Microsoft

Covers advanced troubleshooting and crash analysis

4

Fourth Edition

Microsoft® WINDOWS® INTERNALS

Microsoft Windows Server 2003, Windows XP, and Windows 2000

Foreword by Jim Allchin
Historical perspective by David N. Cutler

Mark E. Russinovich
David Solomon

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

Copyright © 2005 by David Solomon, Mark Russinovich

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 2005921847

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 9 8 7 6 5 4

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/learning/. Send comments to rkpinput@microsoft.com.

Microsoft, Active Desktop, Active Directory, ActiveX, DirectX, Microsoft Press, MSDN, MS-DOS, Outlook, PowerPoint, Visual Basic, Visual C++, Visual Studio, Win32, Windows, Windows NT, Windows Server, and WinFX are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. 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 Editors: Robin Van Steenburgh, Ben Ryan
Project Editor: Valerie Woolley
Development Editor: Sally Stickney
Copy Editor: Roger LeBlanc
Indexer: Lynn Armstrong

SubAssy Part No. X11-16607
Body Part No. X11-16608
Section Part No. X11-20530

To Dave Cutler, father of the Windows kernel

Contents at a Glance

Table of Contents

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

Microsoft is interested in hearing your feedback about this publication 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/*

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

Microsoft is interested in hearing your feedback about this publication 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/*

Historical Perspective

Again I find myself indebted to David Solomon and Mark Russinovich for providing the opportunity to write a few words about their newest edition to a series of books on Windows Internals. It has been over three years since the last publication in this series, and this passage of time has seen two major releases: a very significant update to the client system, and another very significant update to the server system, currently being readied for shipment.

Two of the growing problems faced by the authors of a book such as this are tracing the implementation evolution of the Microsoft Windows NT system and documenting the way in which feature implementation has changed in each version. To this end, the authors have done a remarkable job of providing examples and explanations throughout the book.

(Left to right) David Solomon, David Cutler, and Mark Russinovich

I first met David Solomon when I was working at Digital Equipment Corporation on the VMS operating system for VAX, and he was only 16. Since that time, he has been involved with operating system development and teaching operating system internals. I met Mark Russinovich more recently but have been aware of his expertise in the area of operating systems for some time. He has done some amazing work, such as his NTFS file system running on Microsoft Windows 98 and his "live" Windows kernel debugger, which can be used to peer into the Windows system while it is running.

The beginnings of Windows NT started in October 1988 with a set of goals to produce a portable system that addressed OS/2 compatibility, security, POSIX, multiprocessing, integrated networking, and reliability. With the advent and huge success of Windows 3.0, the system goals were soon changed to natively address Windows compatibility directly and move OS/2 compatibility to a subsystem.

We originally thought we could produce the first Windows NT system in a little over two years. It actually ended up taking us four and a half years for the first release in the summer of 1993, and that release supported the Intel i386, Intel i486, and the MIPS R4000 processors. Six weeks later, we also introduced support for the Digital Alpha processors.

The first release of Windows NT was larger and slower than expected, so the next major push was a project called Daytona, named after the speedway in Florida. The main goals for this release were to reduce the size of the system, increase the speed of the system, and, of course, to make it more reliable. Six months after the release of Windows NT 3.5 in the fall of 1994, we released Windows NT 3.51, an updated version containing support for the IBM PowerPC processor.

The goal for the next version of Windows NT was to update the user interface to be compatible with Windows 95 and to incorporate the Cairo technologies that had been under development at Microsoft for a couple of years. This system took two more years to develop and was introduced in the summer of 1996 as Windows NT 4.0.

The following version of NT saw a name change to Windows 2000 and was the last system for which the client and server systems were released at the same time. This version was built on the same Windows NT technology as the previous versions and introduced significant new features such as Active Directory. Windows 2000 took three and a half years to produce and was the most tested and tuned version of Windows NT technology produced at the time. Windows 2000 was the culmination of over eleven years of development spanning implementations on four architectures.

At the end of Windows 2000 development, we embarked on an ambitious plan to implement new versions of the client and server systems, which would include new enhanced consumer features and improved server capabilities. As plans developed, it became clear that implementation of the server features would cause a lag in the implementation of the client features, and therefore, the releases were split. In August of 2001, Windows XP Professional and Windows XP Home Edition were released, and a little over a year later, in March of 2003, Microsoft Windows Server 2003 was released. In addition to the Intel x86 architecture, these systems contained support for the Intel IA-64, marking Windows NT's first move to 64-bit processing.

This book is the definitive work on the internal structures and workings of Windows XP and Windows Server 2003. In addition, it offers a glimpse into the future of Windows' move to 64-bit computing by covering AMD's introduction of the x64 architecture (AMD64) in 2003 and Intel's announced support (EM64T) in February 2004. A fully supported x64 client and server release is planned in the first half of 2005, and this book contains many insights into the implementation details of the x64 system.

The x64 architecture is the beginning of a new era for Windows NT at a time when the x86 architecture is beginning to show signs of old age. This architecture offers 32-bit x86 compatibility at speed to protect legacy software investments, and provides 64-bit addressing capability to address the most ambitious of new applications. This will protect 32-bit software

investments while providing Windows NT with a breath of new life well into the next decade and beyond.

Although the Windows NT system has undergone several name changes over the past several years, it remains entirely based on the original Windows NT code base. As time has marched on and invention has thrived, the implementation of many internal features has changed significantly. The authors have done a laudable job of assimilating the details of the Windows NT code base and its differing implementations from release to release and platform to platform, and of producing examples and tools that help the reader understand how things work. Every serious operating system developer should have a copy of this book on his or her desk.

David N. Cutler
Senior Distinguished Engineer
Microsoft Corporation

Foreword

Microsoft Windows has been a core part of my life for 14 years. During this time, over release after release, the operating system has evolved in breadth and depth. Producing Windows is one of the most important and complex projects in the world today. Easily 5,000 engineers work on Windows releases. Across virtually all cultures, Windows users comprise an entire spectrum from the largest mission-critical business to theyoungest child. Customers using Windows demand constant improvements in virtually every aspect—from being able to run the largest servers to being easy enough for a pre-schooler to use. Windows comes in many shapes and sizes, from embedded editions to media center editions to data center editions. All these products use the same core Windows internals, which evolve and improve with each release.

This is the definitive book on the core Windows internals. If you want to learn how Windows works internally, in the fastest way possible, then this is the book for you. Understanding all the pieces of such a large product is a daunting task. But if you start at the core concepts of the system and work out, the puzzle fits together a lot more easily. Just as Windows itself has evolved, so has the comprehensive nature of this book, now in its fourth edition. For years, we have used earlier editions of this material to train brand-new employees at Microsoft, so this material is tried and true.

If you're like me, you like to figure out how things really work. Reading "how to use" books or "tips and tricks" has never been sufficient for me. If you understand how something works internally, you know how to better use it, maximize performance and security, diagnose failures, and frankly, have more fun. If you're like me and want to see Windows from the "inside out," then you're starting in the right place.

David and Mark have done an outstanding job detailing the "inside" Windows technical story. The tools that they highlight are a great resource for direct hands-on training and diagnostics work. After you read this book, you'll have a much greater understanding of how the operating system fits together, the latest improvements made throughout the system, and how to get the most from these improvements.

It has been quite a journey—one that is still underway. So, start reading and dive deep into one of the most impressive operating systems ever created.

Jim Allchin
Group Vice President, Platforms
Microsoft Corporation

Acknowledgments

First, special thanks to the following people:

- **Dave Cutler,** Senior Distinguished Engineer and the original architect of Microsoft Windows NT**.** Dave originally approved David Solomon's source code access and has been supportive of his work to explain the internals of Windows NT through his training business as well as during the writing of Inside Windows NT, Second Edition, and Inside Microsoft Windows 2000, Third Edition. Besides reviewing the chapter on processes and threads, Dave answered many questions on the kernel architecture of the system and wrote a historical perspective for this edition.
- **Jim Allchin**, our executive sponsor, for writing the Foreword to this book and championing our cause within Microsoft.
- **Rob Short**, vice president, who made sure we had the resources we needed, as well as access to the relevant people.

We also thank two developers in the Windows division for writing new content that was incorporated into this edition:

- **Adrian Marinescu,** who wrote the greatly expanded heap manager section in the memory management chapter.
- **Samer Arafeh,** who wrote the description of Wow64.

Thanks to our friend Jeffrey Richter, for writing the "What about .NET and WinFX" sidebar in Chapter 1 and for continuing to remind us over many dinners together of his view on how few people should care about what we talk about in this book.

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

Murali Brahmadesam	Pat Hoffer	Daniel Pravat
Molly Brown	Anthony Jones	Dragos Sambotin
Duncan Bryce	Tom Jones	Jon Schwartz
Daniel Bucherer	Joseph Joy	Rob Short
Neal Christian	Shreeniwas Kelkar	Paul Sliwowicz
Neill Clift	Connie La Chasse	Chittur Subbaraman
Mike Danseglio	Mike Lai	Cristian Teodorescu
Joseph Davies	Paul Leach	Andre Vachon
Cenk Ergan	Gerald Maffeo	Landy Wang

Tom Fout	Aaron Margosis	Richard Ward
Nar Ganapathy	Iain McDonald	Brad Waters
David Golds	Kamen Moutafov	Bruce Worthington
Robert Gu	Adi Oltean	Mark Zbikowsk
Jeff Hamblin	Vince Orgovan	Khawar Zuberi

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

Thanks also to Jamie Hanrahan of Azius Developer Training (www.azius.com), who coauthored with David the original Windows Internal Architecture class on which the second edition was based. Jamie, who has a real knack for explaining complicated concepts in a simple and practical fashion, developed several of the explanations, diagrams, and figures.

Thanks to Dave Probert for hosting the share where review drafts were distributed to internal Microsoft reviewers.

And thanks to Jonathan Sloves of AMD for arranging AMD64 test systems to be sent to us to help us develop the 64-bit content and port some of the Sysinternals tools to x64.

Finally, we want to thank the following people from Microsoft Press for their contribution to this book:

- Robin van Steenburgh, acquisitions editor, for patiently working with us to complete this project.
- Sally Stickney, who, for a time, continued as our project editor, but later got drawn into the management vortex. We missed working with you this time!
- Valerie Woolley, who took over as project editor from Sally. You were great (and not as rough on us as Sally was for the last two editions)!
- Roger LeBlanc, who laboriously went through all our chapters to tighten up text, find inconsistencies, and in general bring the manuscript up to the high standards of Microsoft Press.

David Solomon and Mark Russinovich
September, 2004

Introduction

Microsoft Windows Internals, Fourth Edition is intended for advanced computer professionals (both developers and system administrators) who want to understand how the core components of the Microsoft Windows 2000, Windows XP, and Microsoft Windows Server 2003 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 it easier to troubleshoot system problems 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 terms and concepts used throughout the rest of the book. The next three chapters—"System Mechanisms," "Management Mechanisms," and "Startup and Shutdown"—describe key underlying mechanisms in the system. The next eight chapters explain the core components of the operating system: processes, threads, and jobs; memory management; security; the I/O system; storage management; the cache manager; file systems; and networking. The last chapter covers crash dump analysis.

History of the Book

This is the fourth 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 insight into the architecture and design of the system. *Inside Windows NT, Second Edition* (Microsoft Press, 1998) was written by David Solomon. It was updated to cover Windows NT 4.0 and had a greatly increased level of technical depth. *Inside Microsoft 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, as well as 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.

Fourth Edition Changes

This latest edition, now called *Microsoft Windows Internals, Fourth Edition*, has been updated to cover the kernel changes made in Windows XP and Windows Server 2003, including support for 64-bit systems. Hands-on experiments have been updated to reflect changes in tools, and newly added experiments use new tools not available when the third edition was written.

Since the level of kernel change from Windows 2000 to these versions was relatively small (as compared to the changes between Windows NT 4.0 and Windows 2000), the vast majority of this text applies to Windows 2000, Windows XP, and Windows Server 2003. Therefore, unless explicitly stated, everything applies to all three versions.

Hands-On Experiments

Even without access to the source code, much can be gleaned about Windows internals from available tools such as the kernel debugger. When a tool can be used to expose or demonstrate some aspect of Windows internal behavior, 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 Microsoft .NET Framework, the foundation of the next generation 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 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.

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 *windowsinternals@sysinternals.com*. Updates and corrections will be posted on the page *www.sysinternals.com/windowsinternals*.

From Microsoft Press

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

http://www.microsoft.com/learning/support

To connect directly with the Microsoft Learning Knowledge Base and enter a query regarding an issue you might have encountered, go to *http://www.microsoft.com/learning/support/search.asp*.

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 the above mail addresses. For support information regarding Microsoft Windows, go to *www.microsoft.com/windows*. You can also call Standard Support at (425) 635-7011 weekdays between 6 a.m. and 6 p.m. Pacific time, or you can search Microsoft's Support Online at *support.microsoft.com/support*.

System Requirements

To use the *Microsoft Windows Server 2003 Resource Kit* tools, eBooks, and other materials, you need to meet the following minimum system requirements:

- Microsoft Windows Server 2003 or Windows XP operating system
- PC with 233-megahertz (MHz) or higher processor; 550-MHz or higher processor is recommended
- 128 MB of RAM; 256 MB or higher is recommended
- 1.5 to 2 GB of available hard disk space
- Super VGA (800 x 600) or higher resolution video adapter and monitor
- CD or DVD drive
- Keyboard and Microsoft mouse or compatible pointing device
- Adobe Acrobat or Adobe Reader
- Internet connectivity for tools that are downloaded

Note Resource Kit tools are written and tested in English only. Using these tools with a non-English version of Windows might produce unpredictable results. Resource Kit tools are not supported on 64-bit platforms.

An evaluation edition for Windows Server 2003 Enterprise Edition with Service Pack 1 will be available on release of Service Pack 1. You can download the evaluation software from the Microsoft Download Center at *http://www.microsoft.com/downloads/*. (Availability of software on the Download Center is at the discretion of Microsoft Corporation and is subject to change.) To use the evaluation, you need

- 133-MHz or higher processor; 733-MHz or higher processor is recommended for x86-based PCs and Itanium-based PCs.
- 128 MB of RAM; 256 MB of RAM is recommended; 32 GB is recommended for x86-based PCs (32-bit version) and 64 GB is recommended for Itanium-based PCs (64-bit version).
- 1.5 to 2 GB of available hard disk space.
- Super VGA (500 x 600) or higher resolution video adapter and monitor.
- Keyboard and Microsoft mouse or compatible pointing device.

Note Actual requirements, including Internet and network access and any related charges, will vary based on your system configuration and the applications and features you choose to install. Additional hard disk space may be required if you are installing over a network.

Chapter 1

Concepts and Tools

In this chapter, we'll introduce the key Microsoft Windows operating system concepts and terms we'll be using throughout this book, such as the Windows API, processes, threads, virtual memory, kernel mode and user mode, objects, handles, security, and the registry. We'll also introduce the tools that you can use to explore Windows internals, such as the kernel debugger, the Performance tool, and key tools from *www.sysinternals.com*. In addition, we'll explain how you can use the Windows Device Driver Kit (DDK) and Platform Software Development Kit (SDK) as resources for finding further information on Windows internals.

Be sure that you understand everything in this chapter—the remainder of the book is written assuming that you do.

Windows Operating System Versions

This book covers the three most recent versions of the Microsoft Windows operating system based on the Windows NT code base: Windows 2000, Windows XP (32-bit and 64-bit versions), and Windows Server 2003 (32-bit and 64-bit versions). Unless specifically stated, the text applies to all three versions. As background information, Table 1-1 lists the releases of the Windows NT code base, their internal version number, and the external product name.

Table 1-1 Windows Operating System Releases

Product Name	Internal Version Number	Release Date
Windows NT 3.1	3.1	July 1993
Windows NT 3.5	3.5	September 1994
Windows NT 3.51	3.51	May 1995
Windows NT 4.0	4.0	July 1996
Windows 2000	5.0	December 1999
Windows XP	5.1	August 2001
Windows Server 2003	5.2	March 2003

Windows NT vs. Windows 95

From the initial announcement of Windows NT, Microsoft made it clear that it was to be the long-term replacement for Windows 95 (and its subsequent releases, Windows 98 and Windows Millennium Edition). The following list highlights some architectural differences and advantages that Windows NT (and its subsequent releases) has over Windows 95 (and its subsequent releases):

- Windows NT supports multiprocessor systems—Windows 95 doesn't.
- The Windows NT file system supports security (such as discretionary access control). The Windows 95 file system doesn't.
- Windows NT is fully a 32-bit (and now 64-bit) operating system—it contains no 16-bit code, other than support code for running 16-bit Windows applications. Windows 95 contains a large amount of old 16-bit code from its predecessors, Windows 3.1 and MS-DOS.
- Windows NT is fully reentrant—significant parts of Windows 95 are nonreentrant (mainly the 16-bit code taken from Windows 3.1). This nonreentrant code includes the majority of the graphics and window management functions (GDI and USER). When a 32-bit application on Windows 95 attempts to call a system service implemented in nonreentrant 16-bit code, the application must first obtain a system-wide lock (or mutex) to block other threads from entering the nonreentrant code base. And even worse, a 16-bit application holds this lock *while running*. As a result, although the core of Windows 95 contains a preemptive 32-bit multithreaded scheduler, applications often run single threaded because so much of the system is still implemented in nonreentrant code.
- Windows NT provides an option to run 16-bit Windows applications in their own address space—Windows 95 always runs 16-bit Windows applications in a shared address space, in which they can corrupt (and hang) each other.
- Process shared memory on Windows NT is visible only to the processes that are mapping the same shared memory section. On Windows 95, all shared memory is visible and writable from all processes. Thus, any process can write to and corrupt shared memory being used by other cooperating processes.
- Windows 95 has some critical operating system pages that are writable from user mode, thus allowing a user application to corrupt or crash the system.

The one thing Windows 95 can do that Windows NT–based systems will never do is run *all* older MS-DOS and Windows 3.1 applications (notably applications that require direct hardware access) as well as 16-bit MS-DOS device drivers. Whereas 100 percent compatibility with MS-DOS and Windows 3.1 was a mandatory goal for Windows 95, the original goal for Windows NT was to run *most* existing 16-bit applications while preserving the integrity and reliability of the system.

Foundation Concepts and Terms

In the course of this book, we'll be referring to some structures and concepts that might be unfamiliar to some readers. In this section, we'll define the terms we'll be using throughout. You should become familiar with them before proceeding to subsequent chapters.

Windows API

The Windows application programming interface (API) is the system programming interface to the Microsoft Windows operating system family, including Windows 2000, Windows XP, Windows Server 2003, Windows 95, Windows 98, Windows Millennium Edition (Me), and Windows CE. Each operating system implements a different subset of the Windows API. Windows 95, Windows 98, Windows Me, and Windows CE are not addressed in this book.

Note The Windows API is described in the Platform Software Development Kit (SDK) documentation. (See the section "Platform Software Development Kit (SDK)" later in this chapter.) This documentation is available for free viewing online at *msdn.microsoft.com*. It is also included with all subscription levels to the Microsoft Developer Network (MSDN), Microsoft's support program for developers. For more information, see *msdn.microsoft.com*. An excellent description of how to program the Windows base API is Jeffrey Richter's book *Programming Applications for Microsoft Windows* (4th ed., Microsoft Press, 1999).

Prior to the introduction of 64-bit versions of Windows XP and Windows Server 2003, the programming interface to the 32-bit version of the Windows operating systems was called the Win32 API, to distinguish it from the original 16-bit Windows API, which was the programming interface to the original 16-bit versions of Windows. In this book, the term *Windows API* refers to the 32-bit interface to Windows 2000 and both the 32-bit and 64-bit programming interfaces to Windows XP and Windows Server 2003.

The Windows API consists of thousands of callable functions, which are divided into the following major categories:

- Base Services
- Component Services
- User Interface Services
- Graphics and Multimedia Services
- Messaging and Collaboration
- Networking
- Web Services

This book focuses on the internals of the key base services, such as processes and threads, memory management, I/O, and security.

What About .NET and WinFX?

The .NET Framework consists of a library of classes called the Framework Class Library (FCL) and a Common Language Runtime (CLR) that provides a managed code execution environment with features such as just-in-time compilation, type verification, garbage collection, and code access security. By offering these features, the CLR provides a development environment that improves programmer productivity and reduces common programming errors. (For an excellent description of the .NET Framework and its core architecture, see *Applied Microsoft .NET Framework Programming* by Jeffrey Richter.)

The CLR is implemented as a classic COM server whose code resides in a standard user-mode Windows DLL. In fact, all components of the .NET Framework are implemented as standard user-mode Windows DLLs layered over unmanaged Windows API functions. (None of the .NET Framework runs in kernel mode.) Figure 1-1 illustrates the relationship of these components:

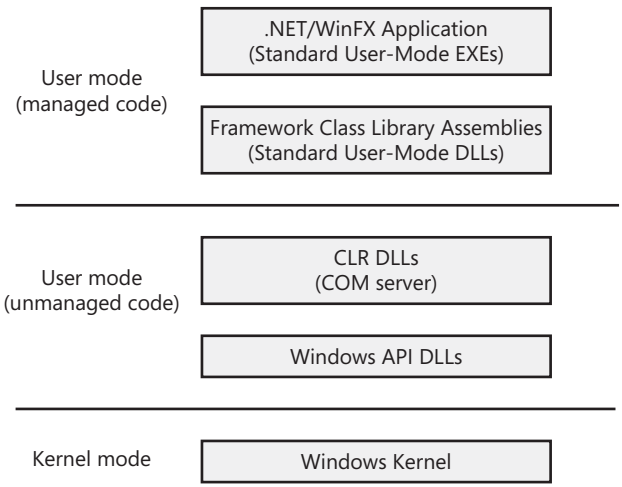


Figure 1-1 Relationship of .NET Framework components

WinFX is "the new Windows API." It is the evolution of the .NET Framework that ships with Windows "Longhorn," the next major release of Windows. It will also be installable on Windows XP and Windows Server 2003. WinFX provides the foundation for the next generation of applications built for the Windows operating system.

History of the Win32 API

Interestingly, Win32 wasn't slated to be the original programming interface to Microsoft Windows NT. Because the Windows NT project started as a replacement for OS/2 version 2, the primary programming interface was the 32-bit OS/2 Presentation Manager API. A year into the project, however, Microsoft Windows 3.0 hit the market and took off. As a result, Microsoft changed direction and made Windows NT the future replacement for the Windows family of products as opposed to the replacement for OS/2. It was at this juncture that the need to specify the Windows API arose—before this, the Windows API existed only as a 16-bit interface.

Although the Windows API would introduce many new functions that hadn't been available on Windows 3.1, Microsoft decided to make the new API compatible with the 16-bit Windows API function names, semantics, and use of data types whenever possible to ease the burden of porting existing 16-bit Windows applications to Windows NT. So those of you who are looking at the Windows API for the first time and wondering why many function names and interfaces seem inconsistent should keep in mind that one reason for the inconsistency was to ensure that the Windows API is compatible with the old 16-bit Windows API.

Services, Functions, and Routines

Several terms in the Windows user and programming documentation have different meanings in different contexts. For example, the word *service* can refer to a callable routine in the operating system, a device driver, or a server process. The following list describes what certain terms mean in this book:

- **Windows API functions** Documented, callable subroutines in the Windows API. Examples include *CreateProcess*, *CreateFile*, and *GetMessage*.
- **Native system services (or executive system services)** The undocumented, underlying services in the operating system that are callable from user mode. For example, *NtCreateProcess* is the internal system service the Windows *CreateProcess* function calls to create a new process. (For a definition of native functions, see the section "System Service Dispatching" in Chapter 3.)
- **Kernel support functions (or routines)** Subroutines inside the Windows operating system that can be called only from kernel mode (defined later in this chapter). For example, *ExAllocatePool* is the routine that device drivers call to allocate memory from the Windows system heaps.

