as in the NT reverse engineering and security fields, where he has published a number of articles and source code, such as documentation for the Linux NTFS project, extensive papers on the Visual Basic metadata and pseudo-code format and NTFS structures and data streams. In the last three years, he has contributed to patches and development in two major commercially used operating system kernels.

Windows Internals, Fifth Edition and MCTS Exam 70-660

This book delivers in-depth, kernel-level insights into the Windows OS. You may also find it helpful when preparing for the following topics in Microsoft Certification Exam 70-660:

Exam Objectives/Skills	See Topic-Related Coverage Here
Identifying Architectural Components	
Identify memory types and mechanisms	Chapter 9
Identify I/O mechanisms	Chapter 7, see "I/O Processing"; Chapter 3
Identify subsystems	Chapter 2, see "Key System Components"
Identify processor functions and architecture	Chapter 3, see "System Service Dispatching"; Chapter 5, see "Overview of Windows Scheduling"
Identify processes and threads	Chapter 5, see "Process Internals," "Thread Internals," and "Thread Scheduling"
Designing Solutions	
Optimize a system for its drivers	Chapter 7, see "Device Drivers" and "The Plug and Play (PnP) Manager"; Chapter 3, see "Interrupt Dispatching" and "System Worker Threads"; Chapter 9, see "Driver Verifier"
Design applications	Chapter 3, see "Windows Global Flags," "Exception Dispatching," and "Synchronization"; Chapter 5, see "Thread Internals"; Chapter 9; Chapter 6
Deploy compatible applications	Chapter 3, see "Windows Global Flags" (limited coverage)
Identify optimal I/O models for applications	Chapter 7, see "I/O Processing" and "I/O Completion Ports"
Monitoring Windows	
Monitor I/O latency	Chapters 1, 7, and 8
Monitor I/O throughput	Chapters 3, 7, 8, and 11
Monitor memory usage	Chapter 9
Monitor CPU utilization	Chapters 5 and 14
Monitor handled and unhandled exceptions	Chapter 3, see "Exception Dispatching"; Chapter 14, see "Windows Error Reporting"
Analyzing User Mode	
Analyze heap leaks	Chapter 9, see "Heap Manager"
Analyze heap corruption	Chapter 9, see "Heap Manager"
Handle leaks	Chapter 3, see "Object Handles and the Process Handle Table"
Resolve image load issues	Chapter 3, see "Image Loader"
Analyze services and host processes	Chapter 4, see "Services"
Analyze cross-process application calls	Not covered
Analyze the modification of executables at run time	Not covered
Analyze GUI performance issues	Not covered

Exam Objectives/Skills	See Topic-Related Coverage Here
Analyzing Kernel Mode	
Find and identify objects in object manager namespaces and identify the objects' attributes	Chapter 3, see "Object Manager"
Analyze Plug and Play (PnP) device failure	Chapter 7, see "The Plug and Play (PnP) Manager"; Chapter 13
Analyze pool corruption	Chapters 1 and 3
Analyze pool leaks	Chapters 1, 3, 9, and 14
Isolate the root cause of S state failure	Chapter 7, see "Power Manager"
Analyze kernel mode CPU utilization	Chapter 14, see "Hung or Unresponsive Systems"; Chapter 3
Debugging Windows	
Debug memory	Chapters 9 and 14
Identify a pending I/O	Chapter 7
Identify a blocking thread	Chapter 3, see "Synchronization"; Chapter 5, see "Thread Scheduling"
Identify a runaway thread	Chapter 5, see "Thread Scheduling"; Chapter 14
Debug kernel crash dumps	Chapters 14, 3, and 5
Debug user crash dumps	Not covered
Set up the debugger	Chapters 14, 13, and 1

For complete information on MCTS Exam 70-660, go to www.microsoft.com/learning/en/us/Exams/70-660.aspx. And for more information on Microsoft certifications, visit www.microsoft.com/learning.