- **Windows services** Processes started by the Windows service control manager. (Although the registry defines Windows device drivers as "services," we don't refer to them as such in this book.) For example, the Task Scheduler service runs in a user-mode process that supports the *at* command (which is similar to the UNIX commands *at* or *cron*).
- **DLL (dynamic-link library)** A set of callable subroutines linked together as a binary file that can be dynamically loaded by applications that use the subroutines. Examples include Msvcrt.dll (the C run-time library) and Kernel32.dll (one of the Windows API subsystem libraries). Windows user-mode components and applications use DLLs extensively. The advantage DLLs provide over static libraries is that applications can share DLLs, and Windows ensures that there is only one in-memory copy of a DLL's code among the applications that are referencing it.

Processes, Threads, and Jobs

Although programs and processes appear similar on the surface, they are fundamentally different. A *program* is a static sequence of instructions, whereas a *process* is a container for a set of resources used when executing the instance of the program. At the highest level of abstraction, a Windows process comprises the following:

- A *private virtual address space*, which is a set of virtual memory addresses that the process can use
- An executable program, which defines initial code and data and is mapped into the process's virtual address space
- A list of open handles to various system resources, such as semaphores, communication ports, and files, that are accessible to all threads in the process
- A security context called an *access token* that identifies the user, security groups, and privileges associated with the process
- A unique identifier called a *process ID* (internally called a *client ID*)
- At least one thread of execution

Each process also points to its parent or creator process. However, if the parent exits, this information is not updated. Therefore, it is possible for a process to point to a nonexistent parent. This is not a problem, as nothing relies on this information being present. The following experiment illustrates this case.

EXPERIMENT: Viewing the Process Tree

One unique attribute about a process that most tools don't display is the parent or creator process ID. You can retrieve this value with the Performance tool (or programmatically) by querying the Creating Process ID. The Tlist.exe tool (in the Windows Debugging Tools) can show the process tree by using */t* switch. Here's an example of output from *tlist /t*:

```
C:\>tlist /t
System Process (0)
System (2)
  smss.exe (21)
    csrss.exe (24)
    winlogon.exe (35)
      services.exe (41)
        spoolss.exe (69)
        llssrv.exe (94)
        LOCATOR.EXE (96)
        RpcSs.exe (112)
        inetinfo.exe (128)
      lsass.exe (44)
   nddeagnt.exe (119)
explorer.exe (123) Program Manager
  OSA.EXE (121)
  WINWORD.EXE (117) Microsoft Word - msch02(s).doc
  cmd.exe (72) Command Prompt - tlist /t
    tlist.EXE (100)
```

The list indents each process to show its parent/child relationship. Processes whose parents aren't alive are left-justified (as is Explorer.exe in the preceding example) because even if a grandparent process exists, there's no way to find that relationship. Windows maintains only the creator process ID, not a link back to the creator of the creator, and so forth.

To demonstrate the fact that Windows doesn't keep track of more than just the parent process ID, follow these steps:

1. Open a Command Prompt window.
2. Type **start cmd** (which starts a second Command Prompt).
3. Bring up Task Manager.
4. Switch to the second Command Prompt.
5. Type **mspaint** (which runs Microsoft Paint).

6. Click the intermediate (second) Command Prompt window.
7. Type **exit**. (Notice that Paint remains.)
8. Switch to Task Manager.
9. Click the Applications tab.
10. Right-click on the Command Prompt task, and select Go To Process.
11. Click on the Cmd.exe process highlighted in gray.
12. Right-click on this process, and select End Process Tree.
13. Click Yes in the Task Manager Warning message box.

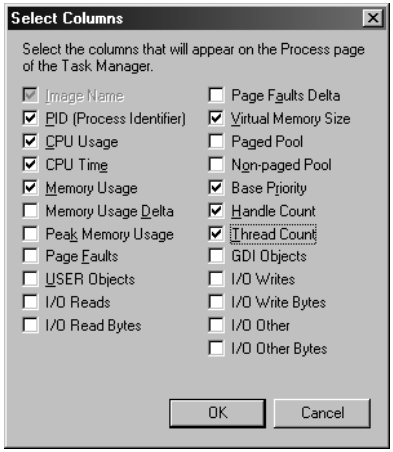
The first Command Prompt window will disappear, but you should still see the Paintbrush window because it was the grandchild of the Command Prompt process you terminated; and because the intermediate process (the parent of Paintbrush) was terminated, there was no link between the parent and the grandchild.

A number of tools for viewing (and modifying) processes and process information are available. The following experiments illustrate the various views of process information you can obtain with some of these tools. These tools are included within Windows itself and within the Windows Support Tools, Windows Debugging Tools, Windows resource kits, the Platform SDK, and from *www.sysinternals.com*. Many of these tools show overlapping subsets of the core process and thread information, sometimes identified by different names.

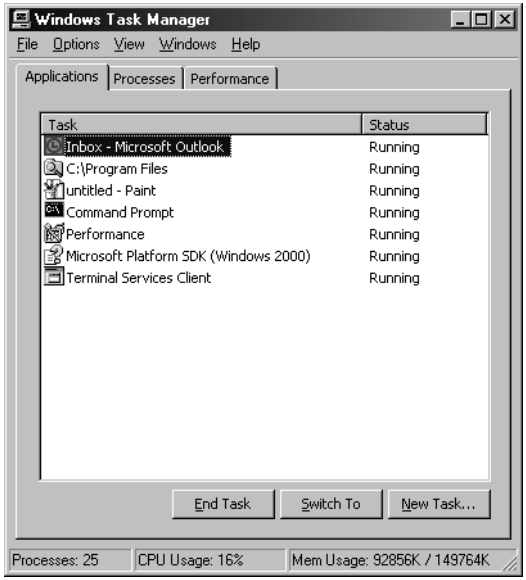
Probably the most widely used tool to examine process activity is Task Manager. (Interestingly, there is no such thing as a "task" in the Windows kernel, so Task Manager is really a tool to manage processes.) The following experiment shows the difference between what Task Manager lists as applications and processes.

EXPERIMENT: Viewing Process Information with Task Manager

The built-in Windows Task Manager provides a quick list of the processes running on the system. You can start Task Manager in one of three ways: (1) press Ctrl+Shift+Esc, (2) right-click on the taskbar and select Task Manager, or (3) press Ctrl+Alt+Delete and click the Task Manager button. Once Task Manager has started, click the Processes tab to see the list of running processes. Notice that processes are identified by the name of the image of which they are an instance. Unlike some objects in Windows, processes can't be given global names. To display additional details, choose Select Columns from the View menu and select additional columns to be added, as shown here:



Although what you see in the Task Manager Processes tab is clearly a list of processes, what the Applications tab displays isn't as obvious. The Applications tab lists the top-level visible windows on all the desktops in the interactive window station. (By default, there are two desktop objects—you can create more by using the Windows *CreateDesktop* function.) The Status column indicates whether or not the thread that owns the window is in a Windows message wait state. "Running" means the thread is waiting for windowing input; "Not Responding" means the thread isn't waiting for windowing input (for example, the thread might be running or waiting for I/O or some Windows synchronization object).



From the Applications tab, you can match a task to the process that owns the thread that owns the task window by right-clicking on the task name and choosing Go To Process.

Process Explorer, from *www.sysinternals.com*, shows more details about processes and threads than any other available tool, which is why you will see it used in a number of experiments throughout the book. The following are some of the unique things that Process Explorer shows or enables:

- Full path name for the image being executed
- Process security token (list of groups and privileges)
- Highlighting to show changes in the process and thread list
- List of services inside service-hosting processes, including display name and description
- Processes that are part of a job and job details
- Processes running .NET/WinFX applications and.NET-specific details (such as the list of appdomains and CLR performance counters)
- Start time for processes and threads
- Complete list of memory mapped files (not just DLLs)
- Ability to suspend a process
- Ability to kill an individual thread
- Easy identification of which processes were consuming the most CPU time over a period of time (The Performance Tool can display process CPU utilization for a given set of processes, but it won't automatically show processes created after the performance monitoring session has started.)

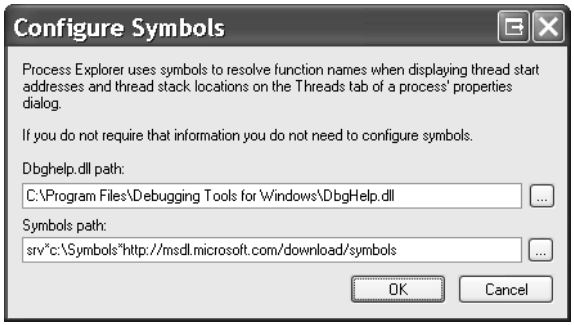
Process Explorer also provides easy access to information available through other tools from one central place, such as:

- Process tree (with ability to collapse parts of the tree)
- Open handles in a process without prior setup (The Microsoft tools to show open handles require the setting of a systemwide flag and a reboot before they can be used.)
- List of DLLs (and memory-mapped files) in a process
- Thread activity within a process
- User-mode thread stacks (including mapping of addresses to names using the debugging tools' symbol engine)
- Kernel-mode thread stacks for system threads (including mapping of addresses to names using the debugging tools' symbol engine)
- Context switch delta (a better representation of CPU activity, as explained in Chapter 6)
- Kernel memory (paged and nonpaged pool) limits (other tools show only current size)

An introductory experiment using Process Explorer follows.

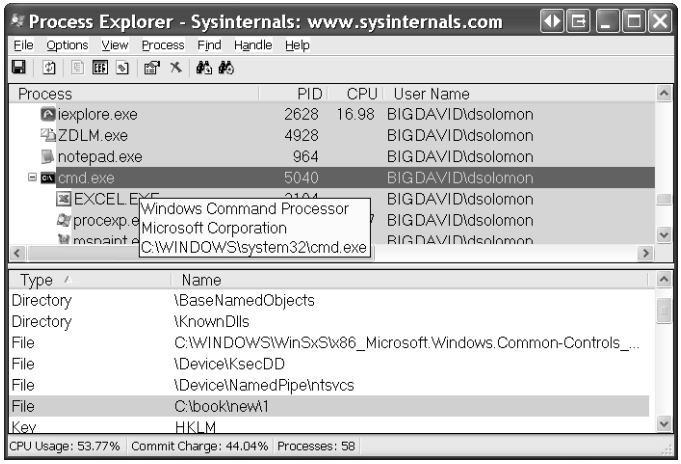
EXPERIMENT: Viewing Process Details with Process Explorer

Download the latest version of Process Explorer from *www.sysinternals.com* and run it. The first time you run it, you will receive a message that symbols are not currently configured. If properly configured, Process Explorer can access symbol information to display the symbolic name of the thread start function and functions on its call stack (available by double-clicking on a process and clicking on the Threads tab). This is useful for identifying what threads are doing within a process. To access symbols, you must have the Debugging Tools installed (described later in this chapter). Then click on Options, choose Configure Symbols, and fill in the appropriate Symbols path. For example:



In the preceding example, the on-demand symbol server is being used to access symbols and a copy of the symbol files are being stored on the local machine in the c:\symbols folder. For more information on configuring use of the symbol server, see *http://www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

When Process Explorer starts, it shows by default the process list on the top half and the open handles for the currently selected process on the bottom half. It also shows the image description, company name, and full path if you hover the mouse pointer over the process name.



Here are a few steps to walk you through some basic capabilities of Process Explorer:

1. Turn off the lower pane by deselecting View, Show Lower Pane. (The lower pane can show open handles or mapped DLLs and memory mapped files—these are explored in Chapters 3 and 7.)
2. Notice that processes hosting services are highlighted by default in pink. Your own processes are highlighted in blue. (These colors can be configured.)
3. Hover your mouse pointer over the image name for processes, and notice the full path displayed by the ToolTip.
4. Click on View, Select Columns, and add the image path.
5. Sort on the process column, and notice the tree view disappears. (You can either display tree view or sort by any of the columns shown.) Click again to sort from Z to A. Then click again and the display returns to tree view.
6. Deselect View, Show Processes From All Users to show only your processes.
7. Go to Options, Difference Highlight Duration, and change the value to 5 seconds. Then launch a new process (anything), and notice the new process highlighted in green for 5 seconds. Exit this new process, and notice the process is highlighted in red for 5 seconds before disappearing from the display. This can be useful to see processes being created and exiting on your system.
8. Finally, double-click on a process and explore the various tabs available from the process properties display. (These will be referenced in various experiments throughout the book where the information being shown is being explained.)

A *thread* is the entity within a process that Windows schedules for execution. Without it, the process's program can't run. A thread includes the following essential components:

- The contents of a set of CPU registers representing the state of the processor.
- Two stacks, one for the thread to use while executing in kernel mode and one for executing in user mode.
- A private storage area called thread-local storage (TLS) for use by subsystems, run-time libraries, and DLLs.
- A unique identifier called a *thread ID* (also internally called a *client ID*—process IDs and thread IDs are generated out of the same namespace, so they never overlap).
- Threads sometimes have their own security context that is often used by multithreaded server applications that impersonate the security context of the clients that they serve.

The volatile registers, stacks, and private storage area are called the thread's *context*. Because this information is different for each machine architecture that Windows runs on, this structure, by necessity, is architecture-specific. The Windows *GetThreadContext* function provides access to this architecture-specific information (called the CONTEXT block).

Fibers vs. Threads

Fibers allow an application to schedule its own "threads" of execution rather than rely on the priority-based scheduling mechanism built into Windows. Fibers are often called "lightweight" threads, and in terms of scheduling, they're invisible to the kernel because they're implemented in user mode in Kernel32.dll. To use fibers, a call is first made to the Windows *ConvertThreadToFiber* function. This function converts the thread to a running fiber. Afterward, the newly converted fiber can create additional fibers with the *CreateFiber* function. (Each fiber can have its own set of fibers.) Unlike a thread, however, a fiber doesn't begin execution until it's manually selected through a call to the *SwitchToFiber* function. The new fiber runs until it exits or until it calls *SwitchToFiber*, again selecting another fiber to run. For more information, see the Platform SDK documentation on fiber functions.

Although threads have their own execution context, every thread within a process shares the process's virtual address space (in addition to the rest of the resources belonging to the process), meaning that all the threads in a process can write to and read from each other's memory. Threads cannot accidentally reference the address space of another process, however, unless the other process makes available part of its private address space as a *shared memory section* (called a *file mapping object* in the Windows API) or unless one process has the right to open another process to use cross-process memory functions such as *ReadProcessMemory* and *WriteProcessMemory*.

In addition to a private address space and one or more threads, each process has a security identification and a list of open handles to objects such as files, shared memory sections, or one of the synchronization objects such as mutexes, events, or semaphores, as illustrated in Figure 1-2.

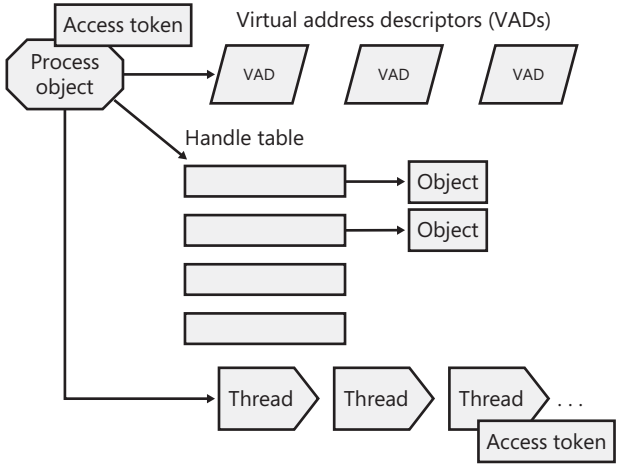


Figure 1-2 A process and its resources

Every process has a security context that is stored in an object called an *access token*. The process access token contains the security identification and credentials for the process. By default, threads don't have their own access token, but they can obtain one, thus allowing individual threads to impersonate the security context of another process—including processes running on a remote Windows system—without affecting other threads in the process. (See Chapter 8 for more details on process and thread security.)

The *virtual address descriptors* (VADs) are data structures that the memory manager uses to keep track of the virtual addresses the process is using. These data structures are described in more depth in Chapter 7.

Windows provides an extension to the process model called a *job*. A job object's main function is to allow groups of processes to be managed and manipulated as a unit. A job object allows control of certain attributes and provides limits for the process or processes associated with the job. It 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. In some ways, the job object compensates for the lack of a structured process tree in Windows—yet in many ways it is more powerful than a UNIX-style process tree.

You'll find out much more about the internal structure of jobs, processes and threads, the mechanics of process and thread creation, and the thread-scheduling algorithms in Chapter 6.

Virtual Memory

Windows implements a virtual memory system based on a flat (linear) address space that provides each process with the illusion of having its own large, private address space. Virtual memory provides a logical view of memory that might not correspond to its physical layout. At run time, the memory manager, with assistance from hardware, translates, or *maps*, the virtual addresses into physical addresses, where the data is actually stored. By controlling the protection and mapping, the operating system can ensure that individual processes don't bump into one another or overwrite operating system data. Figure 1-3 illustrates three virtually contiguous pages mapped to three discontiguous pages in physical memory.

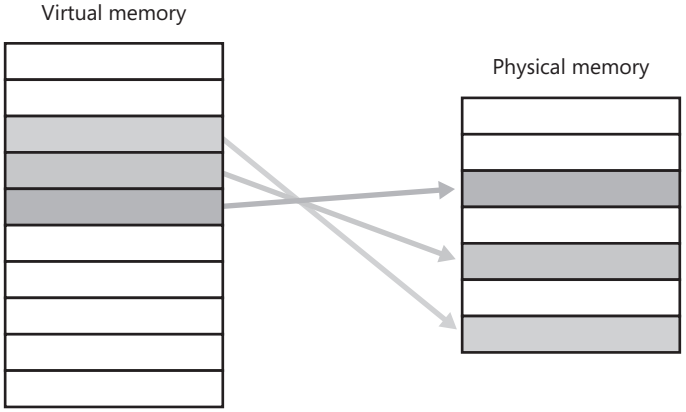


Figure 1-3 Mapping virtual memory to physical memory

Because most systems have much less physical memory than the total virtual memory in use by the running processes, the memory manager transfers, or *pages*, some of the memory contents to disk. Paging data to disk frees physical memory so that it can be used for other processes or for the operating system itself. When a thread accesses a virtual address that has been paged to disk, the virtual memory manager loads the information back into memory from disk. Applications don't have to be altered in any way to take advantage of paging because hardware support enables the memory manager to page without the knowledge or assistance of processes or threads.

The size of the virtual address space varies for each hardware platform. On 32-bit x86 systems, the total virtual address space has a theoretical maximum of 4 GB. By default, Windows allocates half this address space (the lower half of the 4 GB virtual address space, from x00000000 through x7FFFFFFF) to processes for their unique private storage and uses the other half (the upper half, addresses x80000000 through xFFFFFFFF) for its own protected operating system memory utilization. The mappings of the lower half change to reflect the virtual address space of the currently executing process, but the mappings of the upper half always consist of the operating system's virtual memory. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP (SP2 and later), and Windows Server 2003 support boot-time options (the /3GB and /USERVA qualifiers in Boot.ini, described in Chapter 5) that give processes running specially marked programs (the large address space aware flag must be set in the header of the executable image) the ability to use up to 3 GB of private address space (leaving 1 GB for the operating system). This option allows applications such as database servers to keep larger portions of a database in the process address space, thus reducing the need to map subset views of the database. Figure 1-4 shows the two virtual address space layouts supported by 32-bit Windows.

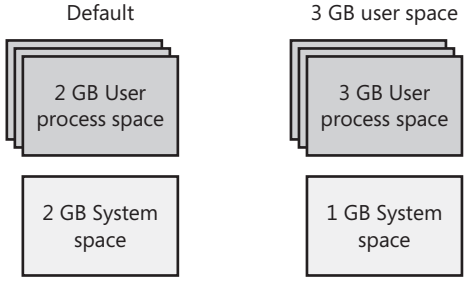


Figure 1-4 Address space layouts for 32-bit Windows

Although 3 GB is better than 2 GB, it's still not enough virtual address space to map very large (multigigabyte) databases. To address this need on 32-bit systems, Windows provides a mechanism called *Address Windowing Extension* (AWE), which allows a 32-bit application to allocate up to 64 GB of physical memory and then map views, or windows, into its 2-GB virtual address space. Although using AWE puts the burden of managing mappings of virtual to

physical memory on the programmer, it does address the need of being able to directly access more physical memory than can be mapped at any one time in a 32-bit process address space.

64-bit Windows provides a much larger address space for processes: 7152 GB on Itanium systems and 8192 GB on x64 systems. Figure 1-5 shows a simplified view of the 64-bit system address space layouts. (For a detailed description, see Chapter 7.) Note that these sizes do not represent the architectural limits for these platforms, but rather implementation limits in the current versions of 64-bit Windows.

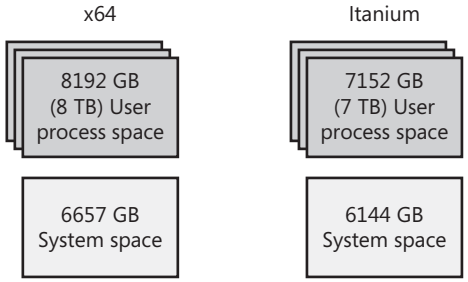


Figure 1-5 Address space layouts for 64-bit Windows

Details of the implementation of the memory manager, including how address translation works and how Windows manages physical memory, are described in Chapter 7.

Kernel Mode vs. User Mode

To protect user applications from accessing and/or modifying critical operating system data, Windows uses two *processor access modes* (even if the processor on which Windows is running supports more than two): *user mode* and *kernel mode*. User application code runs in user mode, whereas operating system code (such as system services and device drivers) runs in kernel mode. Kernel mode refers to a mode of execution in a processor that grants access to all system memory and all CPU instructions. By providing the operating system software with a higher privilege level than the application software has, the processor provides a necessary foundation for operating system designers to ensure that a misbehaving application can't disrupt the stability of the system as a whole.

Note The architecture of the Intel x86 processor defines four privilege levels, or rings, to protect system code and data from being overwritten either inadvertently or maliciously by code of lesser privilege. Windows uses privilege level 0 (or ring 0) for kernel mode and privilege level 3 (or ring 3) for user mode. The reason Windows uses only two levels is that some hardware architectures that were supported in the past (such as Compaq Alpha and Silicon Graphics MIPS) implemented only two privilege levels.

Although each Windows process has its own private memory space, the kernel-mode operating system and device driver code share a single virtual address space. Each page in virtual memory is tagged as to what access mode the processor must be in to read and/or write the page. Pages in system space can be accessed only from kernel mode, whereas all pages in the user address space are accessible from user mode. Read-only pages (such as those that contain executable code) are not writable from any mode.

Windows doesn't provide any protection to private read/write system memory being used by components running in kernel mode. In other words, once in kernel mode, operating system and device driver code has complete access to system space memory and can bypass Windows security to access objects. Because the bulk of the Windows operating system code runs in kernel mode, it is vital that components that run in kernel mode be carefully designed and tested to ensure that they don't violate system security.

This lack of protection also emphasizes the need to take care when loading a third-party device driver, because once in kernel mode the software has complete access to all operating system data. This vulnerability was one of the reasons behind the driver-signing mechanism introduced in Windows, which warns the user if an attempt is made to add an unauthorized (unsigned) driver. (See Chapter 9 for more information on driver signing.) Also, a mechanism called Driver Verifier helps device driver writers to find bugs (such as buffer overruns or memory leaks). Driver Verifier is also explained in Chapter 7.

As you'll see in Chapter 2, user applications switch from user mode to kernel mode when they make a system service call. For example, a Windows *ReadFile* function eventually needs to call the internal Windows routine that actually handles reading data from a file. That routine, because it accesses internal system data structures, must run in kernel mode. The transition from user mode to kernel mode is accomplished by the use of a special processor instruction that causes the processor to switch to kernel mode. The operating system traps this instruction, notices that a system service is being requested, validates the arguments the thread passed to the system function, and then executes the internal function. Before returning control to the user thread, the processor mode is switched back to user mode. In this way, the operating system protects itself and its data from perusal and modification by user processes.

Note A transition from user mode to kernel mode (and back) does *not* affect thread scheduling per se—a mode transition is *not* a context switch. Further details on system service dispatching are included in Chapter 3.

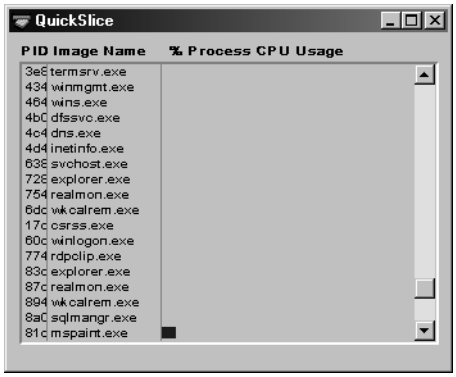
Thus, it's normal for a user thread to spend part of its time executing in user mode and part in kernel mode. In fact, because the bulk of the graphics and windowing system also runs in kernel mode, graphics-intensive applications spend more of their time in kernel mode than in user mode. An easy way to test this is to run a graphics-intensive application such as Microsoft Paint or Microsoft Pinball and watch the time split between user mode and kernel mode using one of the performance counters listed in Table 1-2.

Table 1-2 Mode-Related Performance Counters

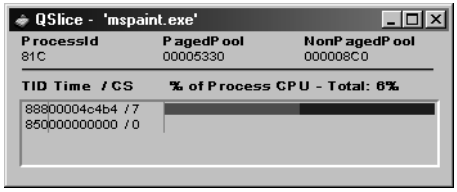
Object: Counter	Function
Processor: % Privileged Time	Percentage of time that an individual CPU (or all CPUs) has run in kernel mode during a specified interval
Processor: % User Time	Percentage of time that an individual CPU (or all CPUs) has run in user mode during a specified interval
Process: % Privileged Time	Percentage of time that the threads in a process have run in kernel mode during a specified interval
Process: % User Time	Percentage of time that the threads in a process have run in user mode during a specified interval
Thread: % Privileged Time	Percentage of time that a thread has run in kernel mode during a specified interval
Thread: % User Time	Percentage of time that a thread has run in user mode during a specified interval

EXPERIMENT: Viewing Thread Activity with QuickSlice

QuickSlice gives a quick, dynamic view of the proportions of system and kernel time that each process currently running on your system is using. On line, the red part of the bar shows the amount of CPU time spent in kernel mode, and the blue part shows the user-mode time. (Although reproduced in the window below in black and white, the bars in the online display are always red and blue.) The total of all bars shown in the QuickSlice window should add up to 100 percent of CPU time. To run QuickSlice, click the Start button, choose Run, and enter **Qslice.exe** (assuming the Windows 2000 resource kit is in your path). For example, try running a graphics-intensive application such as Paint (Mspaint.exe). Open QuickSlice and Paint side by side, and draw squiggles in the Paint window. When you do so, you'll see Mspaint.exe running in the QuickSlice window, as shown here:



For additional information about the threads in a process, you can also double-click on a process (on either the process name or the colored bar). Here you can see the threads within the process and the relative CPU time each thread uses (not across the system):



EXPERIMENT: Kernel Mode vs. User Mode

You can use the Performance tool to see how much time your system spends executing in kernel mode vs. in user mode. Follow these steps:

1. Run the Performance tool by opening the Start menu and selecting Programs/Administrative Tools/Performance.
2. Click the Add button (+) on the toolbar.
3. With the Processor performance object selected, click the % Privileged Time counter and, while holding down the Ctrl key, click the % User Time counter.
4. Click Add, and then click Close.
5. Move the mouse rapidly back and forth. You should notice the % Privileged Time line going up when you move the mouse around, reflecting the time spent servicing the mouse interrupts and the time spent in the graphics part of the windowing system (which, as explained in Chapter 2, runs primarily as a device driver in kernel mode). (See Figure 1-6.)

6. When you're finished, click the New Counter Set button on the toolbar (or just close the tool).

You can also quickly see this activity by using Task Manager. Just click the Performance tab, and then select Show Kernel Times from the View menu. The CPU usage bar will show user-mode time in green and kernel-mode time in red.

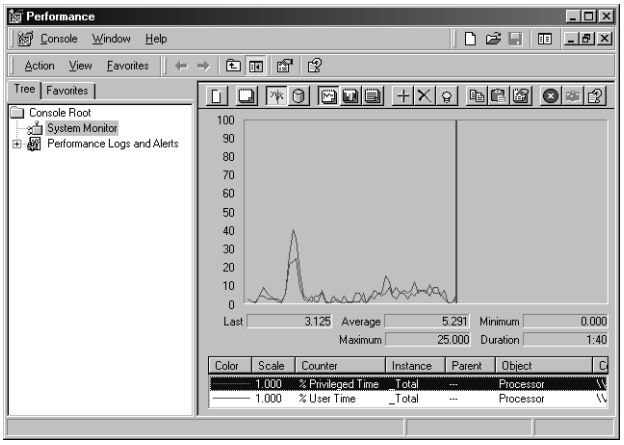


Figure 1-6 Performance tool showing time split between kernel mode and user mode

To see how the Performance tool itself uses kernel time and user time, run it again, but add the individual Process counters % User Time and % Privileged Time for every process in the system:

1. If it's not already running, run the Performance tool again. (If it is already running, start with a blank display by pressing the New Counter Set button on the toolbar.)
2. Click the Add button (+) on the toolbar.
3. Change the Performance Object to Process.
4. Select the % Privileged Time and % User Time counters.
5. Select all processes in the Instance box (except the _Total process).
6. Click Add, and then click Close.
7. Move the mouse rapidly back and forth.
8. Press Ctrl+H to turn on highlighting mode. This highlights the currently selected counter in white on Windows 2000 and black on Windows XP and Windows Server 2003.
9. Scroll through the counters at the bottom of the display to identify the processes whose threads were running when you moved the mouse, and note whether they were running in user mode or kernel mode.

You should see the Performance tool process (by looking in the Instance column for the mmc process) kernel-mode *and* user-mode time go up when you move the mouse because it is executing application code in user mode and calling Windows functions that run in kernel mode. You'll also notice kernel-mode thread activity in a process named csrss when you move the mouse. This activity occurs because the Windows subsystem's kernel-mode raw input thread, which handles keyboard and mouse input, is attached to this process. (See Chapter 2 for more information about system threads.) Finally, the process named Idle that you see spending nearly 100 percent of its time in kernel mode isn't really a process—it's a fake process used to account for idle CPU cycles. As you can observe from the mode in which the threads in the Idle process run, when Windows has nothing to do, it does it in kernel mode.

Terminal Services and Multiple Sessions

Terminal Services refers to the support in Windows for multiple interactive user sessions on a single system. With Windows Terminal Services, a remote user can establish a session on another machine, log in, and run applications on the server. The server transmits the graphical user interface to the client, and the client transmits the user's input back to the server. (This is different than X windows on UNIX systems, which permit running individual applications on a server system with the display remoted to the client, because the entire user session is remoted, not just a single application.)

The first login session at the physical console of the machine is considered the console session, or session zero. Additional sessions can be created through the use of the remote desktop connection program (Mstsc.exe) or on Windows XP systems through the use of fast user switching (described later).

The capability to create a remote session is supported on Windows 2000 Server systems but not Windows 2000 Professional. Windows XP Professional permits a single remote user to connect to the machine, but if someone is logged in at the console, the workstation is locked (that is, someone can be using the system either locally or remotely, but not at the same time).

Windows 2000 Server and Windows Server 2003 Standard Edition support two simultaneous remote connections. (This is to facilitate remote management—for example, use of management tools that require being logged in to the machine being managed.) Windows 2000 Advanced Server, Datacenter Server, Windows Server 2003 Enterprise Edition, and Datacenter Edition can support more than two sessions if appropriately licensed and configured as a terminal server.

Although Windows XP Home and Professional editions do not support multiple remote desktop connections, they do support multiple sessions created locally through a feature called fast user switching. (This feature is disabled on Windows XP Professional if the system joins a domain.) When a user chooses to disconnect their session instead of log off (for example, by

clicking Start, clicking Log Off, and choosing Switch User or by holding down the Windows key and pressing "L"), the current session (that is, the processes running in that session and all the sessionwide data structures that describe the session) remains in the system and the system returns to the main logon screen. If a new user logs in, a new session is created.

For applications that want to be aware of running in a terminal server session, there are a set of Windows APIs for programmatically detecting that as well as for controlling various aspects of terminal services. (See the Platform SDK for details.)

Chapter 2 describes briefly how sessions are created and has some experiments showing how to view session information with various tools, including the kernel debugger. The "Object Manager" section in Chapter 3 describes how the system namespace for objects is instantiated on a per-session basis and how applications that need to be aware of other instances of themselves on the same system can accomplish that. Finally, Chapter 7 covers how the memory manager sets up and manages sessionwide data.

Objects and Handles

In the Windows operating system, an *object* is a single, run-time instance of a statically defined object type. An *object type* comprises a system-defined data type, functions that operate on instances of the data type, and a set of object attributes. If you write Windows applications, you might encounter process, thread, file, and event objects, to name just a few examples. These objects are based on lower-level objects that Windows creates and manages. In Windows, a process is an instance of the process object type, a file is an instance of the file object type, and so on.

An *object attribute* is a field of data in an object that partially defines the object's state. An object of type *process*, for example, would have attributes that include the process ID, a base scheduling priority, and a pointer to an access token object. *Object methods*, the means for manipulating objects, usually read or change the object attributes. For example, the *open* method for a process would accept a process identifier as input and return a pointer to the object as output.

Note Although there is a parameter named *ObjectAttributes* that a caller supplies when creating an object using either the Windows API or native object services, that parameter shouldn't be confused with the more general meaning of the term as used in this book.

The most fundamental difference between an object and an ordinary data structure is that the internal structure of an object is hidden. You must call an object service to get data out of an object or to put data into it. You can't directly read or change data inside an object. This difference separates the underlying implementation of the object from code that merely uses it, a technique that allows object implementations to be changed easily over time.

Objects provide a convenient means for accomplishing the following four important operating system tasks:

- Providing human-readable names for system resources
- Sharing resources and data among processes
- Protecting resources from unauthorized access
- Reference tracking, which allows the system to know when an object is no longer in use so that it can be automatically deallocated

Not all data structures in the Windows operating system are objects. Only data that needs to be shared, protected, named, or made visible to user-mode programs (via system services) is placed in objects. Structures used by only one component of the operating system to implement internal functions are not objects. Objects and handles (references to an instance of an object) are discussed in more detail in Chapter 3.

Security

Windows was designed from the start to be secure and to meet the requirements of various formal government and industry security ratings, such as the Common Criteria for Information Technology Security Evaluation (CCITSE) specification. Achieving a government-approved security rating allows an operating system to compete in that arena. Of course, many of these required capabilities are advantageous features for any multiuser system.

The core security capabilities of Windows include: discretionary (need-to-know) protection for all shareable system objects (such as files, directories, processes, threads, and so forth), security auditing (for accountability of subjects, or users and the actions they initiate), password authentication at logon, and the prevention of one user from accessing uninitialized resources (such as free memory or disk space) that another user has deallocated.

Windows has two forms of access control over objects. The first form—discretionary access control—is the protection mechanism that most people think of when they think of operating system security. It's the method by which owners of objects (such as files or printers) grant or deny access to others. When users log in, they are given a set of security credentials, or a security context. When they attempt to access objects, their security context is compared to the access control list on the object they are trying to access to determine whether they have permission to perform the requested operation.

Privileged access control is necessary for those times when discretionary access control isn't enough. It's a method of ensuring that someone can get to protected objects if the owner isn't available. For example, if an employee leaves a company, the administrator needs a way to gain access to files that might have been accessible only to that employee. In that case, under Windows, the administrator can take ownership of the file so that you can manage its rights as necessary.

Security pervades the interface of the Windows API. The Windows subsystem implements object-based security in the same way the operating system does; the Windows subsystem protects shared Windows objects from unauthorized access by placing Windows security descriptors on them. The first time an application tries to access a shared object, the Windows subsystem verifies the application's right to do so. If the security check succeeds, the Windows subsystem allows the application to proceed.

The Windows subsystem implements object security on a number of shared objects, some of which were built on top of native Windows objects. The Windows objects include desktop objects, window objects, menu objects, files, processes, threads, and several synchronization objects.

For a comprehensive description of Windows security, see Chapter 8.

Registry

If you've worked at all with Windows operating systems, you've probably heard about or looked at the registry. You can't talk much about Windows internals without referring to the registry because it's the system database that contains the information required to boot and configure the system, systemwide software settings that control the operation of Windows , the security database, and per-user configuration settings (such as which screen saver to use).

In addition, the registry is a window into in-memory volatile data, such as the current hardware state of the system (what device drivers are loaded, the resources they are using, and so on) as well as the Windows performance counters. The performance counters, which aren't actually "in" the registry, are accessed through the registry functions. See Chapter 4 for more on how performance counter information is accessed from the registry.

Although many Windows users and administrators will never need to look directly into the registry (because you can view or change most configuration settings with standard administrative utilities), it is still a useful source of Windows internals information because it contains many settings that affect system performance and behavior. (If you decide to directly change registry settings, you must exercise extreme caution; any changes might adversely affect system performance or, worse, cause the system to fail to boot successfully.) You'll find references to individual registry keys throughout this book as they pertain to the component being described. Most registry keys referred to in this book are under HKEY_LOCAL_MACHINE, which we'll abbreviate throughout as HKLM.

For further information on the registry and its internal structure, see Chapter 4.

Unicode

Windows differs from most other operating systems in that most internal text strings are stored and processed as 16-bit-wide Unicode characters. Unicode is an international character set standard that defines unique 16-bit values for most of the world's known character sets. (For more information about Unicode, see *www.unicode.org* as well as the programming documentation in the MSDN Library.)

Because many applications deal with 8-bit (single-byte) ANSI character strings, Windows functions that accept string parameters have two entry points: a Unicode (wide, 16-bit) and an ANSI (narrow, 8-bit) version. The Windows 95, Windows 98, and Windows Millennium Edition implementations of Windows don't implement all the Unicode interfaces to all the Windows functions, so applications designed to run on one of these operating systems as well as Windows typically use the narrow versions. If you call the narrow version of a Windows function, input string parameters are converted to Unicode before being processed by the system and output parameters are converted from Unicode to ANSI before being returned to the application. Thus, if you have an older service or piece of code that you need to run on Windows but this code is written using ANSI character text strings, Windows will convert the ANSI characters into Unicode for its own use. However, Windows never converts the *data* inside files—it's up to the application to decide whether to store data as Unicode or as ANSI.

In previous editions of Windows, Asian and Middle East editions were a superset of the core U.S. and European editions and contained additional Windows functions to handle more complex text input and layout requirements (such as right-to-left text input). As of Windows 2000, all language editions contain the same Windows functions. Instead of having separate language versions, Windows has a single worldwide binary so that a single installation can support multiple languages (by adding various language packs). Applications can also take advantage of Windows functions that allow single worldwide application binaries that can support multiple languages.

Digging into Windows Internals

Although much of the information in this book is based on reading the Windows source code and talking to the developers, you don't have to take *everything* on faith. Many details about the internals of Windows can be exposed and demonstrated by using a variety of available tools, such as those that come with Windows, the Windows Support Tools, the Windows resource kit tools, and the Windows debugging tools. These tool packages are briefly described later in this section.

To encourage your exploration of Windows internals, we've included "Experiment" sidebars throughout the book that describe steps you can take to examine a particular aspect of Windows internal behavior. (You already saw one of these sections earlier in this chapter.) We encourage you to try these experiments so that you can see in action many of the internals topics described in this book.

Table 1-3 shows a list of the tools used in this book and where they come from.

Table 1-3 Tools for Viewing Windows Internals

Tool	Image Name	Origin
Startup Programs Viewer	AUTORUNS	*www.sysinternals.com*
Dependency Walker	DEPENDS	Support Tools, Platform SDK
DLL List	LISTDLLS	*www.sysinternals.com*
EFS Information Dumper	EFSDUMP	*www.sysinternals.com**
File Monitor	FILEMON	*www.sysinternals.com*
Global Flags	GFLAGS	Support Tools
Handle Viewer	HANDLE	*www.sysinternals.com*
Junction tool	JUNCTION	*www.sysinternals.com*
Kernel debuggers	WINDBG, KD	Debugging tools, Platform SDK, Windows DDK
Live Kernel Debugging	LIVEKD	*www.sysinternals.com*
Logon Sessions	LOGINSESSIONS	*www.sysinternals.com*
Object Viewer	WINOBJ	*www.sysinternals.com*
Open Handles	OH	Resource kits
Page Fault Monitor	PFMON	Support Tools, Resource kits, Platform SDK
Pending File Moves	PENDMOVES	*www.sysinternals.com*
Performance tool	PERFMON.MSC	Windows built-in tool
PipeList tool	PIPELIST	*www.sysinternals.com*
Pool Monitor	POOLMON	Support Tools, Windows DDK
Process Explorer	PROCEXP	*www.sysinternals.com*
Get SID tool	PSGETSID	*www.sysinternals.com*
Process Statistics	PSTAT	Support Tools, Windows 2000 Resource kits, Platform SDK, *www.reskit.com*
Process Viewer	PVIEWER (in the Support Tools) or PVIEW (in the Platform SDK)	Platform SDK
Quick Slice	QSLICE	Windows 2000 resource kits
Registry Monitor	REGMON	*www.sysinternals.com*
Service Control	SC	Windows XP, Platform SDK, Windows 2000 resource kits
Task (Process) List	TLIST	Debugging tools
Task Manager	TASKMGR	Windows built-in tool
TDImon	TDIMON	*www.sysinternals.com*

Performance Tool

We'll refer to the Performance tool found in the Administrative Tools folder on the Start menu (or via Control Panel) throughout this book. The Performance tool has three functions: system monitoring, viewing performance counter logs, and setting alerts. For simplicity, when we refer to the Performance tool, we are referring to the System Monitor function within the tool.

The Performance tool can provide more information about how your system is operating than any other single utility. It includes hundreds of counters for various objects. For each major topic described in this book, a table of the relevant Windows performance counters is included.

The Performance tool contains a brief description for each counter. To see the descriptions, select a counter in the Add Counter window and click the Explain button. Or open the Performance Counter Reference help file in the resource kit. For information on how to interpret these counters to detect bottlenecks or plan capacity, see the section "Performance Monitoring" in the *Windows 2000 Server Operations Guide*, which is part of the Windows 2000 Server Resource Kit. These chapters provide an excellent description to anyone seriously interested in understanding Windows performance. For Windows XP and Windows Server 2003 , see the Windows Server 2003 Resource Kit Performance Counters Reference documentation (available online at www.microsoft.com).

Note that all the Windows performance counters are accessible programmatically. The section "HKEY_PERFORMANCE_DATA" in Chapter 4 has a brief description of the components involved in retrieving performance counters through the Windows API.

Windows Support Tools

The Windows Support Tools consist of about 40 tools useful in administering and troubleshooting Windows systems. Many of these tools were formerly part of the Windows NT 4 resource kits.

You can install the Support Tools by running Setup.exe in the \Support\Tools folder on any Windows product distribution media. For Windows 2000, the Support Tools are the same on Windows 2000 Professional, Server, Advanced Server, and Datacenter Server. Windows XP has its own version of the Support Tools, as does Windows Server 2003.

Windows Resource Kits

The Windows resource kits supplement the Support Tools, adding additional tools for system administration and support. The Windows 2003 Resource Kit tools are freely downloadable from *www.microsoft.com* (by searching for "resource kit tools"). They can be installed on Windows XP or Windows Server 2003.

There are two editions of the Windows 2000 resource kits: the Windows 2000 Professional Resource Kit and the Windows 2000 Server Resource Kit. (Supplement 1 is the most recent

version.) Although the latter kit is a superset of the former and can be installed on Windows 2000 Professional systems, none of the experiments in this book use the tools that are included only with the Windows 2000 Server Resource Kit. Unlike the Windows Server 2003 Resource Kit, these tools are not freely downloadable. However, the Windows 2000 Server Resource Kit is included with the MSDN and TechNet subscriptions.

Kernel Debugging

Kernel debugging means examining internal kernel data structures and/or stepping through functions in the kernel. It is a useful way to investigate Windows internals because you can display internal system information not available through any other tools and get a clearer idea of code flows within the kernel.

Kernel debugging can be performed with a variety of tools: the Windows Debugging Tools from Microsoft, LiveKD from *www.sysinternals.com*, or SoftIce from Compuware NuMega. Before describing these tools, let's examine a file that you'll need to perform any type of kernel debugging.

Symbols for Kernel Debugging

Symbol files contain the names of functions and variables. They are generated by the linker and used debuggers to reference and display these names during a debug session. This information is not usually stored in the binary image because it is not needed to execute the code. This means that binaries are smaller and faster. However, this means that when debugging, you must make sure that the debugger can access the symbol files that are associated with the images you are referencing during a debugging session.

To use any of the kernel debugging tools to examine internal Windows kernel data structures (such as the process list, thread blocks, loaded driver list, memory usage information, and so on), you must have the correct symbol files for at least the kernel image, Ntoskrnl.exe. (The section "Architecture Overview" in Chapter 2 explains more about this file.) Symbol table files must match the version of the image they were taken from. For example, if you install a Windows Service Pack or hot fix, you must obtain the matching, updated symbol files for at least the kernel image; otherwise, you'll get a checksum error when you try to load them with the kernel debugger.

While it is possible to download and install symbols for various versions of Windows, updated symbols for hot fixes are not always available. The easiest solution to obtain the correct version of symbols for debugging is to use the Microsoft on-demand symbol server by using a special syntax for the symbol path that you specify in the debugger. For example, the following symbol path causes the debugging tools to load required symbols from the Internet symbol server and keep a local copy in the c:\symbols folder:

```
srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

For detailed instructions on how to use the symbol server, see the Debugging Tools help file or the Web page *www.microsoft.com/whdc/ddk/debugging/symbols.mspx*.

Windows Debugging Tools

The Windows Debugging Tools package contains advanced debugging tools used in this book to explore Windows internals. You can find the latest version at *www.microsoft.com/whdc/ddk/debugging*. These tools can be used to debug user-mode processes as well as the kernel. (See the following sidebar.)

Note The Windows Debugging Tools are updated frequently and released independently of Windows operating system versions, so check often for new versions.

User-Mode Debugging

The debugging tools can also be used to attach to a user-mode process and examine and/or change process memory. There are two options when attaching to a process:

- **Invasive** Unless specified otherwise, when you attach to a running process, the *DebugActiveProcess* Windows function is used to establish a connection between the debugger and the debugee. This permits examining and/or changing process memory, setting breakpoints, and performing other debugging functions. In Windows 2000, when the debugger exits, the debugee process is killed. However, as of Windows XP, you can detach a debugger without killing the target process.
- **Noninvasive** With this option, the debugger simply opens the process with the *OpenProcess* function. It does not attach to the process as a debugger. This allows you to examine and/or change memory in the target process, but you cannot set breakpoints. The advantage of this option is that you can exit the debugger on Windows 2000 without killing the target process.

You can also open user-mode process dump files with the debugging tools. User mode dump files are explained in Chapter 3 in the section on exception dispatching.

There are two primary variants of the Microsoft debuggers that can be used for kernel debugging: a command-line version (Kd.exe) and a graphical user interface (GUI) version (Windbg.exe). Both provide the same set of commands, so which you choose is a matter of personal preference. You can perform three types of kernel debugging with these tools:

- Open a crash dump file created as a result of a Windows system crash. (See Chapter 14 for more information on crash dumps.)
- Connect to a live, running system and examine the system state (or set breakpoints if you're debugging device driver code). This operation requires two computers—a target

and a host. The target is the system being debugged, and the host is the system running the debugger. The target system can be either local (connected to the host via a null modem or IEEE 1394 cable) or remote (connected to the host via a modem). The target system must be booted with the /DEBUG qualifier (either by pressing F8 during the boot process and selecting Debug Mode or by adding a boot selection entry in Boot.ini).

- For Windows XP and Windows Server 2003 systems, connect to the local system and examine the system state. This is called *local kernel debugging*. To initiate local kernel debugging, select the menu item File, select Kernel Debug, click on the Local tab, and click OK. An example output screen is shown in Figure 1-7. Some kernel debugger commands do not work when used in local kernel debugging mode (such as viewing kernel stacks and creating a memory dump with the .dump command). However, you can use the free LiveKd tool from *www.sysinternals.com* in cases where the native local debugging support does not work. (See the next section.)

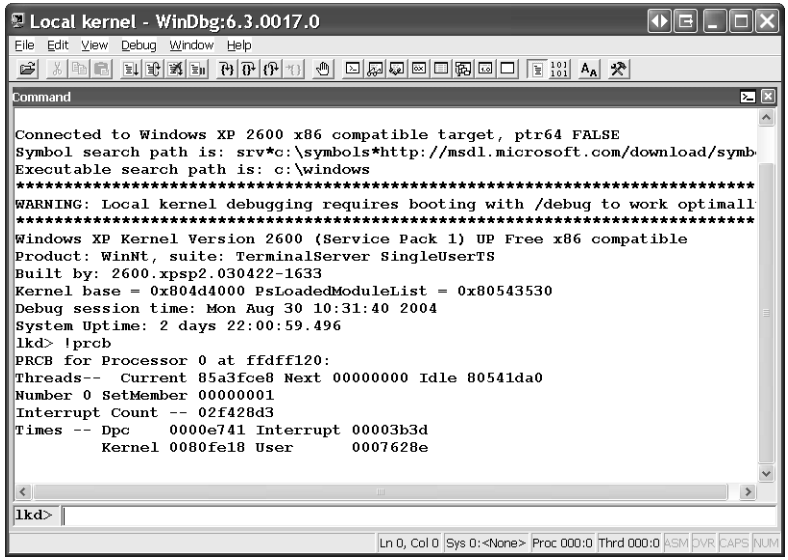


Figure 1-7 Local kernel debugging

Once connected in kernel debugging mode, you can use one of the many *debugger extension commands* (commands that begin with "!") to display the contents of internal data structures such as threads, processes, I/O request packets, and memory management information. Throughout this book, the relevant kernel debugger commands and output are included as they apply to each topic being discussed. In addition, the *dt* (display type) command can format over 400 kernel structures because the kernel symbol files for Windows 2000 Service Pack 3, Windows XP, and Windows Server 2003 contain type information that the debugger can use to format structures.

EXPERIMENT: Displaying Type Information for Kernel Structures

To display the list of kernel structures whose type information is included in the kernel symbols, type **dt nt!_*** in the kernel debugger. A sample partial output is shown below:

```
lkd> dt nt!_*
          nt!_LIST_ENTRY
          nt!_LIST_ENTRY
          nt!_IMAGE_NT_HEADERS
          nt!_IMAGE_FILE_HEADER
          nt!_IMAGE_OPTIONAL_HEADER
          nt!_IMAGE_NT_HEADERS
          nt!_LARGE_INTEGER
```

You can also use the dt command to search for specific structures by using its wildcard lookup capability. For example, if you were looking for the structure name for an interrupt object, type **dt nt!_*interrupt***:

```
lkd> dt nt!_*interrupt*
          nt!_KINTERRUPT
          nt!_KINTERRUPT_MODE
```

Then, you can use dt to format a specific structure as shown below:

```
lkd> dt nt!_kinterrupt
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
   +0x010 ServiceContext   : Ptr32 Void
   +0x014 SpinLock         : Uint4B
   +0x018 TickCount        : Uint4B
   +0x01c ActualLock       : Ptr32 Uint4B
   +0x020 DispatchAddress  : Ptr32
   +0x024 Vector           : Uint4B
   +0x028 Irql             : UChar
   +0x029 SynchronizeIrql  : UChar
   +0x02a FloatingSave     : UChar
   +0x02b Connected        : UChar
   +0x02c Number           : Char
   +0x02d ShareVector      : UChar
   +0x030 Mode             : _KINTERRUPT_MODE
   +0x034 ServiceCount     : Uint4B
   +0x038 DispatchCount    : Uint4B
   +0x03c DispatchCode     : [106] Uint4B
```

Note that dt does not show substructures (structures within structures) by default. To recurse through substructures, use the "-r" switch. For example, using this switch to display the kernel interrupt object shows the format of the _LIST_ENTRY structure stored at the InterruptListEntry field:

```
lkd> dt nt!_kinterrupt -r
nt!_KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry :
      +0x000 Flink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
      +0x004 Blink            : Ptr32
         +0x000 Flink            : Ptr32 _LIST_ENTRY
         +0x004 Blink            : Ptr32 _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
```

The Windows Debugging Tools help file explains how to set up and use the kernel debuggers. Additional details on using the kernel debuggers that are aimed primarily at device driver writers can be found in the Windows DDK documentation. There are also several useful Knowledge Base articles on the kernel debugger. Search for "debugref" in the Windows Knowledge Base (an online database of technical articles) on *support.microsoft.com*.

LiveKd Tool

LiveKd is a free tool from *www.sysinternals.com* that allows you to use the standard Microsoft kernel debuggers just described to examine the running system without requiring a second computer to act as the host (via a null modem cable). While the built-in support for local kernel debugging works only on Windows XP and Windows Server 2003, LiveKd permits local kernel debugging on Windows NT 4.0, Windows 2000, Windows XP, and Windows Server 2003.

You run LiveKd just as you would Windbg or Kd. LiveKd passes any command-line options you specify to the debugger you select. By default, LiveKd runs the new command-line kernel debugger (Kd). To run the GUI debugger (Windbg), specify the −w switch. To see the help files on the switches for LiveKd, specify the −? switch.

LiveKd presents a simulated crash dump file to the debugger, so you can perform any operations in LiveKd that are supported on a crash dump. Because LiveKd is relying on physical memory to back the simulated dump, the kernel debugger might run into situations in which data structures are in the middle of being changed by the system and are inconsistent. Each time the debugger is launched, it gets a snapshot of the system state, so if you want to refresh the snapshot, quit the debugger (with the "q" command) and LiveKd will ask you whether you want to start it again. If the debugger gets in a loop in printing output, press Ctrl+C to interrupt the output, quit, and rerun it. If it hangs, press Ctrl+Break, which will terminate the debugger process and ask you whether you want to run the debugger again.

SoftICE

Another debugging tool that doesn't require two machines for live kernel debugging is a third-party kernel debugger called SoftICE, which you can buy from Compuware NuMega. (See *www.compuware.com* for details.) SoftICE has essentially the same capabilities as the Windows debugging tools, but it also supports stepping between user-mode and kernel-mode code. It also supports the Microsoft kernel extension DLLs, so most of the commands we describe in the book also work in SoftICE. Figure 1-8 shows the SoftICE user interface, which appears in response to the SoftICE activation key (by default, Ctrl+D) as a window on the desktop of the machine on which it's running.

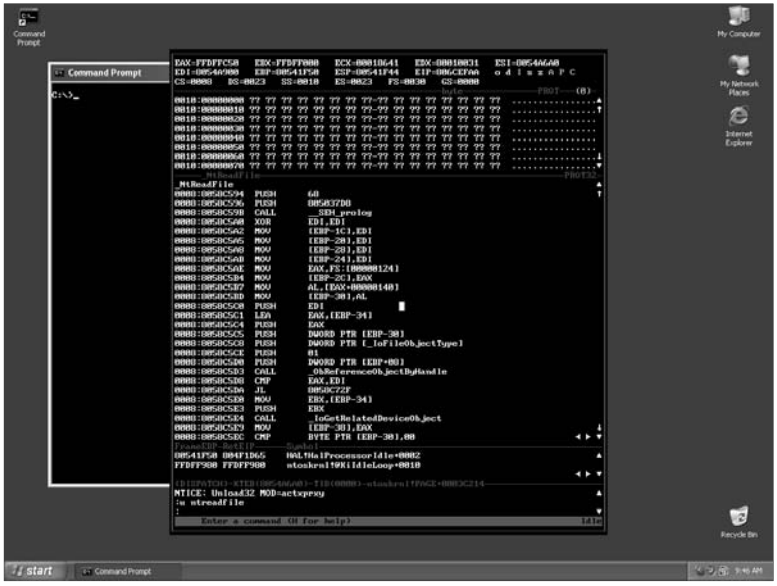


Figure 1-8 The SoftICE interface

Platform Software Development Kit (SDK)

The Platform SDK is part of the MSDN Professional and higher subscription levels, or it can be downloaded for free from *msdn.microsoft.com*. It contains the documentation, C header files, and libraries necessary to compile and link Windows applications. (Although Microsoft Visual C++ comes with a copy of these header files, the versions contained in the Platform SDK always match the latest version of the Windows operating systems, whereas the version that comes with Visual C++ might be an older version that was current when Visual C++ was released.) From an internals perspective, items of interest in the Platform SDK include the Windows API header files (\Program Files\Microsoft SDK\Include) as well as several utilities (Pfmon.exe, Pstat.exe, Pview.exe, Vadump.exe, and Winobj.exe). Some tools in the Platform SDK also come with the Support Tools and Resource Kits. Finally, a few of these tools are also shipped as example source code in both the Platform SDK and the MSDN Library.

Device Driver Kit (DDK)

The Windows DDK is also shipped as part of the MSDN Professional (and higher) subscription levels, but unlike the Platform SDK, it is not available for free download (although you can order the CD-ROM for a minimal cost). The Windows DDK documentation is included in the MSDN Library.

Although the DDK is aimed at device-driver developers, it is an abundant source of Windows-internals information. For example, while Chapter 9 describes the I/O system architecture, driver model, and basic device driver data structures, it does not describe the individual kernel support functions in detail. The DDK documentation contains a comprehensive description of all the Windows kernel support functions and mechanisms used by device drivers in both a tutorial and reference form.

Besides including the documentation, the DDK contains header files (in particular, Ntddk.h and Wdm.h) that define key internal data structures and constants as well as interfaces to many internal system routines. These files are useful when exploring Windows internal data structures with the kernel debugger because although the general layout and content of these structures are shown in this book, detailed field-level descriptions (such as size and data types) are not. A number of these data structures (such as object dispatcher headers, wait blocks, events, mutants, semaphores, and so on) are, however, fully described in the DDK.

So if you want to dig into the I/O system and driver model beyond what is presented in this book, read the DDK documentation (especially the Kernel-Mode Driver Architecture Design Guide and Reference manuals). Another excellent source is *Programming the Microsoft Windows Driver Model, Second Edition* (Microsoft Press) by Walt Oney.

Sysinternals Tools

Many experiments in this book use freeware tools that you can download from *www.sysinternals.com*. Mark Russinovich, coauthor of this book, wrote most of these tools. The most popular tools include Process Explorer, Filemon, and Regmon. Note that many of these utilities involve the installation and execution of kernel-mode device drivers and thus require administrator privileges.

Conclusion

In this chapter, you've been introduced to the key Windows technical concepts and terms that will be used throughout the book. You've also had a glimpse of the many useful tools available for digging into Windows internals. Now we're ready to begin our exploration of the internal design of the system, beginning with an overall view of the system architecture and its key components.

Chapter 2

System Architecture

Now that we've covered the terms, concepts, and tools you need to be familiar with, we're ready to start our exploration of the internal design goals and structure of the Microsoft Windows operating system. This chapter explains the overall architecture of the system—the key components, how they interact with each other, and the context in which they run. To provide a framework for understanding the internals of Windows, let's first review the requirements and goals that shaped the original design and specification of the system.

Requirements and Design Goals

The following requirements drove the specification of Windows NT back in 1989:

- Provide a true 32-bit, preemptive, reentrant, virtual memory operating system
- Run on multiple hardware architectures and platforms
- Run and scale well on symmetric multiprocessing systems
- Be a great distributed computing platform, both as a network client and as a server
- Run most existing 16-bit MS-DOS and Microsoft Windows 3.1 applications
- Meet government requirements for POSIX 1003.1 compliance
- Meet government and industry requirements for operating system security
- Be easily adaptable to the global market by supporting Unicode

To guide the thousands of decisions that had to be made to create a system that met these requirements, the Windows NT design team adopted the following design goals at the beginning of the project:

- **Extensibility** The code must be written to comfortably grow and change as market requirements change.
- **Portability** The system must be able to run on multiple hardware architectures and must be able to move with relative ease to new ones as market demands dictate.
- **Reliability and robustness** The system should protect itself from both internal malfunction and external tampering. Applications should not be able to harm the operating system or other applications.

- **Compatibility** Although Windows NT should extend existing technology, its user interface and APIs should be compatible with older versions of Windows and with MS-DOS. It should also interoperate well with other systems such as UNIX, OS/2, and NetWare.
- **Performance** Within the constraints of the other design goals, the system should be as fast and responsive as possible on each hardware platform.

As we explore the details of the internal structure and operation of Windows, you'll see how these original design goals and market requirements were woven successfully into the construction of the system. But before we start that exploration, let's examine the overall design model for Windows and compare it with other modern operating systems.

Operating System Model

In most multiuser operating systems, applications are separated from the operating system itself—the operating system kernel code runs in a privileged processor mode (referred to as *kernel mode* in this book), with access to system data and to the hardware; application code runs in a nonprivileged processor mode (called *user mode*), with a limited set of interfaces available, limited access to system data, and no direct access to hardware. When a user-mode program calls a system service, the processor traps the call and then switches the calling thread to kernel mode. When the system service completes, the operating system switches the thread context back to user mode and allows the caller to continue.

Windows is similar to most UNIX systems in that it's a monolithic operating system in the sense that the bulk of the operating system and device driver code shares the same kernel-mode protected memory space. This means that any operating system component or device driver can potentially corrupt data being used by other operating system components.

Is Windows a Microkernel-Based System?

Although some claim it as such, Windows isn't a microkernel-based operating system in the classic definition of microkernels, where the principal operating system components (such as the memory manager, process manager, and I/O manager) run as separate processes in their own private address spaces, layered on a primitive set of services the microkernel provides. For example, the Carnegie Mellon University Mach operating system, a contemporary example of a microkernel architecture, implements a minimal kernel that comprises thread scheduling, message passing, virtual memory, and device drivers. Everything else, including various APIs, file systems, and networking, runs in user mode. However, commercial implementations of the Mach microkernel operating system typically run at least all file system, networking, and memory management code in kernel mode. The reason is simple: the pure microkernel design is commercially impractical because it's too inefficient.

> Does the fact that so much of Windows runs in kernel mode mean that it's more susceptible to crashes than a true microkernel operating system? Not at all. Consider the following scenario. Suppose the file system code of an operating system has a bug that causes it to crash from time to time. In a traditional operating system, a bug in kernel-mode code such as the memory manager or the file system would likely crash the entire operating system. In a pure microkernel operating system, such components run in user mode, so theoretically a bug would simply mean that the component's process exits. But in practical terms, the system would crash because recovering from the failure of such a critical process would likely be impossible.

All these operating system components are, of course, fully protected from errant applications because applications don't have direct access to the code and data of the privileged part of the operating system (although they can quickly call other kernel services). This protection is one of the reasons that Windows has the reputation for being both robust and stable as an application server and as a workstation platform yet fast and nimble from the perspective of core operating system services, such as virtual memory management, file I/O, networking, and file and print sharing.

The kernel-mode components of Windows also embody basic object-oriented design principles. For example, they don't in general reach into one another's data structures to access information maintained by individual components. Instead, they use formal interfaces to pass parameters and access and/or modify data structures.

Despite its pervasive use of objects to represent shared system resources, Windows is not an object-oriented system in the strict sense. Most of the operating system code is written in C for portability and because C development tools are widely available. C doesn't directly support object-oriented constructs, such as dynamic binding of data types, polymorphic functions, or class inheritance. Therefore, the C-based implementation of objects in Windows borrows from, but doesn't depend on, features of particular object-oriented languages.

Architecture Overview

With this brief overview of the design goals and packaging of Windows, let's take a look at the key system components that make up its architecture. A simplified version of this architecture is shown in Figure 2-1. Keep in mind that this diagram is basic—it doesn't show everything. (For example, the networking components and the various types of device driver layering are not shown.)

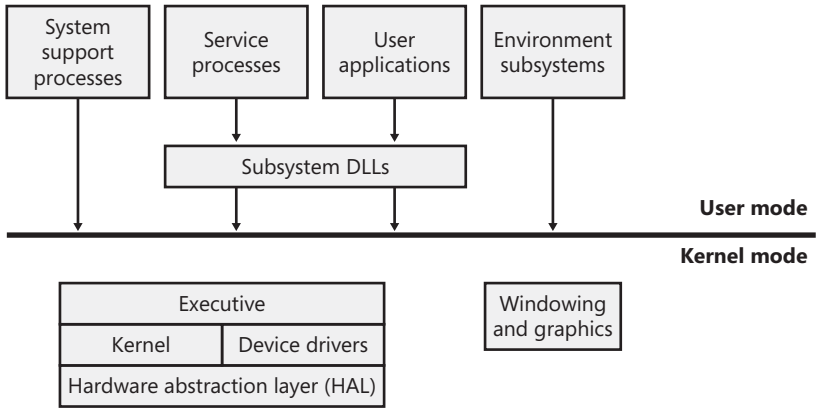


Figure 2-1 Simplified Windows architecture

In Figure 2-1, first notice the line dividing the user-mode and kernel-mode parts of the Windows operating system. The boxes above the line represent user-mode processes, and the components below the line are kernel-mode operating system services. As mentioned in Chapter 1, user-mode threads execute in a protected process address space (although while they are executing in kernel mode, they have access to system space). Thus, system support processes, service processes, user applications, and environment subsystems each have their own private process address space.

The four basic types of user-mode processes are described as follows:

- Fixed (or hardwired) *system support processes*, such as the logon process and the session manager, that are not Windows services. (That is, they are not started by the service control manager. Chapter 4 describes services in detail.)
- *Service processes* that host Windows services, such as the Task Scheduler and Spooler services. Services generally have the requirement that they run independently of user logons. Many Windows server applications, such as Microsoft SQL Server and Microsoft Exchange Server, also include components that run as services.
- *User applications*, which can be one of six types: Windows 32-bit, Windows 64-bit, Windows 3.1 16-bit, MS-DOS 16-bit, POSIX 32-bit, or OS/2 32-bit.
- *Environment subsystem server processes*, which implement part of the support for the operating system *environment*, or personality presented to the user and programmer. Windows NT originally shipped with three environment subsystems: Windows, POSIX, and OS/2. OS/2 was dropped as of Windows 2000. As of Windows XP, only the Windows subsystem is shipped in the base product—an enhanced POSIX subsystem is available as part of the free Services for Unix product.

In Figure 2-1, notice the "Subsystem DLLs" box below the "Service processes" and "User applications" boxes. Under Windows, user applications don't call the native Windows operating system services directly; rather, they go through one or more *subsystem dynamic-link libraries* (DLLs). The

role of the subsystem DLLs is to translate a documented function into the appropriate internal (and generally undocumented) Windows system service calls. This translation might or might not involve sending a message to the environment subsystem process that is serving the user application.

The kernel-mode components of Windows include the following:

- The Windows *executive* contains the base operating system services, such as memory management, process and thread management, security, I/O, networking, and interprocess communication.
- The Windows *kernel* consists of low-level operating system functions, such as thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization. It also provides a set of routines and basic objects that the rest of the executive uses to implement higher-level constructs.
- *Device drivers* include both hardware device drivers that translate user I/O function calls into specific hardware device I/O requests as well as file system and network drivers.
- The *hardware abstraction layer* (HAL) is a layer of code that isolates the kernel, device drivers, and the rest of the Windows executive from platform-specific hardware differences (such as differences between motherboards).
- The *windowing and graphics system* implements the graphical user interface (GUI) functions (better known as the Windows USER and GDI functions), such as dealing with windows, user interface controls, and drawing.

Table 2-1 lists the filenames of the core Windows operating system components. (You'll need to know these filenames because we'll be referring to some system files by name.) Each of these components is covered in greater detail both later in this chapter and in the chapters that follow.

Table 2-1 Core Windows System Files

Filename	Components
Ntoskrnl.exe	Executive and kernel
Ntkrnlpa.exe (32-bit systems only)	Executive and kernel with support for Physical Address Extension (PAE), which allows addressing of up to 64 GB of physical memory
Hal.dll	Hardware abstraction layer
Win32k.sys	Kernel-mode part of the Windows subsystem
Ntdll.dll	Internal support functions and system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Windows subsystem DLLs

Before we dig into the details of these system components, though, let's examine how Windows achieves portability across multiple hardware architectures.

Portability

Windows was designed to run on a variety of hardware architectures, including Intel-based CISC systems as well as RISC systems. The initial release of Windows NT supported the x86 and MIPS architecture. Support for the Digital Equipment Corporation (which was bought by Compaq, who later merged with Hewlett Packard) Alpha AXP was added shortly thereafter. (Although Alpha AXP was a 64-bit processor, Windows NT ran in 32-bit mode. During the development of Windows 2000, a native 64-bit version was running on Alpha AXP, but this never was released.) Support for a fourth processor architecture, the Motorola PowerPC, was added in Windows NT 3.51. Because of changing market demands, however, support for the MIPS and PowerPC architectures was dropped before development began on Windows 2000. Later, Compaq withdrew support for the Alpha AXP architecture, resulting in Windows 2000 being supported only on the x86 architecture. The most recent releases, Windows XP and Windows Server 2003, add support for three 64-bit processor families: the Intel Itanium IA-64 family, the AMD x86-64 family, and the Intel 64-bit Extension Technology (EM64T) for x86 (which is compatible with the AMD x86-64 architecture, although there are slight differences in instructions supported). The latter two processor families are called *64-bit extended systems* and in this book are referred to as *x64*. The most recent releases, Windows XP and Windows Server 2003, add support for three 64-bit processor families: the Intel Itanium IA-64 family, the AMD64 family, and the Intel 64-bit Extension Technology (EM64T) for x86 (which is compatible with the AMD64 architecture, although there are slight differences in instructions supported). (How Windows runs 32-bit applications on 64-bit Windows is explained in Chapter 3.)

Windows achieves portability across hardware architectures and platforms in two primary ways:

- Windows has a layered design, with low-level portions of the system that are processor-architecture-specific or platform-specific isolated into separate modules so that upper layers of the system can be shielded from the differences between architectures and among hardware platforms. The two key components that provide operating system portability are the kernel (contained in Ntoskrnl.exe) and the hardware abstraction layer (or HAL, contained in Hal.dll). Both these components are described in more detail later in this chapter. Functions that are architecture-specific (such as thread context switching and trap dispatching) are implemented in the kernel. Functions that can differ among systems within the same architecture (for example, different motherboards) are implemented in the HAL. The only other component with a significant amount of architecture-specific code is the memory manager, but even that is a small amount compared to the system as a whole.
- The vast majority of Windows is written in C, with some portions in C++. Assembly language is used only for those parts of the operating system that need to communicate directly with system hardware (such as the interrupt trap handler) or that are extremely performance-sensitive (such as context switching). Assembly language code exists not only in the kernel and the HAL but also in a few other places within the core operating system (such as the routines that implement interlocked instructions as well as one module in the local procedure call facility), in the kernel-mode part of the Windows

subsystem, and even in some user-mode libraries, such as the process startup code in Ntdll.dll (a system library explained later in this chapter).

Symmetric Multiprocessing

Multitasking is the operating system technique for sharing a single processor among multiple threads of execution. When a computer has more than one processor, however, it can execute two threads simultaneously. Thus, whereas a multitasking operating system only appears to execute multiple threads at the same time, a multiprocessing operating system actually does it, executing one thread on each of its processors.

As mentioned at the beginning of this chapter, one of the key design goals for Windows was that it had to run well on multiprocessor computer systems. Windows is a *symmetric multiprocessing* (SMP) operating system. There is no master processor—the operating system as well as user threads can be scheduled to run on any processor. Also, all the processors share just one memory space. This model contrasts with *asymmetric multiprocessing* (ASMP), in which the operating system typically selects one processor to execute operating system kernel code while other processors run only user code. The differences in the two multiprocessing models are illustrated in Figure 2-2.

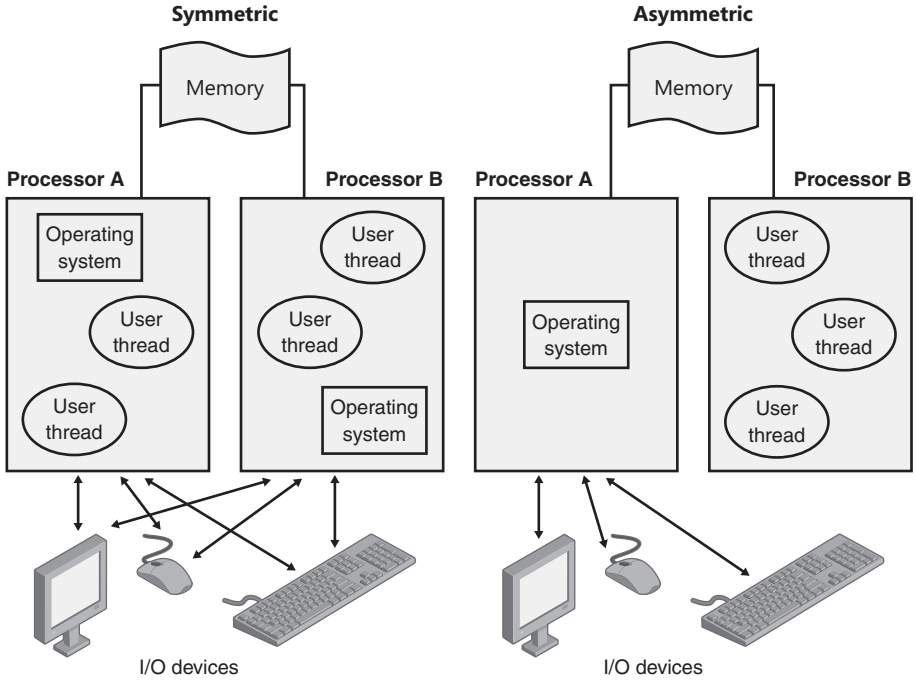


Figure 2-2 Symmetric vs. asymmetric multiprocessing

Windows XP and Windows Server 2003 support two new types of multiprocessor systems: hyperthreading and NUMA (non-uniform memory architecture). These are briefly mentioned in the following paragraphs. (For a complete detailed description of the scheduling support for these systems, see the thread scheduling section in Chapter 6.)

Hyperthreading is a technology introduced by Intel that provides many logical processors on one physical processor. Each logical processor has its CPU state, but the execution engine and onboard cache is shared. This permits one logical CPU to make progress while the other logical CPUs are busy (such as performing interrupt processing work, which prevents threads from running on that logical processor). The scheduling algorithms as of Windows XP have been enhanced to make optimal use of multiprocessor hyperthreaded machines, such as by scheduling threads on an idle physical processor versus choosing an idle logical processor on a physical processor whose other logical processors are busy.

In non-uniform memory architecture NUMA systems, processors are grouped 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. Windows on a NUMA system still runs as an SMP system, in that all processors have access to all memory—it's just that node-local memory is faster to reference than memory attached to other nodes. The system attempts to improve performance by scheduling threads on processors that are in the same node as the memory being used. It attempts to satisfy memory-allocation requests from within the node, but will allocate memory from other nodes if necessary.

Although Windows was originally designed to support up to 32 processors, nothing inherent in the multiprocessor design limits the number of processors to 32—that number is simply an obvious and convenient limit because 32 processors can easily be represented as a bit mask using a native 32-bit data type. In fact, the 64-bit versions of Windows support up to 64 processors, because the native size of a word on a 64-bit machine is 64 bits.

The actual number of supported processors depends on the edition of Windows being used. (See tables 2-3 and 2-4.) This number is stored in the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\LicensedProcessors. (Keep in mind that tampering with that data is a violation of the software license and modifying the registry to allow use of more processors involves more than just changing this value.)

For performance reasons, there are separate uniprocessor and multiprocessor versions of the kernel and HAL (and in the case of Windows 2000, a few other key system files). On Windows 2000, six system files (as explained in the following Note) are different on a multiprocessor system than on a uniprocessor system; on 32-bit Windows XP and Windows Server 2003 systems, only three are different. (See Table 2-2.) On 64-bit Windows systems, there is no PAE kernel, so only the kernel and HAL vary from uniprocessor to multiprocessor systems.

At installation time, the appropriate files are selected and copied to the local \Windows\System32 directory. To determine which files were copied, see the file \Windows\Repair\Setup.log, which itemizes all the files that were copied to the local system disk and where they came from off the distribution media.

Table 2-2 Multiprocessor-Specific vs. Uniprocessor-Specific System Files

Name of File on System Disk	Name of Uniprocessor Version on Distribution Media	Name of Multiprocessor Version on Distribution Media
Ntoskrnl.exe	Ntoskrnl.exe	Ntkrnlmp.exe
Ntkrnlpa.exe (PAE kernel; 32-bit systems only)	Ntkrnlpa.exe in \Windows\<arch>\Driver.cab	Ntkrpamp.exe in \Windows\<arch>\Driver.cab
Hal.dll	Depends on system type (See the list of HALs in Table 2-6.)	Depends on system type (See the list of HALs in Table 2-6.)
Windows 2000 only		
Win32k.sys	\I386\UNIPROC\Win32k.sys	Win32k.sys in \I386\Driver.cab
Ntdll.dll	\I386\UNIPROC\Ntdll.dll	\I386\Ntdll.dll
Kernel32.dll	\I386\UNIPROC\Kernel32.dll	\I386\Kernel32.dll

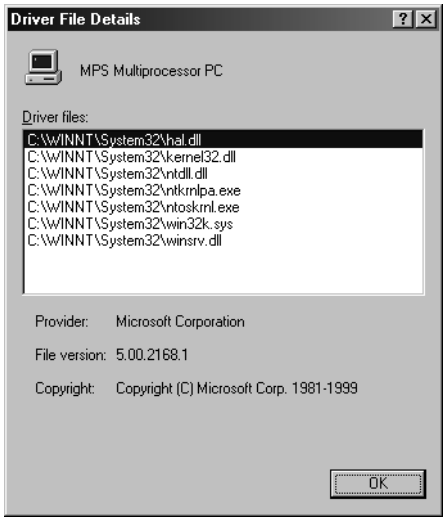
Note If you look in the \I386\UNIPROC folder in the Windows 2000 distribution tree, you'll see a file named Winsrv.dll. Although this file exists in a folder named UNIPROC, implying that there is a uniprocessor version, in fact there is only one version of this image for both multiprocessor and uniprocessor systems. This folder has been removed in Windows XP and Windows Server 2003.

EXPERIMENT: Looking at Multiprocessor-Specific Support Files on Windows 2000

You can see the files that are different for a 32-bit Windows 2000 multiprocessor system by looking at the driver details for the Computer in Device Manager:

1. Open the System properties (either by selecting System from Control Panel or by right-clicking the My Computer icon on your desktop and selecting Properties).
2. Click the Hardware tab.
3. Click Device Manager.
4. Expand the Computer object.
5. Double-click the child node underneath Computer.
6. Click the Driver tab.
7. Click Driver Details.

You should see the following dialog box for a multiprocessor system:



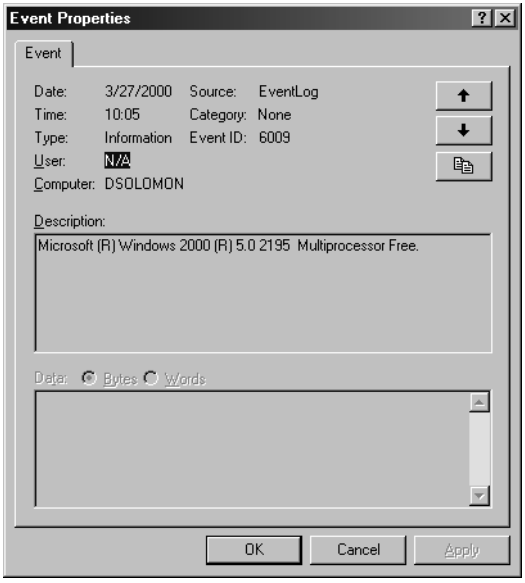
The reason for having uniprocessor versions of these key system files is performance—multiprocessor synchronization is inherently more complex and time consuming than the use of a single processor, so by having special uniprocessor versions of the key system files, this overhead is avoided on uniprocessor systems (which constitute the vast majority of systems running Windows).

Interestingly, although the uniprocessor and multiprocessor versions of Ntoskrnl are generated using conditionally compiled source code, the uniprocessor versions of Ntdll.dll and Kernel32.dll for Windows 2000 are created by patching the x86 LOCK and UNLOCK instructions, which are used to synchronize multiple threads with no-operation (NOP) instructions (which do nothing).

The rest of the system files that make up Windows (including all utilities, libraries, and device drivers) have the same version on both uniprocessor and multiprocessor systems (that is, they handle multiprocessor synchronization issues correctly). You should use this approach on any software you build, whether it is a Windows application or a device driver—keep multiprocessor synchronization issues in mind when you design your software, and test the software on both uniprocessor and multiprocessor systems.

EXPERIMENT: Checking Which Ntoskrnl Version You're Running

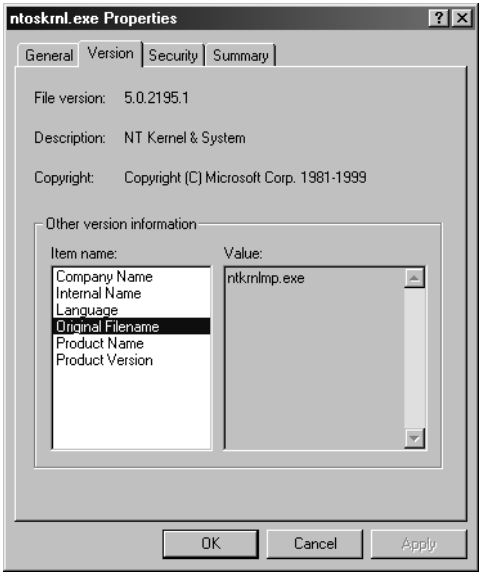
In Windows 2000 and later, there is no utility to show which version of Ntoskrnl you are running. However, an Event Log entry is written each time the system boots that does record the type of kernel image that loaded (uniprocessor vs. multiprocessor and free vs. checked), as shown in the following screen shot. (From the Start menu select Programs/Administrative Tools/Event Viewer, select System Log, and double-click an Event Log entry with an Event ID of 6009, indicating the entry was written at the system start.)



This Event Log entry doesn't indicate whether you booted the PAE version of the kernel image that supports more than 4 GB of physical memory (Ntkrnlpa.exe). However, you can tell if you booted the PAE kernel by looking at the registry value HKLM\SYSTEM\CurrentControlSet\Control\SystemStartOptions. Also, if you boot the PAE kernel, the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PhysicalAddressExtension is set to 1.

You can also determine whether you installed the multiprocessor version of Ntoskrnl (or Ntkrnlpa) by examining the file properties: run Windows Explorer, right-click on Ntoskrnl.exe in your \Windows\System32 folder, and select Properties. Then click on

the Version tab, and select the Original Filename property—if you're running the multiprocessor version, you'll see the following dialog box:



Finally, as mentioned earlier, you can see exactly which kernel image and HAL were selected at installation time by looking at the file \Windows\Repair\Setup.log.

Scalability

One of the key issues with multiprocessor systems is *scalability*. To run correctly on an SMP system, operating system code must adhere to strict guidelines and rules. Resource contention and other performance issues are more complicated in multiprocessing systems than in uniprocessor systems and must be accounted for in the system's design. Windows incorporates several features that are crucial to its success as a multiprocessor operating system:

- The ability to run operating system code on any available processor and on multiple processors at the same time
- Multiple threads of execution within a single process, each of which can execute simultaneously on different processors
- Fine-grained synchronization within the kernel (such as spinlocks, queued spinlocks, and pushlocks, described in Chapter 3) as well as within device drivers and server processes, which allows more components to run concurrently on multiple processors
- Programming mechanisms such as I/O completion ports (described in Chapter 9) that facilitate the efficient implementation of multithreaded server processes that can scale well on multiprocessor systems.

The scalability of the Windows kernel has evolved over time. For example, Windows Server 2003 has per-CPU scheduling queues, which permits thread scheduling decisions to occur in parallel on multiple machines. Multiprocessor thread scheduling details are covered in Chapter 6. Further details on multiprocessor synchronization can be found in Chapter 3.

Differences Between Client and Server Versions

Windows ships in both client and server retail packages. In Windows 2000, the client version is called Windows 2000 Professional. There are three Windows 2000 server versions: Windows 2000 Server, Advanced Server, and Datacenter Server.

There are six client versions of Windows XP: Windows XP Home Edition, Windows XP Professional, Windows XP Starter Edition, Windows XP Tablet PC Edition, Windows XP Media Center Edition, and Windows XP Embedded. The latter three are supersets of Windows XP Professional and are not described in detail in this book because they are all built on the same core operating system as Windows XP Professional.

There are six variants of Windows Server 2003: Windows Server 2003 Web Edition, Standard Edition, Small Business Server, Storage Server, Enterprise Edition, and Datacenter Edition.

These versions differ by:

- The number of processors supported
- The amount of physical memory supported
- The number of concurrent network connections supported (For example, a maximum of 10 concurrent connections are allowed to the file and print services in the client version.)
- Layered services that come with Server editions that don't come with the Professional edition (for example, directory services, clustering, and multiuser Terminal Services support)

Table 2-3 summarizes the differences in memory and processor support for Windows 2000. Table 2-4 lists the same information for Windows XP and Windows Server 2003. For a detailed comparison chart of the different editions of Windows Server 2003, see *http://www.microsoft.com/windowsserver2003/evaluation/features/compareeditions.mspx*.

Table 2-3 Differences Between Windows 2000 Professional and Server

Edition	Number of Processors Supported	Physical Memory Supported
Windows 2000 Professional	2	4 GB
Windows 2000 Server	4	4 GB
Windows 2000 Advanced Server	8	8 GB
Windows 2000 Datacenter Server	32	64 GB

Table 2-4 Differences Between Windows XP and Windows Server 2003

	Number of Processors Supported (32-bit edition)	Physical Memory Supported (32-bit edition)	Number of Processors Supported (64-bit edition)	Physical Memory Supported (Itanium editions)	Physical Memory Supported (x64 editions)
Windows XP Home Edition	1	4 GB	Not available	Not available	Not available
Windows XP Professional	2	4 GB	2	16 GB	16 GB
Windows Server 2003 Web Edition	2	2 GB	Not available	Not available	Not available
Windows Server 2003 Small Business Server	2	2 GB	Not available	Not available	Not available
Windows Server 2003 Standard Edition	4	4 GB	Not available	Not available	Not available
Windows Server 2003 Enterprise Edition	8	32 GB	8	64 GB	64 GB
Windows Server 2003 Datacenter Edition	32	128 GB on x64; 64 GB on x86	64	512 GB (1024 GB in SP1)	Not available

Although there are several client and server retail packages of the Windows operating system, they share a common set of core system files, including the kernel image, Ntoskrnl.exe (and the PAE version, Ntkrnlpa.exe); the HAL libraries; the device drivers; and the base system utilities and DLLs. These files are identical for all editions of Windows 2000.

Note Windows XP was the first client release of the Windows NT code base to ship without corresponding server versions. Instead, development continued on what became Windows Server 2003 for over a year after the release of Windows XP. Therefore, the core system files are not identical for Windows XP and Windows Server 2003. Even so, the differences are not major (and in many cases, components were unchanged).

So if the kernel image for Windows 2000 Professional and Windows 2000 Server are identical (and similar for Windows XP and Windows Server 2003), how does the system know which edition is booted? By querying the registry values ProductType and ProductSuite under the HKLM\ SYSTEM\CurrentControlSet\Control\ProductOptions key. ProductType is used to

distinguish whether the system is a client system or a server system (of any flavor). The valid values are listed in Table 2-5. The result is stored in the system global variable *MmProductType*, which can be queried from a device driver using the kernel-mode support function *MmIsThisAnNtAsSystem*, documented in the Windows DDK.

Table 2-5 ProductType Registry Values

Edition of Windows	Value of ProductType
Windows 2000 Professional, Windows XP Professional, Windows XP Home Edition	WinNT
Windows Server (domain controller)	LanmanNT
Windows Server (server only)	ServerNT

A different registry value, ProductSuite, distinguishes the various flavors of Windows Server systems (Standard, Enterprise, Datacenter, and so on) as well as distinguishing a Windows XP Home from a Windows XP Professional system.

If user programs need to determine which edition of Windows is running, they can call the Windows *VerifyVersionInfo* function, documented in the Platform SDK. Device drivers can call the kernel-mode function *RtlGetVersion*, documented in the Windows DDK.

So if the core files are essentially the same for the client and server versions, how do the systems differ in operation? In short, Server systems are by default optimized for system throughput as high-performance application servers, whereas the client version, although it has server capabilities, is optimized for response time for interactive desktop use. For example, based on the product type, several resource allocation decisions are made differently at system boot time, such as the size and number of operating system heaps (or pools), the number of internal system worker threads, and the size of the system data cache. Also, run-time policy decisions, such as the way the memory manager trades off system and process memory demands, differ between the server and client editions. Even some thread scheduling details have different default behavior in the two families (the default length of the time slice, or thread *quantum*—see Chapter 6 for details). Where there are significant operational differences in the two products, these are highlighted in the pertinent chapters throughout the rest of this book. Unless otherwise noted, everything in this book applies to both the client and server versions.

Checked Build

There is a special debug version of Windows 2000 Professional, Windows XP Professional, and Windows Server 2003 called the *checked build* (available only with the MSDN Professional or higher subscription). It is a recompilation of the Windows source code with a compile-time flag defined called "DBG" (to cause compile time conditional debugging and tracing code to be included). Also, to make it easier to understand the machine code, the post-processing of the Windows binaries to optimize code layout for faster execution is not performed. (See the section "Performance-Optimized Code" in the Debugging Tools help file.)

The checked build is provided primarily to aid device driver developers because it performs more stringent error checking on kernel-mode functions called by device drivers or other system code. For example, if a driver (or some other piece of kernel-mode code) makes an invalid call to a system function that is checking parameters (such as acquiring a spinlock at the wrong interrupt level), the system will stop execution when the problem is detected rather than allow some data structure to be corrupted and the system to possibly crash at a later time.

EXPERIMENT: Determining If You Are Running the Checked Build

There is no built-in tool to display whether you are running the checked build or the retail build (called the free build). However, this information is available through the "Debug" property of the Windows Management Instrumentation (WMI) Win32_OperatingSystem class. The following sample Visual Basic script displays this property:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _
 & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")
Set colOperatingSystems = objWMIService.ExecQuery _
 ("SELECT * FROM Win32_OperatingSystem"S)
For Each objOperatingSystem in colOperatingSystems
 Wscript.Echo "Caption: " & objOperatingSystem.Caption
 Wscript.Echo "Debug: " & objOperatingSystem.Debug
 Wscript.Echo "Version: " & objOperatingSystem.Version
Next
```

To try this, type in the preceding script and save it as file. The following is the output from running the script:

```
C:\>cscript osversion.vbs
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

Caption: Microsoft Windows XP Professional
Debug: False
Version: 5.1.2600
```

This system is not running the checked build, as the Debug flag shown here says False.

Much of the additional code in the checked-build binaries is a result of using the ASSERT macro, which is defined in the DDK header file Ntddk.h and documented in the DDK documentation. This macro tests a condition (such as the validity of a data structure or parameter), and if the expression evaluates to FALSE, the macro calls the kernel-mode function *RtlAssert*, which calls *DbgPrint* to send the text of the debug message to a debug message buffer. If a kernel debugger is attached, this message is displayed automatically followed by a prompt asking the user what to do about the assertion failure (breakpoint, ignore, terminate process, or terminate thread). If the system wasn't booted with the kernel debugger (using the /DEBUG

switch in Boot.ini) and no kernel debugger is currently attached, failure of an ASSERT test will crash the system. For a list of ASSERT checks made by some of the kernel support routines, see the section "Checked Build ASSERTs" in the Windows DDK documentation.

Note On the checked build, if you compare Ntoskrnl.exe and Ntkrnlmp.exe or Ntkrnlpa.exe and Ntkrpamp.exe, you'll find that they are identical—they are all multiprocessor versions of the same files. In other words, there is no debug uniprocessor version of the kernel images provided with the checked build.

The checked build is also useful for system administrators because of the additional detailed informational tracing that can be enabled for certain components. (For detailed instructions, see the Microsoft Knowledge Base Article number 314743 entitled *HOWTO: Enable Verbose Debug Tracing in Various Drivers and Subsystems*.) This information output is sent to an internal debug message buffer using the *DbgPrint* function referred to earlier. To view the debug messages, you can either attach a kernel debugger to the target system (which requires booting the target system in debugging mode), use the *!dbgprint* command while performing local kernel debugging, or use the Dbgview.exe tool from *www.sysinternals.com*.

You don't have to install the entire checked build to take advantage of the debug version of the operating system. You can just copy the checked version of the kernel image (Ntoskrnl.exe) and the appropriate HAL (Hal.dll) to a normal retail installation. The advantage of this approach is that device drivers and other kernel code get the rigorous checking of the checked build without having to run the slower debug versions of all components in the system. For detailed instructions on how to do this, see the section "Installing Just the Checked Operating System and HAL" in the Windows DDK documentation. Because Microsoft doesn't supply a checked build version of Windows 2000 Server, you can also apply this technique to run the checked version of the kernel on a Windows 2000 Server system.

Finally, the checked build can also be useful for testing user-mode code only because the timing of the system is different. (This is because of the additional checking taking place within the kernel and the fact that the components are compiled without optimizations.) Often, multithreaded synchronization bugs are related to specific timing conditions. By running your tests on a system running the checked build (or at least the checked kernel and HAL), the fact that the timing of the whole system is different might cause latent timing bugs to surface that do not occur on a normal retail system.

Key System Components

Now that we've looked at the high-level architecture of Windows, let's delve deeper into the internal structure and the role each key operating system component plays. Figure 2-3 is a more detailed and complete diagram of the core Windows system architecture and components than was shown earlier in the chapter (in Figure 2-2). Note that it still does not show all components (networking in particular, which is explained in Chapter 13).

The following sections elaborate on each major element of this diagram. Chapter 3 explains the primary control mechanisms the system uses (such as the object manager, interrupts, and so forth). Chapter 5 describes the process of starting and shutting down Windows, and Chapter 4 details management mechanisms such as the registry, service processes, and Windows Management Instrumentation (WMI). Then the remaining chapters explore in even more detail the internal structure and operation of key areas such as processes and threads, memory management, security, the I/O manager, storage management, the cache manager, the Windows file system (NTFS), and networking.

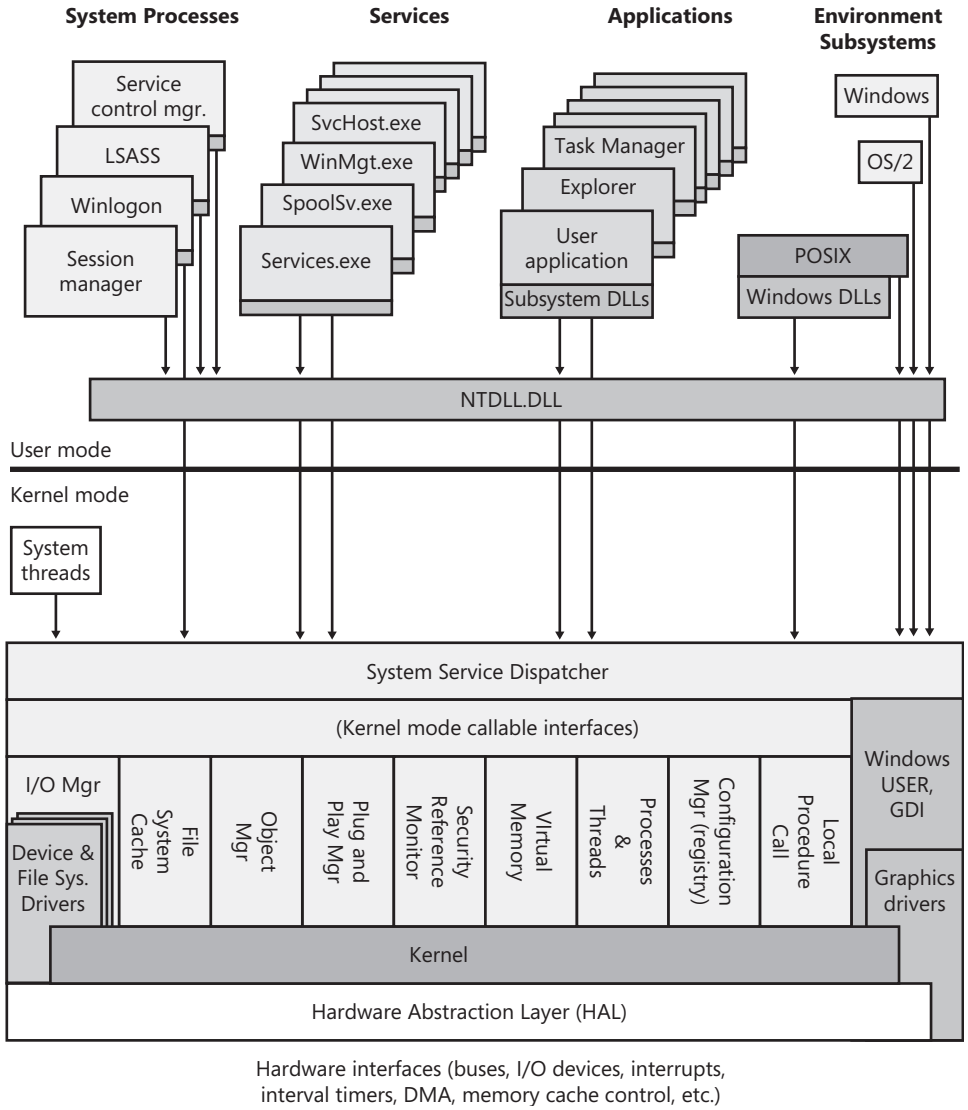


Figure 2-3 Windows architecture

Environment Subsystems and Subsystem DLLs

As shown in Figure 2-3, Windows originally had three environment subsystems: OS/2, POSIX, and Windows. As stated earlier, the OS/2 subsystem was removed in Windows 2000. Although the basic POSIX subsystem that originally shipped with Windows no longer ships with the system as of Windows XP, a greatly enhanced version is available for free as part of the Services for UNIX product.

As we'll explain shortly, of the three, the Windows subsystem is special in that Windows can't run without it. (It owns the keyboard, mouse, and display, and it is required to be present even on server systems with no interactive users logged in.) In fact, the other two subsystems are configured to start on demand, whereas the Windows subsystem must always be running.

The subsystem startup information is stored under the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems. Figure 2-4 shows the values under this key.

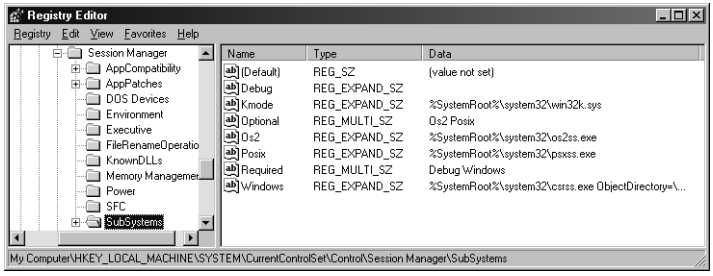


Figure 2-4 Registry Editor showing Windows startup information

The Required value lists the subsystems that load when the system boots. The value has two strings: Windows and Debug. The Windows value contains the file specification of the Windows subsystem, Csrss.exe, which stands for Client/Server Run-Time Subsystem. (See the Note later in this section.) Debug is blank (because it's used for internal testing) and therefore does nothing. The Optional value indicates that the OS/2 and POSIX subsystems will be started on demand. The registry value Kmode contains the filename of the kernel-mode portion of the Windows subsystem, Win32k.sys (explained later in this chapter).

The role of an environment subsystem is to expose some subset of the base Windows executive system services to application programs. Each subsystem can provide access to different subsets of the native services in Windows. That means that some things can be done from an application built on one subsystem that can't be done by an application built on another subsystem. For example, a Windows application can't use the POSIX *fork* function.

Each executable image (.exe) is bound to one and only one subsystem. When an image is run, the process creation code examines the subsystem type code in the image header so that it can notify the proper subsystem of the new process. This type code is specified with the /SUBSYSTEM qualifier of the *link* command in Microsoft Visual C++ and can be viewed with the Exetype tool in the Windows resource kits.

Note As a historical note, the reason the Windows subsystem process is called Csrss.exe is that in the original design of Windows NT, all the subsystems were going to execute as threads inside a single systemwide environment subsystem process. When the POSIX and OS/2 subsystems were removed and put in their own processes, the filename for the Windows subsystem process wasn't changed.

Function calls can't be mixed between subsystems. In other words, a POSIX application can call only services exported by the POSIX subsystem, and a Windows application can call only services exported by the Windows subsystem. As you'll see later, this restriction is one reason why the original POSIX subsystem, which implements a very limited set of functions (only POSIX 1003.1), wasn't a useful environment for porting UNIX applications.

As mentioned earlier, user applications don't call Windows system services directly. Instead, they go through one or more subsystem DLLs. These libraries export the documented interface that the programs linked to that subsystem can call. For example, the Windows subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) implement the Windows API functions. The POSIX subsystem DLL (Psxdll.dll) implements the POSIX API functions.

EXPERIMENT: Viewing the Image Subsystem Type

You can see the image subsystem type by using either the Exetype tool in the Windows resource kits or the Dependency Walker tool (Depends.exe) in the Windows Support Tools and Platform SDK. For example, notice the image types for two different Windows images, Notepad.exe (the simple text editor) and Cmd.exe (the Windows command prompt):

```
C:\>exetype \Windows\system32\notepad.exe
File "\Windows\system32\notepad.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows GUI subsystem

C:\>exetype \Windows\system32\cmd.exe
File "\Windows\system32\cmd.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Runs under the Windows character-based subsystem
```

This shows that Notepad is a GUI program while Cmd is a console or character-based program. And although the output of the Exetype tool implies there are two different subsystems for GUI and character-based programs, there is just one Windows subsystem. Also, Windows isn't supported on the Intel 386 processor (or the 486 for that matter)—the text output by the Exetype program hasn't been updated.

When an application calls a function in a subsystem DLL, one of three things can occur:

- The function is entirely implemented in user mode inside the subsystem DLL. In other words, no message is sent to the environment subsystem process, and no Windows executive system services are called. The function is performed in user mode, and the results are returned to the caller. Examples of such functions include *GetCurrentProcess* (which always returns -1, a value that is defined to refer to the current process in all process-related functions) and *GetCurrentProcessId*. (The process ID doesn't change for a running process, so this ID is retrieved from a cached location, thus avoiding the need to call into the kernel.)
- The function requires one or more calls to the Windows executive. For example, the Windows *ReadFile* and *WriteFile* functions involve calling the underlying internal (and undocumented) Windows I/O system services *NtReadFile* and *NtWriteFile*, respectively.
- The function requires some work to be done in the environment subsystem process. (The environment subsystem processes, running in user mode, are responsible for maintaining the state of the client applications running under their control.) In this case, a client/server request is made to the environment subsystem via a message sent to the subsystem to perform some operation. The subsystem DLL then waits for a reply before returning to the caller.

Some functions can be a combination of the second and third items just listed, such as the Windows *CreateProcess* and *CreateThread* functions.

Although Windows was designed to support multiple, independent environment subsystems, from a practical perspective, having each subsystem implement all the code to handle windowing and display I/O would result in a large amount of duplication of system functions that, ultimately, would have negatively affected both system size and performance. Because Windows was the primary subsystem, the Windows designers decided to locate these basic functions there and have the other subsystems call on the Windows subsystem to perform display I/O. Thus, the POSIX and OS/2 subsystems call services in the Windows subsystem to perform display I/O. (In fact, if you examine the subsystem type for these images, you'll see that they are Windows executables.)

Let's take a closer look at each of the environment subsystems.

Windows Subsystem

The Windows subsystem consists of the following major components:

- The environment subsystem process (Csrss.exe) contains support for:
  - Console (text) windows
  - Creating and deleting processes and threads
  - Portions of the support for 16-bit virtual DOS machine (VDM) processes

  - Other miscellaneous functions, such as *GetTempFile*, *DefineDosDevice*, *ExitWindowsEx*, and several natural language support functions
- The kernel-mode device driver (Win32k.sys) contains:
  - The window manager, which controls window displays; manages screen output; collects input from keyboard, mouse, and other devices; and passes user messages to applications.
  - The Graphics Device Interface (GDI), which is a library of functions for graphics output devices. It includes functions for line, text, and figure drawing and for graphics manipulation.
- Subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) translate documented Windows API functions into the appropriate and mostly undocumented kernel-mode system service calls to Ntoskrnl.exe and Win32k.sys.
- Graphics device drivers are hardware-dependent graphics display drivers, printer drivers, and video miniport drivers.

Applications call the standard USER functions to create user interface controls, such as windows and buttons, on the display. The window manager communicates these requests to the GDI, which passes them to the graphics device drivers, where they are formatted for the display device. A display driver is paired with a video miniport driver to complete video display support.

The GDI provides a set of standard two-dimensional functions that let applications communicate with graphics devices without knowing anything about the devices. GDI functions mediate between applications and graphics devices such as display drivers and printer drivers. The GDI interprets application requests for graphic output and sends the requests to graphics display drivers. It also provides a standard interface for applications to use varying graphics output devices. This interface enables application code to be independent of the hardware devices and their drivers. The GDI tailors its messages to the capabilities of the device, often dividing the request into manageable parts. For example, some devices can understand directions to draw an ellipse; others require the GDI to interpret the command as a series of pixels placed at certain coordinates. For more information about the graphics and video driver architecture, see the "Design Guide" section of the book *Graphics Drivers* in the Windows DDK.

Prior to Windows NT 4, the window manager and graphics services were part of the user-mode Windows subsystem process. In Windows NT 4, the bulk of the windowing and graphics code was moved from running in the context of the Windows subsystem process to a set of callable services running in kernel mode (in the file Win32k.sys). The primary reason for

this shift was to improve overall system performance. Having a separate server process that contains the Windows graphics subsystem required multiple thread and process context switches, which consumed considerable CPU cycles and memory resources even though the original design was highly optimized.

For example, for each thread on the client side there was a dedicated, paired server thread in the Windows subsystem process waiting on the client thread for requests. A special interprocess communication facility called *fast LPC* was used to send messages between these threads. Unlike normal thread context switches, transitions between paired threads via fast LPC don't cause a rescheduling event in the kernel, thereby enabling the server thread to run for the remaining time slice of the client thread before having to take its turn in the kernel's preemptive thread scheduler. Moreover, shared memory buffers were used to allow fast passing of large data structures, such as bitmaps, and clients had direct but read-only access to key server data structures to minimize the need for thread/process transitions between clients and the Windows server. Also, GDI operations were (and still are) batched. *Batching* means that a series of graphics calls by a Windows application aren't "pushed" over to the server and drawn on the output device until a GDI batching queue is filled. You can set the size of the queue by using the Windows *GdiSetBatchLimit* function, and you can flush the queue at any time with *GdiFlush*. Conversely, read-only properties and data structures of GDI, once they were obtained from the Windows subsystem process, were cached on the client side for fast subsequent access.

Despite these optimizations, however, the overall system performance was still not adequate for graphics-intensive applications. The obvious solution was to eliminate the need for the additional threads and resulting context switches by moving the windowing and graphics system into kernel mode. Also, once applications have called into the window manager and the GDI, those subsystems can access other Windows executive components directly without the cost of user-mode or kernel-mode transitions. This direct access is especially important in the case of the GDI calling through video drivers, a process that involves interaction with video hardware at high frequencies and high bandwidths.

So, what remains in the user-mode process part of the Windows subsystem? All the drawing and updating for console or text windows are handled by it because console applications have no notion of repainting a window. It's easy to see this activity—simply open a command prompt and drag another window over it, and you'll see the Windows subsystem consuming CPU time as it repaints the console window. But other than console window support, only a few Windows functions result in sending a message to the Windows subsystem process anymore: process and thread creation and termination, network drive letter mapping, and creation of temporary files. In general, a running Windows application won't be causing many, if any, context switches to the Windows subsystem process.

Is Windows Less Stable with USER and GDI in Kernel Mode?

Some people wondered whether moving this much code into kernel mode would substantially affect system stability. The reason the impact on system stability has been minimal is that prior to Windows NT 4 (and this is still true today), a bug (such as an access violation) in the user-mode Windows subsystem process (Csrss.exe) results in a system crash because the Windows subsystem process was (and still is) a vital process to the running of the system. Because it was the process that contained the data structures that described the windows on the display, the death of that process would kill the user interface. However, even a Windows system operating as a server, with no interactive processes, can't run without this process, because server processes might be using window messaging to drive the internal state of the application. With Windows, an access violation in the same code now running in kernel mode simply crashes the system more quickly, because exceptions in kernel mode result in a system crash.

There is, however, one additional theoretical danger that didn't exist prior to moving the windowing and graphics system into kernel mode. Because this body of code is now running in kernel mode, a bug (such as the use of a bad pointer) could result in corrupting kernel-mode protected data structures. Prior to Windows NT 4, such references would have caused an access violation because kernel-mode pages aren't writable from user mode. But a system crash would have then resulted, as described earlier. With the code now running in kernel mode, a bad pointer reference that caused a write operation to some kernel-mode page might not immediately cause a system crash, but if it corrupted some data structure, a crash would likely result soon after. There is a small chance, however, that such a reference could corrupt a memory buffer (rather than a data structure), possibly resulting in returning corrupt data to a user program or writing bad data to the disk.

Another area of possible impact can come from the move of the graphics drivers into kernel mode. Previously, some portions of a graphics driver ran within Csrss and others ran in kernel mode. Now, the entire driver runs in kernel mode. Although Microsoft doesn't develop all the graphics device drivers supported in Windows, it does work directly with hardware manufacturers to help ensure that they are able to produce reliable and efficient drivers. All drivers shipped with the system are submitted to the same rigorous testing as other executive components.

Finally, it's important to understand that this design (running the windowing and graphics subsystem in kernel mode) is not fundamentally risky. It is identical to the approaches many other device drivers use (for example, network card drivers and hard disk drivers). All these drivers have been operating in kernel mode since the inception of Windows NT with a high degree of reliability.

Some people speculated that the move of the window manager and the GDI into kernel mode would hurt the preemptive multitasking capability of Windows. The theory was that with all the additional Windows processing time spent in kernel mode, other

threads would have less opportunity to be run preemptively. This view was based on a misunderstanding of the Windows architecture. It is true that in many other nominally preemptive operating systems, executing in kernel mode is never preempted by the operating system scheduler—or is preempted only at a certain limited number of predefined points of kernel reentrancy. In Windows, however, threads running anywhere in the executive are preempted and scheduled alongside threads running in user mode, and all code within the executive is fully reentrant. Among other reasons, this capability is necessary to achieve a high degree of system scalability on SMP hardware.

Another line of speculation was that SMP scaling would be hurt by this change. The theory went like this: Previously, an interaction between an application and the window manager or the GDI involved two threads, one in the application and one in Csrss.exe. Therefore, on an SMP system, the two threads could run in parallel, thus improving throughput. This analysis shows a misunderstanding of how Windows NT technology worked prior to Windows NT 4. In most cases, calls from a client application to the Windows subsystem process run synchronously; that is, the client thread entirely blocks waiting on the server thread and begins to run again only when the server thread has completed the call. Therefore, no parallelism on SMP hardware can ever be achieved. This phenomenon is easily observable with a busy graphics application using the Performance tool on an SMP system. The observer will discover that on a two-processor system each processor is approximately 50 percent loaded, and it's relatively easy to find the single Csrss thread that is paired off with the busy application thread. Indeed, because the two threads are fairly intimate with each other and sharing state, the processors' caches must be flushed constantly to maintain coherency. This constant flushing is the reason that with Windows NT 3.51 a single-threaded graphics application typically runs slightly slower on an SMP machine than on a single processor system.

As a result, the changes in Windows NT 4 increased SMP throughput of applications that make heavy use of the window manager and the GDI, especially when more than one application thread is busy. When two application threads are busy on a two-processor Windows NT 3.51–based machine, a total of four threads (two in the application plus two in Csrss) are battling for time on the two processors. Although only two are typically ready to run at any given time, the lack of a consistent pattern in which threads run results in a loss of locality of reference and cache coherency. This loss occurs because the busy threads are likely to get shuffled from one processor to another. In the Windows NT 4 design, each of the two application threads essentially has its own processor, and the automatic thread affinity of Windows tends to run the same thread on the same processor indefinitely, thus maximizing locality of reference and minimizing the need to synchronize the private per-processor memory caches.

So in summary, moving the window manager and the GDI from user mode to kernel mode has provided improved performance without any significant decrease in system stability or reliability, even in the case of multiple sessions being created in a Terminal Service enabled configuration.

POSIX Subsystem

POSIX, an acronym loosely defined as "a portable operating system interface based on UNIX," refers to a collection of international standards for UNIX-style operating system interfaces. The POSIX standards encourage vendors implementing UNIX-style interfaces to make them compatible so that programmers can move their applications easily from one system to another.

Windows implements only one of the many POSIX standards, POSIX.1, formally known as ISO/IEC 9945-1:1990 or IEEE POSIX standard 1003.1-1990. This standard was included primarily to meet U.S. government procurement requirements set in the mid-to-late 1980s that mandated POSIX.1 compliance as specified in Federal Information Processing Standard (FIPS) 151-2, developed by the National Institute of Standards and Technology. Windows NT 3.5, 3.51, and 4 have been formally tested and certified according to FIPS 151-2.

Because POSIX.1 compliance was a mandatory goal for Windows, the operating system was designed to ensure that the required base system support was present to allow for the implementation of a POSIX.1 subsystem (such as the *fork* function, which is implemented in the Windows executive, and the support for hard file links in the Windows file system). However, because POSIX.1 defines a limited set of services (such as process control, interprocess communication, simple character cell I/O, and so on), the POSIX subsystem that comes with Windows 2000 isn't a complete programming environment. And because applications can't mix calls between subsystems on Windows, by default, POSIX applications are limited to the strict set of services defined in POSIX.1. This restriction means that a POSIX executable on Windows can't create a thread or a window or use remote procedure calls (RPCs) or sockets.

To address this limitation, Microsoft provides a product called Windows Services for Unix, which includes (as of version 3.5) an enhanced POSIX subsystem environment that provides nearly 2000 UNIX functions and 300 UNIX-like tools and utilities. (See *http://www.microsoft.com/windows/sfu/default.asp* for more information on Windows Services for Unix.)

This enhanced POSIX subsystem assists in porting UNIX applications to Windows. However, because the programs are still linked as POSIX executables, they cannot call Windows functions. To port UNIX applications to Windows and allow the use of Windows functions, you can purchase UNIX-to-Windows porting packages, such as the MKS Toolkit products available from Mortice Kern Systems Inc. (*www.mkssoftware.com*). With this approach, a UNIX application can be recompiled and relinked as a Windows executable and can slowly start to integrate calls to native Windows functions.

EXPERIMENT: Watching the POSIX Subsystem Start

The POSIX subsystem is configured by default to start the first time a POSIX executable is run, so you can watch it start by running a POSIX program, such as one of the POSIX utilities that comes with the Windows Services for Unix. (You can also find a small set of POSIX utilities in the \Apps\POSIX folder on the Windows 2000 resource kit tools media—they are not installed as part of the resource kit tools installation.) Follow these steps to watch the POSIX subsystem start:

1. Start a command prompt.
2. Run Process Explorer and check that the POSIX subsystem isn't already running (that is, that there's no Psxss.exe process on the system). Make sure Process Explorer is displaying the process list in tree view (by pressing Ctrl+T).
3. Run a POSIX program, such as the C Shell or Korn Shell included with Windows Services for Unix (or a POSIX tool from the Windows 2000 resource kit, such as \Apps\POSIX\Ls.exe).
4. Go back to Process Explorer and notice the new Psxss.exe process that is a child of Smss.exe (which, depending on your different highlight duration, might still be highlighted as a new process on the display).

To compile and link a POSIX application in Windows requires the POSIX headers and libraries from the Platform SDK. POSIX executables are linked against the POSIX subsystem library, Psxdll.dll. Because by default Windows is configured to start the POSIX subsystem on demand, the first time you run a POSIX application, the POSIX subsystem process (Psxss.exe) must be started. It remains running until the system reboots. (If you kill the POSIX subsystem process, you won't be able to run more POSIX applications until you reboot.) The POSIX image itself isn't run directly—instead, a special support image called Posix.exe is launched, which in turn creates a child process to run the POSIX application.

OS/2 Subsystem

The OS/2 environment subsystem, like the built-in POSIX subsystem, is fairly limited in usefulness in that it supports only OS/2 1.2 16-bit character-based or video I/O (VIO) applications. Although Microsoft did sell a replacement OS/2 1.2 Presentation Manager subsystem for Windows NT 4, it didn't support OS/2 2.*x* (or later) applications (and it isn't available for Windows 2000 or later).

Also, because Windows doesn't allow direct hardware access by user applications, OS/2 programs that contain I/O privilege segments that attempt to perform IN/OUT instructions (to access some hardware device) as well as advanced video I/O (AVIO) aren't supported. Applications that use the CLI/STI instructions are supported—but all the other OS/2 applications

in the system and all the other threads in the OS/2 process issuing the CLI instructions are suspended until an STI instruction is executed.

The 16-MB memory limitation on native OS/2 1.2 doesn't apply to Windows—the OS/2 subsystem uses the 32-bit virtual address space of Windows to provide up to 512 MB of memory to OS/2 1.2 applications, as illustrated in Figure 2-5.

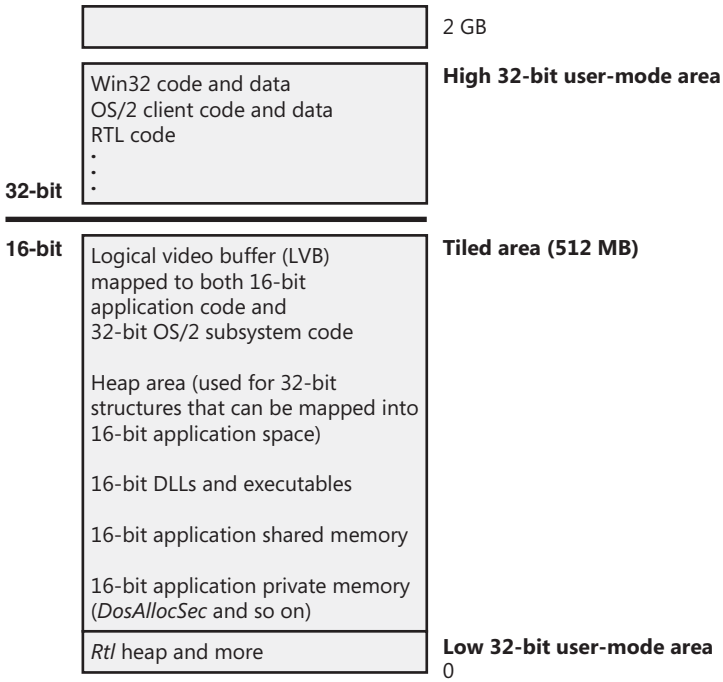


Figure 2-5 OS/2 subsystem virtual memory layout

The tiled area is 512 MB of virtual address space that is reserved up front and then committed or decommitted when 16-bit applications need segments. The OS/2 subsystem maintains a local descriptor table (LDT) for each process, with shared memory segments at the same LDT slot for all OS/2 processes.

As we'll discuss in detail in Chapter 6, threads are the elements of a program that execute, and as such they must be scheduled for processor time. Although Windows priority levels range from 0 through 31, the 64 OS/2 priority levels (0 through 63) are mapped to Windows dynamic priorities 1 through 15. OS/2 threads never receive Windows real-time priorities 16 through 31.

As with the POSIX subsystem, the OS/2 subsystem starts automatically the first time you activate a compatible OS/2 image. It remains running until the system is rebooted.

For more information on how Windows handles running POSIX and OS/2 applications, see the section "Flow of *CreateProcess*" in Chapter 6.

Ntdll.dll

Ntdll.dll is a special system support library primarily for the use of subsystem DLLs. It contains two types of functions:

- System service dispatch stubs to Windows executive system services
- Internal support functions used by subsystems, subsystem DLLs, and other native images

The first group of functions provides the interface to the Windows executive system services that can be called from user mode. There are more than 200 such functions, such as *NtCreateFile*, *NtSetEvent*, and so on. As noted earlier, most of the capabilities of these functions are accessible through the Windows API. (A number are not, however, and are for use within the operating system.)

For each of these functions, Ntdll contains an entry point with the same name. The code inside the function contains the architecture-specific instruction that causes a transition into kernel mode to invoke the system service dispatcher (explained in more detail in Chapter 3), which after verifying some parameters, calls the actual kernel-mode system service that contains the real code inside Ntoskrnl.exe.

Ntdll also contains many support functions, such as the image loader (functions that start with *Ldr*), the heap manager, and Windows subsystem process communication functions (functions that start with *Csr*), as well as general run-time library routines (functions that start with *Rtl*). It also contains the user-mode asynchronous procedure call (APC) dispatcher and exception dispatcher. (APCs and exceptions are explained in Chapter 3.)

Executive

The Windows executive is the upper layer of Ntoskrnl.exe. (The kernel is the lower layer.) The executive includes the following types of functions:

- Functions that are exported and callable from user mode. These functions are called *system services* and are exported via Ntdll. Most of the services are accessible through the Windows API or the APIs of another environment subsystem. A few services, however, aren't available through any documented subsystem function. (Examples include LPCs and various query functions such as *NtQueryInformationProcess*, specialized functions such as *NtCreatePagingFile*, and so on.)
- Device driver functions that are called through the use of the *DeviceIoControl* function. This provides a general interface from user mode to kernel mode to call functions in device drivers that are not associated with a read or write.
- Functions that can be called only from kernel mode that are exported and documented in the Windows DDK or Windows Installable File System (IFS) Kit. (For information on the Windows IFS Kit, go to *http://www.microsoft.com/whdc/ddk/ifskit/default.mspx*.)

- Functions that are exported and callable from kernel mode but are not documented in the Windows DDK or IFS Kit (such as the functions called by the boot video driver, which start with *Inbv*).
- Functions that are defined as global symbols but are not exported. These include internal support functions called within Ntoskrnl, such as those that start with *Iop* (internal I/O manager support functions) or *Mi* (internal memory management support functions).
- Functions that are internal to a module that are not defined as global symbols.

The executive contains the following major components, each of which is covered in detail in a subsequent chapter of this book:

- The *configuration manager* (explained in Chapter 4) is responsible for implementing and managing the system registry.
- The *process and thread manager* (explained in Chapter 6) creates and terminates processes and threads. The underlying support for processes and threads is implemented in the Windows kernel; the executive adds additional semantics and functions to these lower-level objects.
- The *security reference monitor* (or SRM, described in Chapter 8) enforces security policies on the local computer. It guards operating system resources, performing run-time object protection and auditing.
- The *I/O manager* (explained in Chapter 9) implements device-independent I/O and is responsible for dispatching to the appropriate device drivers for further processing.
- The *Plug and Play (PnP) manager* (explained in Chapter 9) determines which drivers are required to support a particular device and loads those drivers. It retrieves the hardware resource requirements for each device during enumeration. Based on the resource requirements of each device, the PnP manager assigns the appropriate hardware resources such as I/O ports, IRQs, DMA channels, and memory locations. It is also responsible for sending proper event notification for device changes (addition or removal of a device) on the system.
- The *power manager* (explained in Chapter 9) coordinates power events and generates power management I/O notifications to device drivers. When the system is idle, the power manager can be configured to reduce power consumption by putting the CPU to sleep. Changes in power consumption by individual devices are handled by device drivers but are coordinated by the power manager.
- The *WDM Windows Management Instrumentation routines* (explained in Chapter 4) enable device drivers to publish performance and configuration information and receive commands from the user-mode WMI service. Consumers of WMI information can be on the local machine or remote across the network.

- The *cache manager* (explained in Chapter 11) improves the performance of file-based I/O by causing recently referenced disk data to reside in main memory for quick access (and by deferring disk writes by holding the updates in memory for a short time before sending them to the disk). As you'll see, it does this by using the memory manager's support for mapped files.
- The *memory manager* (explained in Chapter 7) implements *virtual memory*, a memory management scheme that provides a large, private address space for each process that can exceed available physical memory. The memory manager also provides the underlying support for the cache manager.
- The *logical prefetcher* (explained in Chapter 7) accelerates system and process startup by optimizing the loading of data referenced during the startup of the system or a process.

In addition, the executive contains four main groups of support functions that are used by the executive components just listed. About a third of these support functions are documented in the DDK because device drivers also use them. These are the four categories of support functions:

- The *object manager*, which creates, manages, and deletes Windows executive objects and abstract data types that are used to represent operating system resources such as processes, threads, and the various synchronization objects. The object manager is explained in Chapter 3.
- The *LPC facility* (explained in Chapter 3) passes messages between a client process and a server process on the same computer. LPC is a flexible, optimized version of *remote procedure call* (RPC), an industry-standard communication facility for client and server processes across a network.
- A broad set of common *run-time library* functions, such as string processing, arithmetic operations, data type conversion, and security structure processing.
- Executive support routines, such as system memory allocation (paged and nonpaged pool), interlocked memory access, as well as two special types of synchronization objects: resources and fast mutexes.

Kernel

The kernel consists of a set of functions in Ntoskrnl.exe that provide fundamental mechanisms (such as thread scheduling and synchronization services) used by the executive components, as well as low-level hardware architecture-dependent support (such as interrupt and exception dispatching), that are different on each processor architecture. The kernel code is written primarily in C, with assembly code reserved for those tasks that require access to specialized processor instructions and registers not easily accessible from C.

Like the various executive support functions mentioned in the preceding section, a number of functions in the kernel are documented in the DDK (and can be found by searching for functions beginning with *Ke*) because they are needed to implement device drivers.

Kernel Objects

The kernel provides a low-level base of well-defined, predictable operating system primitives and mechanisms that allow higher-level components of the executive to do what they need to do. The kernel separates itself from the rest of the executive by implementing operating system mechanisms and avoiding policy making. It leaves nearly all policy decisions to the executive, with the exception of thread scheduling and dispatching, which the kernel implements.

Outside the kernel, the executive represents threads and other shareable resources as objects. These objects require some policy overhead, such as object handles to manipulate them, security checks to protect them, and resource quotas to be deducted when they are created. This overhead is eliminated in the kernel, which implements a set of simpler objects, called *kernel objects*, that help the kernel control central processing and support the creation of executive objects. Most executive-level objects encapsulate one or more kernel objects, incorporating their kernel-defined attributes.

One set of kernel objects, called *control objects*, establishes semantics for controlling various operating system functions. This set includes the APC object, the *deferred procedure call* (DPC) object, and several objects the I/O manager uses, such as the interrupt object.

Another set of kernel objects, known as *dispatcher objects*, incorporates synchronization capabilities that alter or affect thread scheduling. The dispatcher objects include the kernel thread, mutex (called *mutant* internally), event, kernel event pair, semaphore, timer, and waitable timer. The executive uses kernel functions to create instances of kernel objects, to manipulate them, and to construct the more complex objects it provides to user mode. Objects are explained in more detail in Chapter 3, and processes and threads are described in Chapter 6.

Hardware Support

The other major job of the kernel is to abstract or isolate the executive and device drivers from variations between the hardware architectures supported by Windows. This job includes handling variations in functions such as interrupt handling, exception dispatching, and multiprocessor synchronization.

Even for these hardware-related functions, the design of the kernel attempts to maximize the amount of common code. The kernel supports a set of interfaces that are portable and semantically identical across architectures. Most of the code that implements this portable interface is also identical across architectures.

Some of these interfaces are implemented differently on different architectures, however, or some of the interfaces are partially implemented with architecture-specific code. These

architecturally independent interfaces can be called on any machine, and the semantics of the interface will be the same whether or not the code varies by architecture. Some kernel interfaces (such as spinlock routines, which are described in Chapter 3) are actually implemented in the HAL (described in the next section) because their implementation can vary for systems within the same architecture family.

The kernel also contains a small amount of code with x86-specific interfaces needed to support old MS-DOS programs. These x86 interfaces aren't portable in the sense that they can't be called on a machine based on any other architecture; they won't be present. This x86-specific code, for example, supports calls to manipulate global descriptor tables (GDTs) and LDTs, hardware features of the x86.

Other examples of architecture-specific code in the kernel include the interface to provide translation buffer and CPU cache support. This support requires different code for the different architectures because of the way caches are implemented.

Another example is context switching. Although at a high level the same algorithm is used for thread selection and context switching (the context of the previous thread is saved, the context of the new thread is loaded, and the new thread is started), there are architectural differences among the implementations on different processors. Because the context is described by the processor state (registers and so on), what is saved and loaded varies depending on the architecture.

Hardware Abstraction Layer

As mentioned at the beginning of this chapter, one of the crucial elements of the Windows design is its portability across a variety of hardware platforms. The hardware abstraction layer (HAL) is a key part of making this portability possible. The HAL is a loadable kernel-mode module (Hal.dll) that provides the low-level interface to the hardware platform on which Windows is running. It hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms—any functions that are both architecture-specific and machine-dependent.

So rather than access hardware directly, Windows internal components as well as user-written device drivers maintain portability by calling the HAL routines when they need platform-dependent information. For this reason, the HAL routines are documented in the Windows DDK. To find out more about the HAL and its use by device drivers, refer to the DDK.

Although several HALs are included with Windows (as shown in Table 2-6), only one is chosen at installation time and copied to the system disk with the filename Hal.dll. (Other operating systems, such as VMS, select the equivalent of the HAL at system boot time.) Therefore, you can't assume that a system disk from one x86 installation will boot on a different processor if the HAL that supports the other processor is different.

Table 2-6 List of x86 HALs in \Windows\Driver Cache\i386\Driver.cab

HAL File Name	Systems Supported
Hal.dll	Standard PCs
Halacpi.dll	Advanced Configuration and Power Interface (ACPI) PCs
Halapic.dll	Advanced Programmable Interrupt Controller (APIC) PCs
Halaacpi.dll	APIC ACPI PCs
Halmps.dll	Multiprocessor PCs
Halmacpi.dll	Multiprocessor ACPI PCs
Halborg.dll	Silicon Graphics Workstation (Windows 2000 only; platform no longer marketed)
Halsp.dll	Compaq SystemPro (Windows XP only)

Note As of Windows Server 2003, no vendor-specific HALs are shipped with the base system.

EXPERIMENT: Viewing the Base HALs Included with Windows

To view the HALs included with Windows, open the file Driver.cab in the appropriate architecture-specific folder underneath \Windows\Driver Cache. (For example, for x86 systems, the file name is \Windows\Driver Cache\i386\Driver.cab.) Scroll down to the files beginning with "Hal" and you should see the files listed in Table 2-6.

EXPERIMENT: Determining Which HAL You're Running

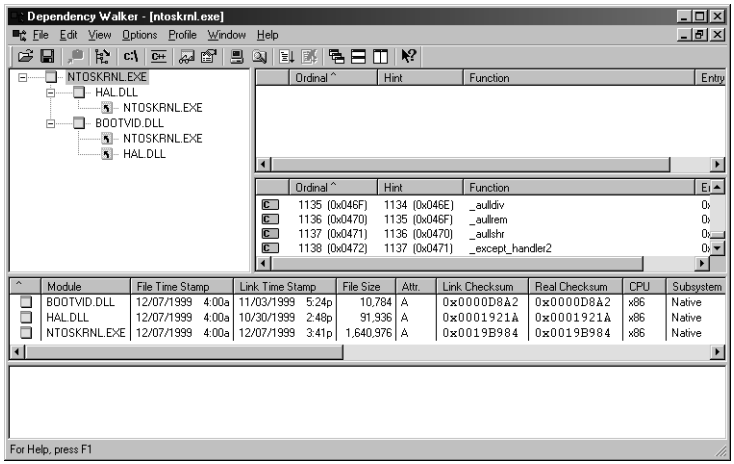
There are two ways to determine which HAL you're running:

1. Open the file \Windows\Repair\Setup.log, search for *Hal.dll*, and look at the filename after the equals sign. This is the name of the HAL on the distribution media extracted from Driver.cab.
2. In Device Manager (right-click on the My Computer icon on your desktop, select Properties, click on the Hardware tab, and then click Device Manager), look at the name of the "driver" under the Computer device type. For example, the following screen shot is from a system running the ACPI HAL:

EXPERIMENT: Viewing NTOSKRNL and HAL Image Dependencies

You can view the relationship of the kernel and HAL images by examining their export and import tables using the Dependency Walker tool (Depends.exe), which is contained in the Windows Support Tools and the Platform SDK. To examine an image in the Dependency Walker, select Open from the File menu to open the desired image file.

Here is a sample of output you can see by viewing the dependencies of Ntoskrnl using this tool:



Notice that Ntoskrnl is linked against the HAL, which is in turn linked against Ntoskrnl. (They both use functions in each other.) Ntoskrnl is also linked against Bootvid.dll, the boot video driver that is used to implement the GUI startup screen. On Windows XP and later, you will see an additional DLL, Kdcom.dll, in the list. This contains kernel debugger infrastructure code that used to be part of Ntoskrnl.exe.

For a detailed description of the information displayed by this tool, see the Dependency Walker help file (Depends.hlp).

Device Drivers

Although device drivers are explained in detail in Chapter 9, this section provides a brief overview of the types of drivers and explains how to list the drivers installed and loaded on your system.

Device drivers are loadable kernel-mode modules (typically ending in .sys) that interface between the I/O manager and the relevant hardware. They run in kernel mode in one of three contexts:

- In the context of the user thread that initiated an I/O function
- In the context of a kernel-mode system thread
- As a result of an interrupt (and therefore not in the context of any particular process or thread—whichever process or thread was current when the interrupt occurred)

As stated in the preceding section, device drivers in Windows don't manipulate hardware directly, but rather they call functions in the HAL to interface with the hardware. Drivers are typically written in C (sometimes C++) and therefore, with proper use of HAL routines, can be source code portable across the CPU architectures supported by Windows and binary portable within an architecture family.

There are several types of device drivers:

- *Hardware device drivers* manipulate hardware (using the HAL) to write output to or retrieve input from a physical device or network. There are many types of hardware device drivers, such as bus drivers, human interface drivers, mass storage drivers, and so on.
- *File system drivers* are Windows drivers that accept file-oriented I/O requests and translate them into I/O requests bound for a particular device.
- *File system filter drivers*, such as those that perform disk mirroring and encryption, intercept I/Os and perform some added-value processing before passing the I/O to the next layer.
- *Network redirectors and servers* are file system drivers that transmit file system I/O requests to a machine on the network and receive such requests, respectively.
- *Protocol drivers* implement a networking protocol such as TCP/IP, NetBEUI, and IPX/SPX.
- *Kernel streaming filter drivers* are chained together to perform signal processing on data streams, such as recording or displaying audio and video.

Because installing a device driver is the only way to add user-written kernel-mode code to the system, some programmers have written device drivers simply as a way to access internal operating system functions or data structures that are not accessible from user mode (but that are documented and supported in the DDK). For example, many of the utilities from *www.sysinternals.com* combine a Windows GUI application and a device driver that is used to gather internal system state and call kernel-mode-only accessible functions not accessible from the user-mode Windows API.

Windows Driver Model (WDM)

Windows 2000 added support for Plug and Play, Power Options, and an extension to the Windows NT driver model called the Windows Driver Model (WDM). Windows 2000 and later can run legacy Windows NT 4 drivers, but because these don't support Plug and Play and Power Options, systems running these drivers will have reduced capabilities in these two areas.

From the WDM perspective, there are three kinds of drivers:

- A *bus driver* services a bus controller, adapter, bridge, or any device that has child devices. Bus drivers are required drivers, and Microsoft generally provides them; each type of bus (such as PCI, PCMCIA, and USB) on a system has one bus driver. Third parties can write bus drivers to provide support for new buses, such as VMEbus, Multibus, and Futurebus.
- A *function driver* is the main device driver and provides the operational interface for its device. It is a required driver unless the device is used raw (an implementation in which I/O is done by the bus driver and any bus filter drivers, such as SCSI PassThru). A function driver is by definition the driver that knows the most about a particular device, and it is usually the only driver that accesses device-specific registers.
- A *filter driver* is used to add functionality to a device (or existing driver) or to modify I/O requests or responses from other drivers (and is often used to fix hardware that provides incorrect information about its hardware resource requirements). Filter drivers are optional and can exist in any number, placed above or below a function driver and above a bus driver. Usually, system original equipment manufacturers (OEMs) or independent hardware vendors (IHVs) supply filter drivers.

In the WDM driver environment, no single driver controls all aspects of a device: a bus driver is concerned with reporting the devices on its bus to the PnP manager, while a function driver manipulates the device.

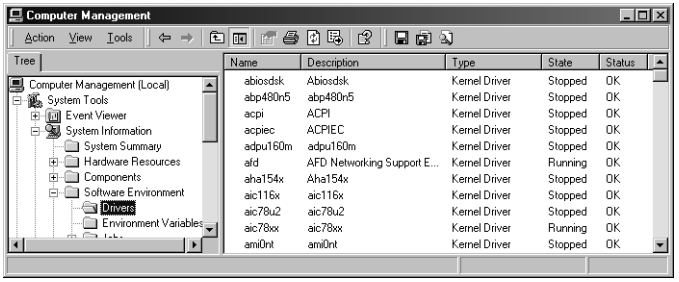
In most cases, lower-level filter drivers modify the behavior of device hardware. For example, if a device reports to its bus driver that it requires four I/O ports when it actually requires 16 I/O ports, a lower-level device-specific function filter driver could intercept the list of hardware resources reported by the bus driver to the PnP manager, and update the count of I/O ports.

Upper-level filter drivers usually provide added-value features for a device. For example, an upper-level device filter driver for a keyboard can enforce additional security checks.

Interrupt processing is explained in Chapter 3. Further details about the I/O manager, WDM, Plug and Play, and Power Options are included in Chapter 9.

EXPERIMENT: Viewing the Installed Device Drivers

You can list the installed drivers by running Computer Management. (From the Start menu, select Programs, Administrative Tools, and then Computer Management; or from Control Panel, open Administrative Tools and select Computer Management.) From within Computer Management, expand System Information and then Software Environment, and open Drivers. Here's an example output of the list of installed drivers:



This window displays the list of device drivers defined in the registry, their type, and their state (Running or Stopped). Device drivers and Windows service processes are both defined in the same place: HKLM\ SYSTEM\CurrentControlSet\Services. However, they are distinguished by a type code—for example, type 1 is a kernel-mode device driver. (For a complete list of the information stored in the registry for device drivers, see Table 4-7.)

Alternatively, you list the currently loaded device drivers with the Drivers utility (Drivers.exe in the Windows 2000 resource kits) or the Pstat utility (Pstat.exe in the Windows XP Support Tools, Windows Server 2003 Support Tools, Windows 2000 resource kits, and the Platform SDK). Here is a partial output from the Drivers utility:

```
C:\>drivers
  ModuleName    Code    Data  Bss    Paged    Init          LinkDate
-------------------------------------------------------------------
ntoskrnl.exe 429184   96896    0   775360  138880  Tue Dec 07 18:41:11 1999
     hal.dll  25856    6016    0    16160   10240  Tue Nov 02 20:14:22 1999
 BOOTVID.DLL   5664    2464    0        0     320  Wed Nov 03 20:24:33 1999
    ACPI.sys  92096    8960    0    43488    4448  Wed Nov 10 20:06:04 1999
  WMILIB.SYS    512       0    0     1152     192  Sat Sep 25 14:36:47 1999
     pci.sys  12704    1536    0    31264    4608  Wed Oct 27 19:11:08 1999
  isapnp.sys  14368     832    0    22944    2048  Sat Oct 02 16:00:35 1999
compbatt.sys   2496       0    0     2880    1216  Fri Oct 22 18:32:49 1999
   BATTC.SYS    800       0    0     2976     704  Sun Oct 10 19:45:37 1999
intelide.sys   1760      32    0        0     128  Thu Oct 28 19:20:03 1999
 PCIIDEX.SYS   4544     480    0    10944    1632  Wed Oct 27 19:02:19 1999
  pcmcia.sys  32800    8864    0    23680    6240  Fri Oct 29 19:20:08 1999
  ftdisk.sys   4640      32    0    95072    3392  Mon Nov 22 14:36:23 1999
-------------------------------------------------------------------
      Total 4363360  580320  0 3251424  432992
```

Each loaded kernel-mode component (Ntoskrnl, the HAL, as well as device drivers) is shown, along with the sizes of the sections in each image.

The Pstat utility also shows the loaded driver list, but only after it first displays the process list and the threads in each process. Pstat includes one important piece of information that the Drivers utility doesn't: the load address of the module in system space. As we'll explain later, this address is needed to map running system threads to the device driver in which they exist.

Peering into Undocumented Interfaces

Examining the names of the exported or global symbols in key system images (such as Ntoskrnl.exe, Hal.dll, or Ntdll.dll) can be enlightening—you can get an idea of the kinds of things Windows can do versus what happens to be documented and supported today. Of course, just because you know the names of these functions doesn't mean that you can or should call them—the interfaces are undocumented and are subject to change. We suggest that you look at these functions purely to gain more insight into the kinds of internal functions Windows performs, not to bypass supported interfaces.

For example, looking at the list of functions in Ntdll.dll gives you the list of all the system services that Windows provides to user-mode subsystem DLLs versus the subset that each subsystem exposes. Although many of these functions map clearly to documented and supported Windows functions, several are not exposed via the Windows API. (See the article "Inside the Native API" from *www.sysinternals.com*.)

Conversely, it's also interesting to examine the imports of Windows subsystem DLLs (such as Kernel32.dll or Advapi32.dll) and which functions they call in Ntdll.

Another interesting image to dump is Ntoskrnl.exe—although many of the exported routines that kernel-mode device drivers use are documented in the Windows DDK, quite a few are not. You might also find it interesting to take a look at the import table for Ntoskrnl and the HAL; this table shows the list of functions in the HAL that Ntoskrnl uses and vice versa.

Table 2-7 lists most of the commonly used function name prefixes for the executive components. Each of these major executive components also uses a variation of the prefix to denote internal functions—either the first letter of the prefix followed by an *i* (for *internal*) or the full prefix followed by a *p* (for *private*). For example, *Ki* represents internal kernel functions, and *Psp* refers to internal process support functions.

Table 2-7 Commonly Used Prefixes

Prefix	Component
Cc	Cache manager
Cm	Configuration manager
Ex	Executive support routines
FsRtl	File system driver run-time library
Hal	Hardware abstraction layer
Io	I/O manager
Ke	Kernel
Lpc	Local procedure call
Lsa	Local security authentication
Mm	Memory manager
Nt	Windows system services (most of which are exported as Windows functions)
Ob	Object manager
Po	Power manager
Pp	PnP manager
Ps	Process support
Rtl	Run-time library
Se	Security
Wmi	Windows Management Instrumentation
Zw	Mirror entry point for system services (beginning with *Nt*) that sets previous access mode to kernel, which eliminates parameter validation, because *Nt* system services validate parameters only if previous access mode is user

You can decipher the names of these exported functions more easily if you understand the naming convention for Windows system routines. The general format is:

<Prefix><Operation><Object>

In this format, *Prefix* is the internal component that exports the routine, *Operation* tells what is being done to the object or resource, and *Object* identifies what is being operated on.

For example, *ExAllocatePoolWithTag* is the executive support routine to allocate from a paged or nonpaged pool. *KeInitializeThread* is the routine that allocates and sets up a kernel thread object.

System Processes

The following system processes appear on every Windows system. (Two of these—Idle and System—are not full processes, as they are not running a user-mode executable.)

- Idle process (contains one thread per CPU to account for idle CPU time)
- System process (contains the majority of the kernel-mode system threads)
- Session manager (Smss.exe)
- Windows subsystem (Csrss.exe)
- Logon process (Winlogon.exe)
- Service control manager (Services.exe) and the child service processes it creates (such as the system-supplied generic service host process, Svchost.exe)
- Local security authentication server (Lsass.exe)

To understand the relationship of these processes, it is helpful to view the process "tree"—that is, the parent/child relationship between processes. Seeing which process created each process helps to understand where each process comes from. Figure 2-6 is a partial screen snapshot of the process tree with comments put on the first few system processes. (Process Explorer allows you to add a comment for individual processes and optionally display that as a column on the display.)

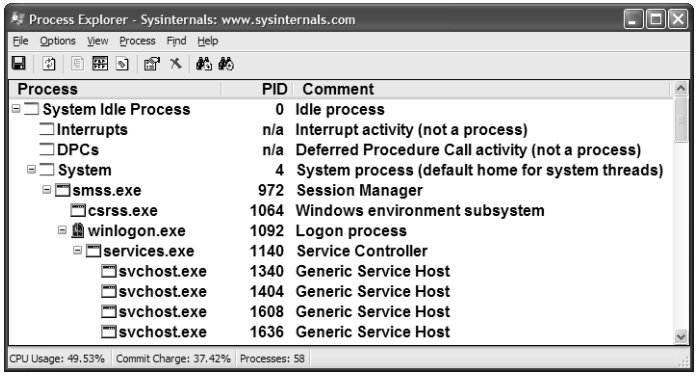


Figure 2-6 Initial System Process Tree

The next sections explain the key system processes shown in Figure 2-6. Although these sections briefly indicate the order of process startup, Chapter 5 contains a detailed description of the steps involved in booting and starting Windows.

Idle Process

The first process listed in Figure 2-6 is the system idle process. As we'll explain in Chapter 6, processes are identified by their image name. However, this process (as well as the process

named System) isn't running a real user-mode image (in that there is no "System Idle Process.exe" in the \Windows directory). In addition, the name shown for this process differs from utility to utility (because of implementation details). Table 2-8 lists several of the names given to the Idle process (process ID 0). The Idle process is explained in detail in Chapter 6.

Table 2-8 Names for Process ID 0 in Various Utilities

Utility	Name for Process ID 0
Task Manager	System Idle Process
Process Viewer (Pviewer.exe)	Idle
Process Status (Pstat.exe)	Idle Process
Process Explode (Pview.exe)	System Process
Task List (Tlist.exe)	System Process
QuickSlice (Qslice.exe)	Systemprocess

Now let's look at system threads and the purpose of each of the system processes that are running real images.

Interrupts and DPCs

The two lines labeled Interrupts and DPCs represent time spent servicing interrupts and deferred procedure calls. These mechanisms are explained in Chapter 3. Note that while Process Explorer displays these as entries in the process list, they are not processes. They are shown because they account for CPU time not charged to any process. (For example, a system with heavy interrupt activity will not appear as a process consuming CPU time.) Note that Task Manager includes interrupt and DPC time in the system idle time. Thus a system with heavy interrupt activity will appear to be idle when using Task Manager.

System Process and System Threads

The System process (process ID 8 in Windows 2000 and process ID 4 in Windows XP and Windows Server 2003) is the home for a special kind of thread that runs only in kernel mode: a *kernel-mode system thread*. System threads have all the attributes and contexts of regular user-mode threads (such as a hardware context, priority, and so on) but are different in that they run only in kernel-mode executing code loaded in system space, whether that is in Ntoskrnl.exe or in any other loaded device driver. In addition, system threads don't have a user process address space and hence must allocate any dynamic storage from operating system memory heaps, such as a paged or nonpaged pool.

System threads are created by the *PsCreateSystemThread* function (documented in the DDK), which can be called only from kernel mode. Windows as well as various device drivers create system threads during system initialization to perform operations that require thread context, such as issuing and waiting for I/Os or other objects or polling a device. For example, the memory manager uses system threads to implement such functions as writing dirty pages to the page file or mapped files, swapping processes in and out of memory, and so forth. The ker-

nel creates a system thread called the *balance set manager* that wakes up once per second to possibly initiate various scheduling and memory management–related events. The cache manager also uses system threads to implement both read-ahead and write-behind I/Os. The file server device driver (Srv.sys) uses system threads to respond to network I/O requests for file data on disk partitions shared to the network. Even the floppy driver has a system thread to poll the floppy device. (Polling is more efficient in this case because an interrupt-driven floppy driver consumes a large amount of system resources.) Further information on specific system threads is included in the chapters in which the component is described.

By default, system threads are owned by the System process, but a device driver can create a system thread in any process. For example, the Windows subsystem device driver (Win32k.sys) creates system threads in the Windows subsystem process (Csrss.exe) so that they can easily access data in the user-mode address space of that process.

When you're troubleshooting or going through a system analysis, it's useful to be able to map the execution of individual system threads back to the driver or even to the subroutine that contains the code. For example, on a heavily loaded file server, the System process will likely be consuming considerable CPU time. But the knowledge that when the System process is running "some system thread" is running isn't enough to determine which device driver or operating system component is running.

So if threads in the System process are running, first determine which ones are running (for example, with the Performance tool). Once you find the thread (or threads) that is running, look up in which driver the system thread began execution (which at least tells you which driver likely created the thread) or examine the call stack (or at least the current address) of the thread in question, which would indicate where the thread is currently executing.

Both of these techniques are illustrated in the following experiments.

EXPERIMENT: Identifying System Threads in the System Process

You can see that the threads inside the System process must be kernel-mode system threads because the start address for each thread is greater than the start address of system space (which by default begins at 0x80000000, unless the system was booted with the /3GB Boot.ini switch). Also, if you look at the CPU time for these threads, you'll see that those that have accumulated any CPU time have run only in kernel mode.

To find out which driver created the system thread, look up the start address of the thread (which you can display with Pviewer.exe) and look for the driver whose base address is closest to (but before) the start address of the thread. Both the Pstat utility (at the end of its output) as well as the *!drivers* kernel debugger command list the base address of each loaded device driver.

To quickly find the current address of the thread, use the *!stacks 0* command in the kernel debugger. Here is sample output from a live system (using LiveKd):

```
kd> !stacks 0
Proc.Thread  Thread  ThreadState  Blocker
                                  [System]
 8.000004  8146edb0  BLOCKED  ntoskrnl!MmZeroPageThread+0x5f
 8.00000c  8146e730  BLOCKED  ?? Kernel stack not resident ??
 8.000010  8146e4b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000014  8146d030  BLOCKED  ?? Kernel stack not resident ??
 8.000018  8146ddb0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00001c  8146db30  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000020  8146d8b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000024  8146d630  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000028  8146d3b0  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.00002c  8146c030  BLOCKED  ntoskrnl!ExpWorkerThread+0x73
 8.000030  8146cdb0  BLOCKED  ntoskrnl!ExpWorkerThreadBalanceManager+0x55
 8.000034  8146b470  BLOCKED  ntoskrnl!MiDereferenceSegmentThread+0x44
 8.000038  8146b1f0  BLOCKED  ntoskrnl!MiModifiedPageWriterWorker+0x31
 8.00003c  8146a030  BLOCKED  ntoskrnl!KeBalanceSetManager+0x7e
 8.000040  8146adb0  BLOCKED  ntoskrnl!KeSwapProcessOrStack+0x24
 8.000044  8146a5b0  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
 8.000048  8146a330  BLOCKED  ntoskrnl!FsRtlWorkerThread+0x33
 8.00004c  81461030  BLOCKED  ACPI!ACPIWorker+0x46
 8.000050  8143a770  BLOCKED  ntoskrnl!MiMappedPageWriter+0x4d
 8.000054  81439730  BLOCKED  dmio!voliod_loop+0x399
 8.000058  81436c90  BLOCKED  NDIS!ndisWorkerThread+0x22
 8.00005c  813d9170  BLOCKED  ltmdmntt!WakeupTimerThread+0x27
 8.000060  813d8030  BLOCKED  ltmdmntt!WriteRegistryThread+0x1c
 8.000070  8139c850  BLOCKED  raspptp!MainPassiveLevelThread+0x78
 8.000074  8139c5d0  BLOCKED  raspptp!PacketWorkingThread+0xc0
 8.00006c  81384030  BLOCKED  rasacd!AcdNotificationRequestThread+0xd8
 8.000080  81333330  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
 8.000084  813330b0  BLOCKED  rdbss!RxSpinUpRequestsDispatcher+0x58
 8.00008c  81321db0  BLOCKED  ?? Kernel stack not resident ??
 8.00015c  81205570  BLOCKED  INO_FLTR+0x68bd
 8.000160  81204570  BLOCKED  INO_FLTR+0x80e9
 8.000178  811fcdb0  BLOCKED  irda!RxThread+0xfa
 8.0002d0  811694f0  BLOCKED  ?? Kernel stack not resident ??
 8.0002d4  81168030  BLOCKED  ?? Kernel stack not resident ??
 8.000404  811002b0  BLOCKED  rdbss!RxpWorkerThreadDispatcher+0x6f
 8.000430  810f4990  READY    parallel!ParallelThread+0x3e
 8.00069c  80993030  READY    rdbss!RxpWorkerThreadDispatcher+0x6f
```

The first column is the process ID and thread ID (in the form "process ID.thread ID"). The second column is the current address of the thread. The third column indicates whether the thread is in a wait state, ready state, or running state. (See Chapter 6 for a description of thread states.) The last column is the top-most address on the thread's stack. The information in this last column makes it easy to see which driver each thread started in. For the threads in Ntoskrnl, the name of the function gives a further indication of what the thread is doing.

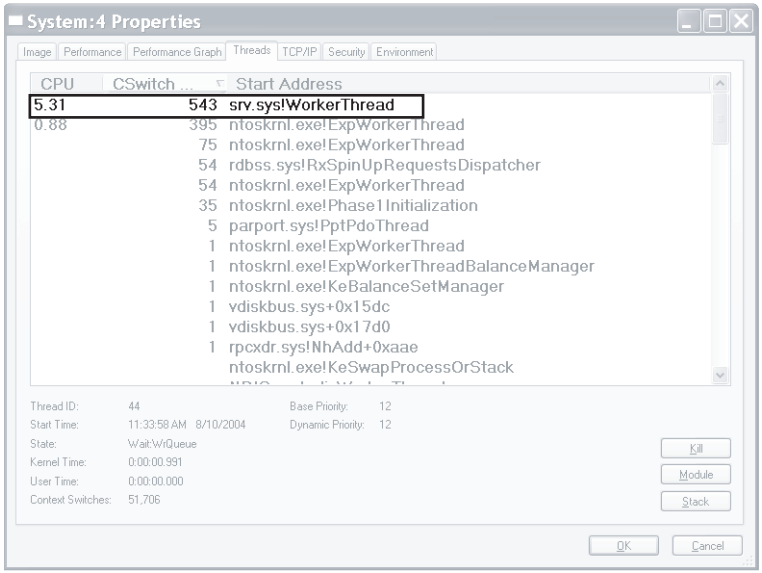
However, if the thread running is one of the system worker threads (ExpWorkerThread), you still don't really know what the thread is doing because any device driver can submit work to a system worker thread. Therefore, the only way to trace back worker thread activity is to set a breakpoint at ExQueueWorkItem. When you reach the breakpoint, type **!dso**

work_queue_item esp+4. This command will dump the first argument to ExQueueWorkItem (a work queue structure), which in turn contains the address of the worker routine to be called in the context of the worker thread. Alternatively, you can look at the caller by using the *k* command in the kernel debugger, which displays the current call stack. The current call stack will show the driver that is queuing the work to the worker thread (as opposed to the routine to be called from the worker thread).

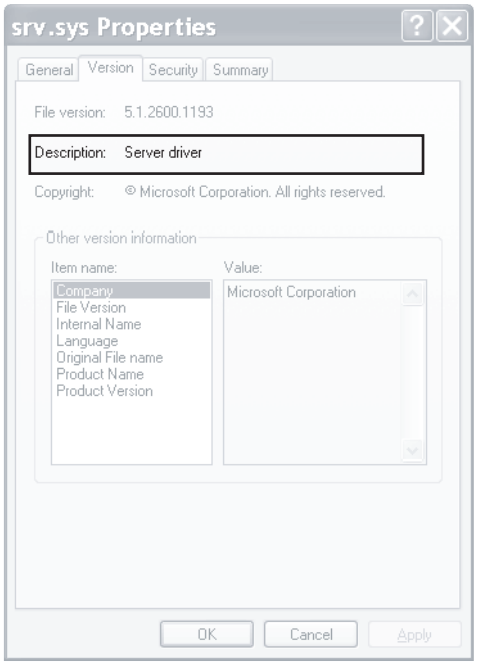
EXPERIMENT: Mapping a System Thread to a Device Driver

In this experiment, we'll see how to map CPU activity in the System process to the responsible system thread (and the driver it falls in) generating the activity. This is important because when the System process is running, you must go to the thread granularity to really understand what's going on. For this experiment, we will generate system thread activity by generating file server activity on your machine. (The file server driver, Srv.sys, creates system threads to handle inbound requests for file I/O. See Chapter 13 for more information on this component.)

1. Open a command prompt.
2. Do a directory listing of your entire C drive using a network path to access your C drive. For example, if your computer name is COMPUTER1, type "**dir \\computer1\c$ /s**". (The /s switch lists all subdirectories.)
3. Run Process Explorer, and double-click on the System process.
4. Click on the Threads tab.
5. Sort by the CSwitch Delta (context switch delta) column. You should see one or more threads in Srv.sys running such as the following:



If you see a system thread running and you are not sure what the driver is, press the Module button, which will bring up the file properties. Pressing the Module button while highlighting the thread in Srv.sys previously shown results in the following display:



Session Manager (Smss)

The Session Manager (\Windows\System32\Smss.exe) is the first user-mode process created in the system. The kernel-mode system thread that performs the final phase of the initialization of the executive and kernel creates the actual Smss process.

The Session Manager is responsible for a number of important steps in starting Windows, such as opening additional page files, performing delayed file rename and delete operations, and creating system environment variables. It also launches the subsystem processes (normally just Csrss.exe) and the Winlogon process, which in turn creates the rest of the system processes.

Much of the configuration information in the registry that drives the initialization steps of Smss can be found under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager. Some of these are explained in Chapter 5 in the section on Smss. (For a more complete description of the keys and values, see the Registry Entries help file, Regentry.chm, in the Windows 2000 resource kits.)

After performing these initialization steps, the main thread in Smss waits forever on the process handles to Csrss and Winlogon. If either of these processes terminates unexpectedly, Smss crashes the system (using the crash code STATUS_SYSTEM_PROCESS_TERMINATED, or 0xC000021A), because Windows relies on their existence. Meanwhile, Smss waits for

requests to load subsystems, debug events, and requests to create new terminal server sessions. (For a description of terminal services, see the section "Terminal Services and Multiple Sessions" in Chapter 1.)

Terminal Services session creation is performed by Smss. When a request comes in to Smss to create a session, it first calls NtSetSystemInformation with a request to set up kernel-mode session data structures. This in turn calls the internal memory manager function **MmSessionCreate**, which sets up the session virtual address space that will contain the session paged pool and the per-session data structures allocated by the kernel-mode part of the Win32 subsystem (Win32k.sys) and other session-space device drivers. (See Chapter 7 for more details.) Smss then creates an instance of Winlogon and Csrss for the session.

Winlogon, LSASS and Userinit

The Windows logon process (\Windows\System32\Winlogon.exe) handles interactive user logons and logoffs. Winlogon is notified of a user logon request when the *secure attention sequence* (SAS) keystroke combination is entered. The default SAS on Windows is the combination Ctrl+Alt+Delete. The reason for the SAS is to protect users from password-capture programs that simulate the logon process, because this keyboard sequence cannot be intercepted by a user mode application.

The identification and authentication aspects of the logon process are implemented in a replaceable DLL named GINA (Graphical Identification and Authentication). The standard Windows GINA, Msgina.dll, implements the default Windows logon interface. However, developers can provide their own GINA DLL to implement other identification and authentication mechanisms in place of the standard Windows username/password method (such as one based on a voice print). In addition, Winlogon can load additional network provider DLLs that need to perform secondary authentication. This capability allows multiple network providers to gather identification and authentication information all at one time during normal logon.

Once the username and password have been captured, they are sent to the local security authentication server process (\Windows\System32\Lsass.exe, described in Chapter 8) to be authenticated. LSASS calls the appropriate authentication package (implemented as a DLL) to perform the actual verification, such as checking whether a password matches what is stored in the active directory or the SAM (the part of the registry that contains the definition of the users and groups).

Upon a successful authentication, LSASS calls a function in the security reference monitor (for example, *NtCreateToken*) to generate an access token object that contains the user's security profile. This access token is then used by Winlogon to create the initial process(es) in the user's session. The initial process(es) are stored in the registry value *Userinit* under the registry key HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. (The default is Userinit.exe, but there can be more than one image in the list.)

Userinit performs some initialization of the user environment (such as running the login script and applying group policies) and then looks in the registry at the *Shell* value (under the same Winlogon key referred to previously) and creates a process to run the system-defined shell (by

default, Explorer.exe). Then Userinit exits. This is the reason Explorer.exe is shown with no parent—its parent has exited, and as explained earlier, tlist left-justifies processes whose parent isn't running. (Another way of looking at it is that Explorer is the grandchild of Winlogon.)

Winlogon is active not only during user logon and logoff but also whenever it intercepts the SAS from the keyboard. For example, when you press Ctrl+Alt+Delete while logged in, the Windows Security dialog box comes up, providing the options to log off, start the Task Manager, lock the workstation, shut down the system, and so forth. Winlogon is the process that handles this interaction.

For a complete description of the steps involved in the logon process, see the section "Smss, Csrss, and Winlogon" in Chapter 5. For more details on security authentication, see Chapter 8. For details on the callable functions that interface with LSASS (the functions that start with **Lsa**), see the documentation in the Platform SDK.

Service Control Manager (SCM)

Recall from earlier in the chapter that "services" on Windows can refer either to a server process or to a device driver. This section deals with services that are user-mode processes. Services are like UNIX "daemon processes" or VMS "detached processes" in that they can be configured to start automatically at system boot time without requiring an interactive logon. They can also be started manually (such as by running the Services administrative tool or by calling the Windows *StartService* function). Typically, services do not interact with the logged-on user, although there are special conditions when this is possible. (See Chapter 4.)

The service control manager is a special system process running the image \Windows\System32\Services.exe that is responsible for starting, stopping, and interacting with service processes. Service programs are really just Windows images that call special Windows functions to interact with the service control manager to perform such actions as registering the service's successful startup, responding to status requests, or pausing or shutting down the service. Services are defined in the registry under HKLM\SYSTEM\CurrentControlSet\Services. The resource kit Registry Entries help file (Regentry.chm) documents the subkeys and values for services.

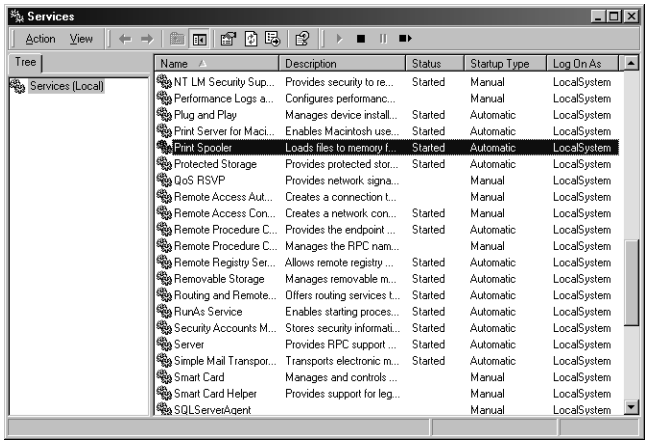
Keep in mind that services have three names: the process name you see running on the system, the internal name in the registry, and the display name shown in the Services administrative tool. (Not all services have a display name—if a service doesn't have a display name, the internal name is shown.) With Windows, services can also have a description field that further details what the service does.

To map a service process to the services contained in that process, use the *tlist /s* command. Note that there isn't always one-to-one mapping between service process and running services, however, because some services share a process with other services. In the registry, the type code indicates whether the service runs in its own process or shares a process with other services in the image.

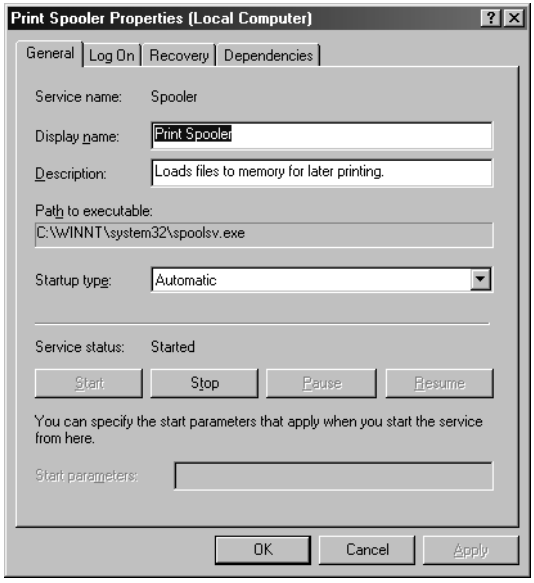
A number of Windows components are implemented as services, such as the Spooler, Event Log, Task Scheduler, and various networking components.

EXPERIMENT: Listing Installed Services

To list the installed services, select Administrative Tools from Control Panel, and then select Services. You should see output like this:



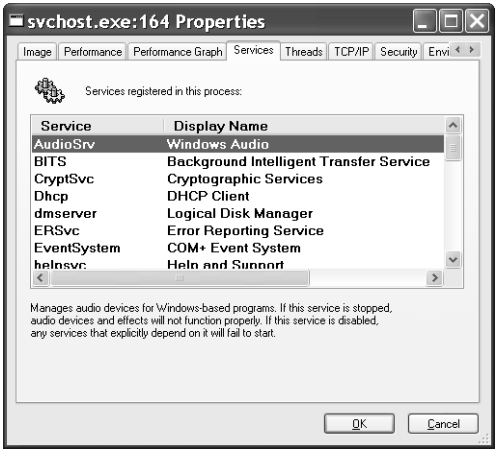
To see the detailed properties about a service, right-click on a service and select Properties. For example, here are the properties for the Print Spooler service (highlighted in the previous figure):



Notice that the Path To Executable field identifies the program that contains this service. Remember that some services share a process with other services—mapping isn't always one to one.

EXPERIMENT: Viewing Service Details Inside Service Processes

Process Explorer highlights processes hosting one service or more. (You can configure this by selecting the Configure Highlighting entry in the Options menu.) If you double-click on a service-hosting process, you will see a Services tab that lists the services inside the process: the name of the registry key that defines the service, the display name seen by the administrator, and the description text for that service (if present). For example, listing the services in a Svchost.exe process on Windows XP running under the System account looks like this:



For more details on services, see Chapter 4.

Conclusion

In this chapter, we've taken a broad look at the overall system architecture of Windows. We've examined the key components of Windows and seen how they interrelate. In the next chapter, we'll look in more detail at the core system mechanisms that these components are built on, such as the object manager and synchronization.